Finding Counterexamples for the SAG-based Schedulability Analysis

Zhao, Yimi

Award date:
2022

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Interconnected Resource-aware Intelligent Systems Research Group

Finding Counterexamples for the SAG-based Schedulability Analysis

Master Thesis

Yimi Zhao

Supervisors:
Dr. ir. Geoffrey Nelissen
Dr. Mitra Nasri
Srinidhi Srinivasan

Eindhoven, September, 2022

Abstract

Real-time systems are systems whose correct behavior depends not only on their functional correctness, but also temporal correctness. To confirm that timing requirement associated with different tasks are met during the execution of the system, one can perform a schedulability analysis. Nasri et al. proposed to develop a specialized model checker, called schedule-abstraction graph, tailored to schedulability analysis.

The schedule-abstraction graph (SAG) explores the space of all possible scheduling scenarios that could occur in the system. In case of systems with a single-core platform, it guarantees that it does not include any impossible scheduling scenario. To defer the state-space explosion, SAG uses efficient merging techniques and even partial-order reduction. As a result, the SAG is three to height orders of magnitude faster than other exact schedulability analyses that are based on formal method-based analyses such as UPPAAL and scales well in terms of the number of tasks and processors. From these investigations, it is generally concluded that the SAG is an effective way to perform schedulability analysis.

However, the schedule-abstraction graph still has some limitations. One of them is that it does not provide an example of an execution scenario in which timing requirements are violated when it deems a system unschedulable. Therefore, extending the capabilities of the schedule abstraction graph analysis framework with the capability to generate counterexamples to schedulability should help system designers understand why timing violations arise and what changes to the system are required to meet associated timing requirements.

The present report proposed an approach to structurally find counterexamples using the schedule-abstraction graph when a system fails to meet its timing requirements and assists the system developer in finding the appropriate solutions. Our proposed solution is to implement a plan-space algorithm which can start from the system state and proceed backwards, checking at each point whether any potential system states can achieve the causal link onward to reach the initial system state eventually. Consequently, a linear plan created by this algorithm represents the schedule of specific jobs to reach a system state at which a deadline miss occurs.

To implement this plan-space algorithm mentioned above, we need to save/compute extra features while building the schedule-abstraction graph. These features might consume a lot of memory or significantly increase the current schedule-abstraction graph's run time. Therefore, instead of trying to do everything at once, we optimize the process by dividing it in three steps. Each step executes a different algorithm. They manipulate and compute different data. At the end of the third step, a scheduling example that leads to a deadline miss is then produced.

Preface

As time flies, my graduation project journey will come to the end and it has only been possible because of the dedicated support I have received along the way.

My deepest and sincerest gratitude goes to my supervisor Dr. ir. Geoffrey Nelissen for his continuous and invaluable guidance. I would like to thank him for his insight, support, empathy and patience. I also thank Dr. Mitra Nasri for her continuous guidance, encouragement, concern, and ever-cheery disposition throughout the thesis process, which was so important for me. I consider myself one of the luckiest people as their student.

I would like to thank my committee members, Dr. ir. Ion Barosan, PhD student Srinidhi Srinivasan for their valuable comments and suggestion. Especially Nidhi, I am beyond grateful to call you a good friend.

Above all my sincere appreciation goes to my family for supporting and encouraging me throughout this long process, and under their protection, I have been able to enjoy my studies and my life. Especially to our dog, Little Flower, who has been with me since I started school until I am about to complete my master's degree. Your presence in our family brought us so much happiness.

Acronyms

CE	Counterexample
DAG	Directed Acyclic Graph
DM	Deadline Monotonic
EDF	Earliest-Deadline First
JLFP	Job-Level Fixed-Priority
RTA	Response-Time Analyses
SAG	Schedule-Abstraction Graph
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time systems are systems whose correct behavior depends not only on their functional correctness, but also temporal correctness [9]. Applications of real-time systems can be found in medical equipment, aerospace, robotics, and military domains [9]. To check if a real-time system meets its timing constraints, one can use a *schedulability test* [27, 9], whose goal is to determine whether a given set of tasks will always satisfy their timing constraints when they are scheduled by a given scheduling policy on a given platform.

Schedulability tests. There are three categorizes of schedulability tests: (i) *necessary tests*, (ii) *sufficient tests*, and (iii) *exact tests*. If a task set is rejected by a necessary test, there exists at least one scheduling scenario (a schedule generated based on a plausible combination of arrival and execution times of the tasks) in which the timing constraints of the task set are violated. A schedulability test is sufficient if task sets that pass the test always meet their timing constraints when scheduled by the given scheduling policy. Examples of sufficient schedulability tests are the test of Liu and Layland [27] or hyperbolic bound [6] for rate-monotonic scheduling policy. A schedulability test is exact if it is both necessary and sufficient.

Even though schedulability tests are helpful means for verifying temporal correctness, they are insufficient for the design process as a whole since they only result in a yes/no answer. They neither provide examples of execution scenarios in which a timing property was violated, nor they give an insight on the causes of such timing violation. The only category of schedulability tests that can provide a counter example in case of timing violation are those that are based on model checking (typically using UPPAAL model checker) [22, 21, 40, 37, 40] In these tests, each task is modeled as a timed automata. The model checker checks whether a state with a missed deadline can be reached. However, these solutions do not scale well when confronting a large number of tasks or complex task models (e.g., when there is release jitter, execution-time variation, precedence constraints, etc.), or a complex execution platform (e.g., multiprocessor platforms).

Response-time analyses. As an attempt to have a scalable solution that provides more insight about the cause of timing violation, the response-time analysis (RTA) has been introduced [4]. RTA provides an upper bound on the worst-case response time (WCRT) of each task in the task set when tasks are scheduled by a given scheduling policy on a given execution platform. Hence, they enable the system architects to know which of the tasks may miss their timing constraints.

There are two categories of response-time analyses: those that are based on closed-form equations [4, 16, 32, 10] and the schedule-abstraction graph (SAG) based analyses [29, 30, 31]. The former category obtains the worst-case response time by forming a closed-form equation that calculates the response time of each task when it is interfered (and hence delayed) the most by other tasks [4, 16, 32, 10]. In other words, it has a hypothesis about what would be the worst-case scheduling scenario that could result in the WCRT and then use that scenario to obtain the bound of the response time. These analyses provide a tight bound on the WCRT only when that particular scheduling scenario can indeed occur in the system, e.g., for preemptive or non-preemptive sporadic tasks (with non-deterministic arrival pattern) scheduled on a single-core platform [4, 16]. In these cases, if the timing properties are violated, the worst-case scheduling scenario is the scenario in which the timing violation has happened. However, as soon as the task or platform models become more complex, e.g., when tasks are non-preemptive and periodic, have offsets or release jitter, or have precedence constraints, the closed-form equation-based RTA become pessimistic, because the hypothetical scenario they used to derive the WCRT may never occur. As a result, the upper bound they provide might be much larger than the actual WCRT of the task, namely, they will have a large number of false positive as shown by [29, 30, 31, 40, 33]. Moreover, in these cases, they cannot provide a realistic execution scenario in which a timing violation happens.

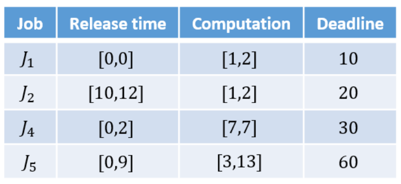
Recently, Nasri et al. [29, 30, 31, 33, 36] proposed a specialized model checker, called schedule-abstraction graph (SAG), tailored to schedulability analysis. Instead of hypothesising a worst-case scheduling scenario, SAG explores the space of all possible scheduling scenarios that could occur in the system. In case of systems with a single-core platform, it guarantees that it does not include any impossible scheduling scenario. To defer the state-space explosion, SAG uses efficient merging techniques and even partial-order reduction [35, 36]. As a result, SAG is 3000 times faster than other exact schedulability analyses that are based on formal method-based analyses such as UPPAAL [40] and scales well in terms of the number of tasks and processors. From these investigations, it is generally concluded that the SAG is an effective way for schedulability analysis. However, despite its thorough search for all possible scheduling scenarios that could occur in the system, it does not yet provide concrete counterexamples in which a timing violation occurs.

In our work, we propose an approach to structurally find counterexamples from the schedule-abstraction graph analysis when the analysis reports a timing violation for a task set.

1.1 System Model and Background

We first introduce our system model and necessary background information to assist in understanding the rest of the thesis. We consider a system with multiple hard real-time periodic tasks run on a single processor.

Job: A periodic task implements a specific functionality which is activated periodically Each instance of a periodic task is called a **job** shown in Figure 1.1. Timing characteristics of a job are typically modeled by a 4-tuple $J_i = \{[r_i^{min}, r_i^{max}], D_i, [C_i^{min}, C_i^{max}], p_i\}$, where r_i^{min} and r_i^{max} are the earliest and latest release times of J_i respectively. This release time interval is caused by a non-deterministic release jitter, e.g., due to the interrupt service routine latency, kernel overheads, etc. As a result, a job may possibly be released at r_i^{min}, and it will certainly have been released at r_i^{max}. Furthermore, D_i denots the relative deadline, C_i^{min} denotes the best-case execution time (BCET), and C_i^{max} denotes the worst-case execution time (WCET) of the job J_i. Finally, p_i denotes the job-level fixed priority. We assume that a numerically smaller value of p_i implies higher priority [31].



Job	Release time	Computation	Deadline
J_1	[0,0]	[1,2]	10
J_2	[10,12]	[1,2]	20
J_4	[0,2]	[7,7]	30
J_5	[0,9]	[3,13]	60

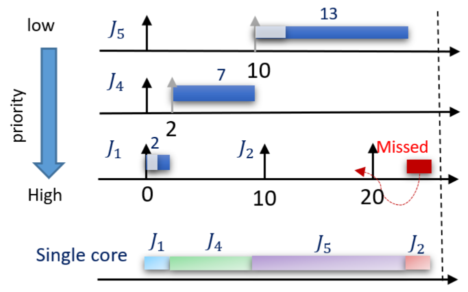


Figure 1.1: A basic example for illustrating different properties of job J_i

Schedulability A task set is schedulable if and only if all jobs of all tasks meet their deadlines D_i, i.e., $R_i \leq D_i, \forall i$, where R_i is the worst-case response time (WCRT) of the job J_i. The response time of a job J_i is the length of the interval starting from the (earliest) release time and ending at the finish time shown in Figure 1.2.

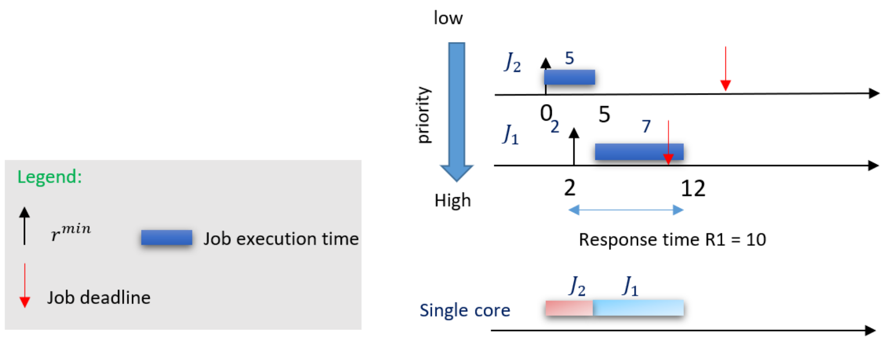


Figure 1.2: A basic example for explaining the response time of the job, in this example J_2 's response time is $12 - 2 = 10$.

Schedulability test. A test that checks whether a set of tasks (or jobs) could be scheduled feasibly by the given scheduling policy, i.e., all jobs will always meet their deadlines under any execution scenario. A schedulability test can be necessary, sufficient, or exact, as discussed in the previous section. The test is sufficient if, when it answers "Ture", all deadlines will be met. A necessary test is when it answer is "False", there must be a situation where deadlines will be missed. The test is exact if it is both sufficient and necessary that a "Ture" answer means that the task set is schedulable, and a "False" answer implies that the task set is not schedulable.

Figure 1.3 shows the three types schedulability tests and how they compare to each other.

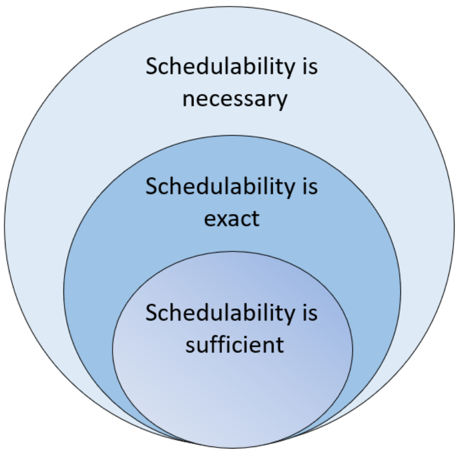


Figure 1.3: Three schedulability tests are compared to one another.

Scheduling policy: Scheduling is the process of deciding which task must be executed on what resource at what time. A scheduling policy is an algorithm that takes those decisions at runtime. It can be classified as:

- **work-conserving** or **non-work-conserving**. A Work-conserving scheduling policy does not keep the processor idle while there is a ready job in the system. On the other hand, a non-work-conserving scheduling policy may decide to keep the processor idle even when there are jobs that are waiting to be executed. Therefore, a non-work-conserving policy may result in idle times even though there are ready jobs.

  We can also categorise the scheduling policy as

- **preemptive** or **non-preemptive**. A preemptive scheduling policy forces a lower-priority tasks (or jobs) to yield the processor when a higher-priority job is ready to be executed even though they might not have completed yet. A non-preemptive scheduling policy will always execute jobs to completion once they start executing shown in Figure 1.4.

- **Job-level fixed-priority (JLFP) scheduling policy:** In a JLFP policy, each job has a static priority that is known before the system starts. This policy includes a broad range of widely known and implemented shceduling policies such as task-level fixed-priority policy (FP), the earliest-deadline first (EDF) policy, etc. In FP scheduling, all jobs of a task inherit the priority of the task, while in EDF scheduling, the priority of each job is equal to its absolute deadline. Other common schemes are deadline monotonic, where the priority depends on the relative deadline of the task (the smaller the deadline, the higher the priority), and rate monotonic, where the priority is monotonic to the period of the task.

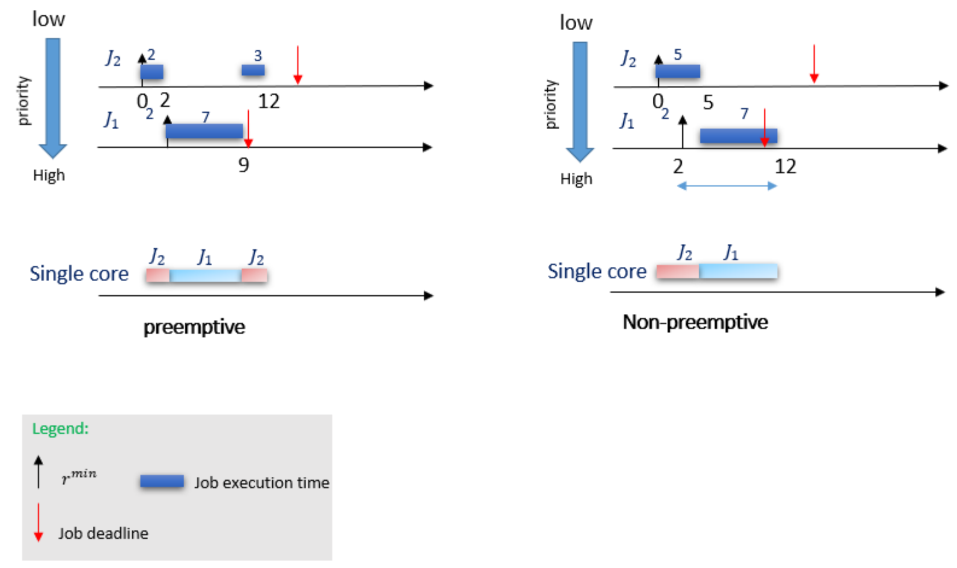


Figure 1.4: It is a basic example for illustrating the preemptive and non-preemptive features of the Scheduling policy. In preemptive mode, J_1 is released when J_2 is executing, and the processor stops the lower priority job J_2 and switches to run the higher priority job J_1. Even if job J_1 is released while J_2 is executing in non-preemptive mode, the processor can start J_1 until J_2's computation time is complete.

Our system model: In this project we consider **non-preemptive job-level fixed-priority scheduling policies** to schedule a set of jobs on a **single core** platform. In our analysis, we deem a job set schedulable if and only if all jobs meet their deadline in every possible execution scenario of the jobs (defined in Definition 1.1.1). A system is thus deemed unschedulable if there is at least one execution scenario in which there is a job with a response time larger than its deadline.

Definition 1.1.1 (From [29]). An *execution scenario* $\gamma = (C, r)$ for a set of jobs J is a sequence of execution times $C = \langle C_1, C_2, \ldots, C_m \rangle$ and release times $r = \langle r_1, r_2, \ldots, r_m \rangle$ such that, for each job J_i, $C_i \in [C_i^{min}, C_i^{max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$.

In each execution scenario, the scheduler may take different scheduling decisions, i.e., may decide to dispatch jobs in a different order. Note however, that there is only one possible schedule per execution scenario. One way to implicitly explore all possible execution scenarios of a job set is by using the *schedule-abstraction graph*.

1.2 Challenges and Motivations

By providing a collection of jobs, a set of resources, and a assigned of underlying scheduling regulations, the schedule-abstraction graph performs the response time analysis. It determines the worst-case response time of every job by exploring the space of possible decisions that may be taken by a given a scheduling policy at run-time. Schedule-abstraction graph is used as an **exact** schedulability test for job sets that are scheduled on a uni-processor platform using a JLFP policy [29]. This schedule-abstraction graph is defined as a directed-acyclic graph $G = (V, E)$, where V is a collection of system states, and E is a collection of labelled edges shown in Figure 1.5. Every path from an initial state to a destination state in this graph represents a sequence of scheduling decisions for different sets of execution scenarios. The underlying scheduling technique of the schedule-abstraction graph makes a job-dispatch decision to dispatch a job from the source vertex to the destination vertex of the edge.

A path is defined by the set of jobs that were already dispatched and the system state represents the availability interval of the processing element after a set of jobs (on any path that reaches to that state) have been dispatched on the platform. More formally, a state v_p records the availability of the processor after executing a set of jobs. This availability is represented as an interval $\left[A_1^{min}, A_1^{max}\right]$, where A_1^{min} and A_1^{max} represent the earliest time and the latest time at which the processor will become available to execute new jobs. Alternatively, A_1^{min} and A_1^{max} may be interpreted as the earliest and latest time at which the set of jobs that are on any path from v_1 to v_p will complete their execution.

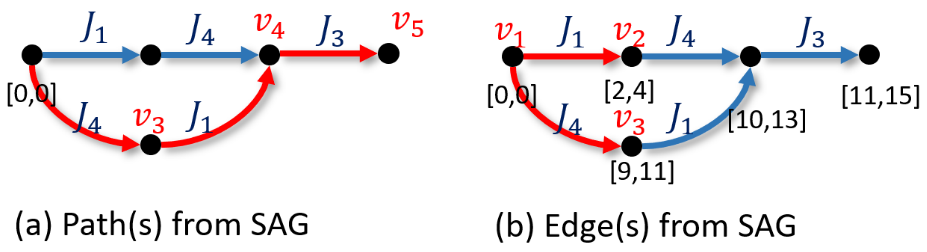
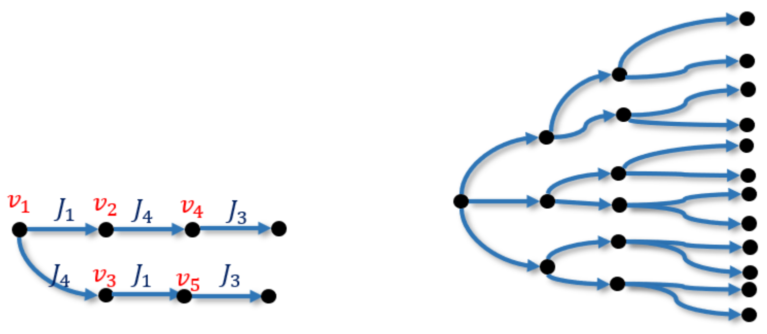


Figure 1.5: A basic example for the schedule abstraction graph. A path from (a) represents a series of scheduling decisions, an edge from (b) represents one scheduling decision. J_1 is dispatched at state v_1, the interval $[2, 4]$ is possible finished time and certainly finished time of J_1

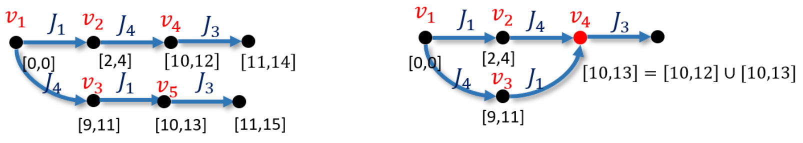
The schedule-abstraction graph is constructed in two phases: expansion and merging phases. the expansion phase expands the not-yet-completed paths in the graph by appending a new vertex for any job that might be *dispatched next* by the scheduling policy.

However, the size of the graph may grow exponentially by considering all potential combinations of scheduling decisions as shown in Figure 1.6. To avoid such a state-space explosion, Nasri et al.[31] introduced a merging phase that restricts the growth of the graph by combining the terminal vertices of paths that contains the same set of dispatched jobs and have intersecting availability intervals. When merging two states, a union of the finish-time intervals in these states are stored as the finish-time interval of the merged state. For example, as depicted in Figure 1.7, the $A_1^{min}(v_4) = \min\{A_1^{min}(v_4), A_1^{min}(v_5)\}$ and $A_1^{max}(v_4) = \max\{A_1^{max}(v_4), A_1^{max}(v_5)\}$. In line with merging phase, the label of the merged state is updated.


v_1 J_1 v_2 J_4 v_4 J_3
J_4 v_3 J_1 v_5 J_3

(a) Expansion phase of SAG (b) Expansion phase of SAG

Figure 1.6: A basic example for the expansion phase of schedule abstraction graph. Fig (a) is an illustration of the expansion phase. The decision is made to dispatch jobs 1 and 2 as the next jobs that can be dispatched in state 1. Dispatching job 1 will reach/create a state 2 (v_2). As a result, Path P is expanded from state 1(v_1) to state 2 (v_2). The size of a graph can be quite considerable, as shown in Fig (b).



(a) Expansion phase of SAG (b) Merge phase of SAG

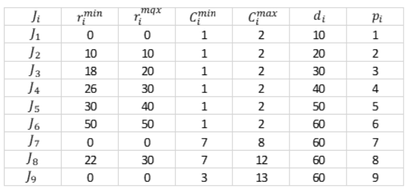
Figure 1.7: State 4 in Fig (b) explains the merge phase by merging v_4 and v_5 from the Fig (a) as a new v_4 which has a updated finish time interval.

As stated in Definition 1.1.1, various alternative release times and execution times of a set of jobs yield different execution scenarios. Regarding a deadline miss situation, an execution scenario that leads to such deadline miss can be defined as definition 1.2.1.

Definition 1.2.1. A deadline miss execution scenario for a job $J_x \in J$ is a sequence of execution times $C^{miss} = <C_1, C_2, ..., C_m>$ and release times $r^{miss} = <r_1, r_2, ..., r_m>$ such that, for each job $J_i \in J$, $C_i \in [C_i^{min}, C_i^{max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$, and J_x finishes its execution at time t such that $t > D_i$.

Example 1.2.1. Consider a job set shown in Figure 1.8. In this figure, we demonstrate a graph with no merge. We found that when job J_2 is dispatched next after state v_4 (resulting in the state v_6), it misses its deadline which was at time 20. In fact, every path that follows state v_4 results in deadline misses (for one or more jobs). This is due to the execution of a lower-priority job J_9 which can be dispatched at time $t = 9$ in scenarios in which the processor becomes available at time 8 or 9, i.e., one time unit earlier than job J_2 is released.

Moreover, we see that another deadline miss may occur at state v_{25}. Indeed, the scheduling policy decided to dispatch the job J_4 after state v_{22}. This results in $LFT_4 = 41$, which exceeds the deadline of J_4. These observations clearly indicate that the $[J_1, J_7, J_9, J_2...]$ and $[J_1, J_7, J_2, J_9, J_3, J_8, J_4, J_5, J_6]$ job dispatching sequence may cause an unschedulable outcome.



J_i	r_i^{min}	r_i^{mqx}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	0	0	1	2	10	1
J_2	10	10	1	2	20	2
J_3	18	20	1	2	30	3
J_4	26	30	1	2	40	4
J_5	30	40	1	2	50	5
J_6	50	50	1	2	60	6
J_7	0	0	7	8	60	7
J_8	22	30	7	12	60	8
J_9	0	0	3	13	60	9

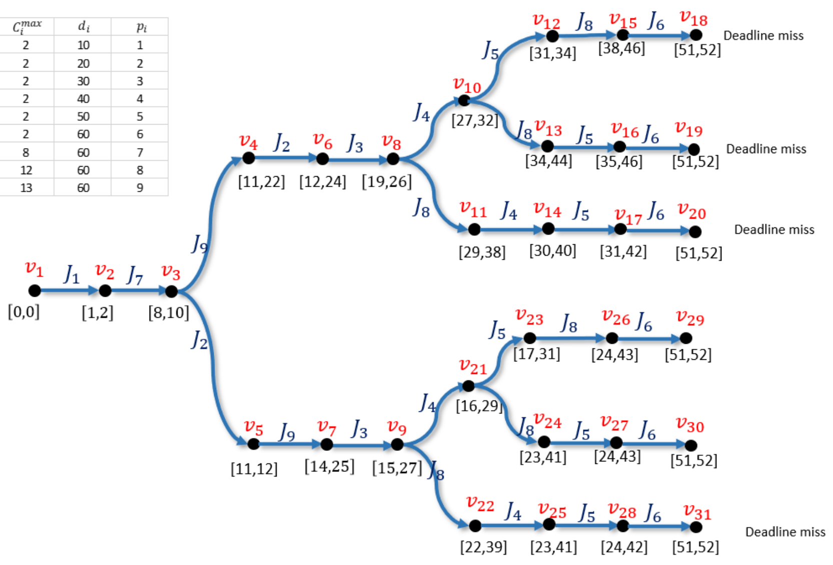

v_1 J_1 v_2 J_7 v_3
[0,0] [1,2] [8,10]
J_9 J_2
v_4 J_2 v_6 J_3 v_8
[11,22] [12,24] [19,26]
J_4 J_8
v_{10} [27,32]
v_{12} J_8 v_{15} J_6 v_{18}
J_5 [31,34] [38,46] [51,52] Deadline miss
J_8 v_{13} J_5 v_{16} J_6 v_{19}
[34,44] [35,46] [51,52] Deadline miss
v_{11} J_4 v_{14} J_5 v_{17} J_6 v_{20}
[29,38] [30,40] [31,42] [51,52] Deadline miss
v_5 J_9 v_7 J_3 v_9
[11,12] [14,25] [15,27]
J_4 J_8
v_{21} [16,29]
J_5 v_{23} J_8 v_{26} J_6 v_{29}
[17,31] [24,43] [51,52]
J_8 v_{24} J_5 v_{27} J_6 v_{30}
[23,41] [24,43] [51,52]
v_{22} J_4 v_{25} J_5 v_{28} J_6 v_{31}
[22,39] [23,41] [24,42] [51,52] Deadline miss

Figure 1.8: This figure demonstrate all the execution scenario without apply merging technique. In total four sub paths have deadline miss result, path $[J_1, J_7, J_9, J_2, J_3, J_4, J_5, J_8, J_6]$, $[J_1, J_7, J_9, J_2, J_3, J_4, J_8, J_5, J_6]$, $[J_1, J_7, J_9, J_2, J_3, J_8, J_4, J_5, J_6]$ and $[J_1, J_7, J_2, J_9, J_3, J_8, J_4, J_5, J_6]$, respectively

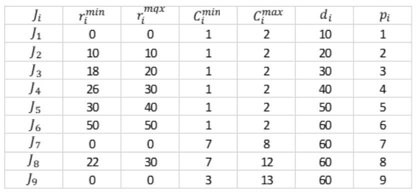
The sequence of scheduling decisions that led to a deadline miss is clear in the previous example (Example 1.2.1). However, in order to avoid over-consumption of the memory and slow down the growth of the graph (to reduce the number generated states), the schedule abstraction graph analysis merges paths which have the same set of jobs and have overlapping availability intervals. After applying the merge phase, the schedule-abstraction graph of figure 1.8 would in fact become like the schedule abstraction graph reported in figure 1.9.

It is still clear in the graph of figure 1.8, that a deadline miss happens. Unfortunately, the paths shown in figure 1.9 cannot tell the exact sub-path leading to the deadline miss result. Consequently, we cannot conclude what job dispatching sequence leads to the deadline miss. Collecting the set of paths in the schedule-abstraction graph that result in unschedulable outcomes is hard. Thus determining a set that leads to a deadline miss will be our **challenge 1**.

To solve this challenge, we must further investigate what techniques can be used to find which sub-paths of a graph lead to a deadline miss.

Example 1.2.2. A simple deadline miss example is shown in Figure 1.10.

Our aim is to get a general picture of the deadline miss execution scenarios as the counterexample. Assume we identified the specific path claimed as unschedulable e.g,. path $[J_1, J_4, J_5, J_2]$ in Example 1.2.2. However, execution scenarios with a deadline miss are not easy to demonstrate due to how to assign the release time, start time, and execution time to the job is uncertain. In the example trace, it appears that job J_2 missed its deadline. The execution scenarios of Example 1.2.2 that result in deadline misses are not unique. Six different execution scenarios derived from the same unschedulable path are presented in Figure 1.10. This uncertainty leads to our **challenge 2** that is how to assign the release time, start time and execution time for the execution scenario that causes a missed deadline.



J_i	r_i^{min}	r_i^{mqx}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	0	0	1	2	10	1
J_2	10	10	1	2	20	2
J_3	18	20	1	2	30	3
J_4	26	30	1	2	40	4
J_5	30	40	1	2	50	5
J_6	50	50	1	2	60	6
J_7	0	0	7	8	60	7
J_8	22	30	7	12	60	8
J_9	0	0	3	13	60	9

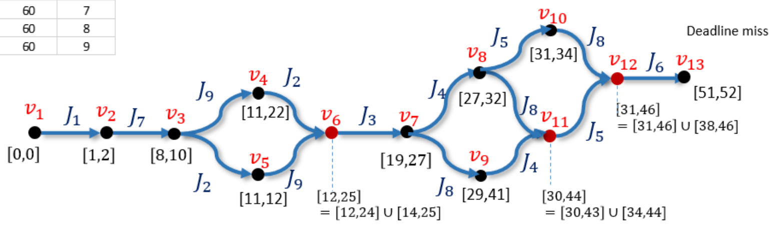
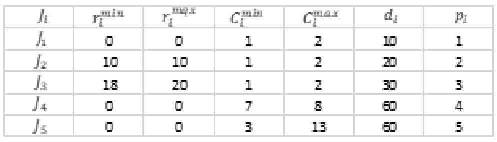


Figure 1.9: The schedule abstraction graph is depicted after involving the merge mechanism. The state v_6, v_{11} and v_{12} are the merge vertex. The labels in the merge vertex are calculated by formula: $[e_p, l_p] \leftarrow [e_p, l_p] \cup [e_q, l_q]$.



J_i	r_i^{min}	r_i^{mqx}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	0	0	1	2	10	1
J_2	10	10	1	2	20	2
J_3	18	20	1	2	30	3
J_4	0	0	7	8	60	4
J_5	0	0	3	13	60	5

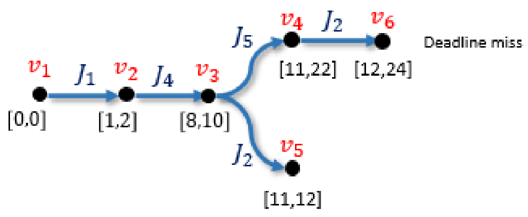
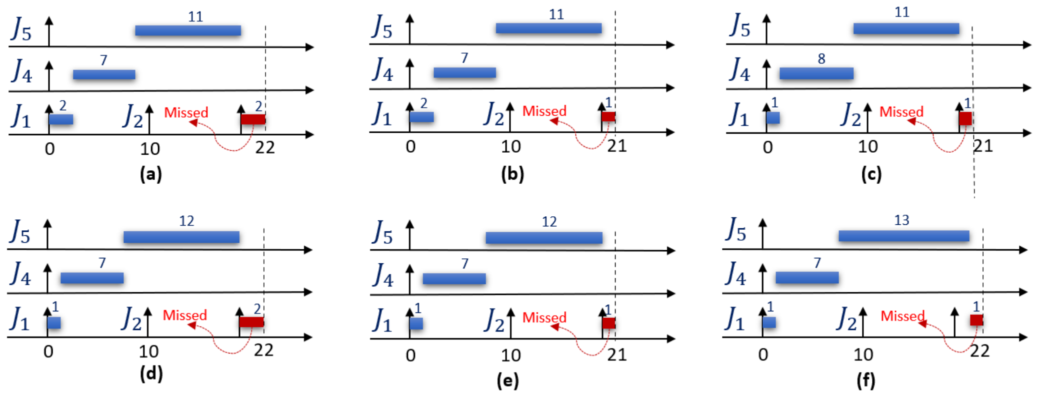


Figure 1.10: Example of a job j_2 missed it deadline



(a) J5 11; J4 7; J1 2; J2 Missed 2; 0 10 22
(b) J5 11; J4 7; J1 2; J2 Missed 1; 0 10 21
(c) J5 11; J4 8; J1 1; J2 Missed 1; 0 10 21
(d) J5 12; J4 7; J1 1; J2 Missed 2; 0 10 22
(e) J5 12; J4 7; J1 1; J2 Missed 1; 0 10 21
(f) J5 13; J4 7; J1 1; J2 Missed 1; 0 10 22

Figure 1.11: Part (a) of the figure shows one execution scenario, the computation time of job J_1, J_4, J_5, J_2 are 2, 7, 2 and 11 respectively. Similarly, part (b) - (f) of the figure demonstrated other execution scenarios lead to deadline miss.

To make matters worse, the tool that implements the schedule abstraction graph analysis optimizes the analysis in order to optimize memory usage. Therefore, it removes states from the memory when they do participate to the analysis anymore. For example, the state S1 in the dot graph (figure 1.12) will be deleted from memory after the states S2, and S3 have been generated. As a result, our third challenge, **challenge 3**, is to figure out how to make a counterexample when the existing implementation of the scheduling abstraction graph analysis does not keep track of the whole graph and thus reduces the amount of accessible information.

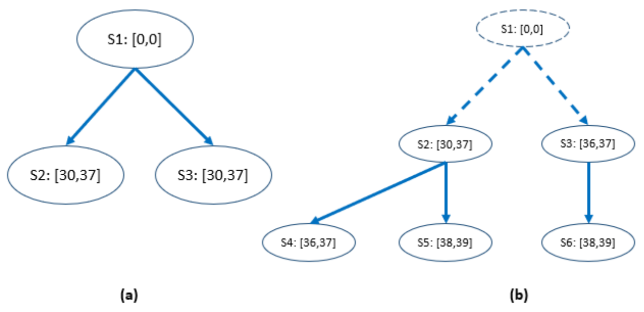


Figure 1.12: Part (a) of the figure shows an original state S1, after dispatching job j_i the new state S2 and S3 will create respectively. When the system obtained the new states S2 and S3, the previous state S1 is erased from memory shown in part (b).

We aim to deliver a counterexample as the end goal of the project that explains why a job deadline was missed. A counterexample based on the schedule-abstraction graph should avoid being too expensive but still provide a detailed execution scenario, including the release time, starting time and execution time of the jobs involved in the deadline miss execution scenario. To construct a counterexample of an execution scenario, we might need to save/compute the extra features while building the graph, these features, however, might consume a lot of memory or significantly increase the current schedule-abstraction graph's working time. This brings us to our final challenge, **challenge 4**.

The current schedule-abstraction graph does not provide a counterexample when there is a deadline miss. Building a counterexample is a complex task due to the challenges we mentioned previously, and since the project is time-constrained, the thesis scope aims to answer the following research question shown in the next section.

1.3 Research Question

As discussed in the previous section, the schedule abstraction graph checks whether there is an execution scenario in which the system reaches a state with a missed deadline. Our main goal in this project is to address the following question: How can we generate an explaining example of an execution scenario that leads to such deadline miss using the schedule-abstraction graph? To answer this general question, we refined it into four specific research questions (RQs).

RQ 1 How can we identify a set of paths in the schedule abstraction graph that leads to a deadline miss?
The first question involves collecting and analysing a path representing a sequence of scheduling decisions that led to a job missing its deadline. The **challenge 1** indicated in the preceding section will be addressed by answering RQ 1.

RQ 2 How can we generate the counterexample from a set of paths in the schedule abstraction that lead to a deadline miss?
Using the set of paths found by answering the previous research question, we must devise a technique to generate one (or a set of) counterexample(s) that explicitly show(s) how a deadline is missed based on a certain sets of events and scheduling decisions. RQ 2 focuses on addressing the **Challenge 2**.

RQ 3 Is there already enough information available in the current implementation of the SAG framework to build a counterexamples? If not, what is the minimal amount of information that should be added to the graph?

RQ 3 is intended to identify how to acquire adequate information for generating the counterexample due to the difficulties indicated in **Challenge 3**.

RQ 4 What techniques can be used to efficiently implement the counterexample generation feature to the existing implementation of the schedule-abstraction graph framework?
We need to implement our new automatic counterexample generation technique in the existing schedule abstraction graph framework. This built-in tool must not be a costly procedure that consumes a lot of memory or significantly increases the current tool's working time. Addressing this last research question allows us to overcome **Challenge 4**.

1.4 Organization

The organization of this paper is laid out as follows. In Chapter 1, we introduce the topic of this thesis, the related challenges, motivations and the specific research questions that will be addressed during the thesis. We study the state of the art in Chapter 2. Our initial approach for generating counterexamples using the schedule-abstraction graph is described in Chapter 3. Chapter 4 explains the existing implementation of the current schedule-abstraction graph to provide a high-level overview of the analysis tool and thus make the implementation challenges apparent. Chapter 5 presents the thesis planning, risk analysis, and the next steps. Finally, in Chapter 6, we conclude this report.

Chapter 2

Related Work

This section explains the model checking, recent work in backward chain algorithms and a brief overview of how to generate a counterexample.

2.1 Model checking

Considerable effort has been expended on the development of model-checking tools for analysis of real-time systems because these can verify whether or not the RTS meets the specification (timing) requirement. When a model falsifies a specification, one possibility for model checking is to generate a counterexample. The counterexample is an error trace. By analysing it, the user can locate the source of the error[1]. A.P. Kaleeswara et al. present a literature review about counterexample explanation. This survey provides a conceptual model for the model checking and counterexample explanation shown in Figure 2.1 [24]. Thus, in this chapter, we will follow the model checking and counterexample explanation process to discuss the state of the art in research on these topics.

So far, effective model checking tools for plan/schedule analysis have been developed, and the table below summarizes a comparison between different model checking technologies.

	Forward Looking	Backward Chain
Terminology	Linear Temporal Logic	Landmark Discovery
Terminology	Computational Tree Logic	Partial Order Planning

Table 2.1: Comparison of planning/scheduling implementations

In 1977, the linear temporal logic was proposed by Pnueli [34]. This approach is applied to the system with linear time-varying properties as the system proceeds through a sequence of discrete states and expresses properties on a temporal level (the trace/timeline of the system). The end goal can be achieved by satisfying the conditions over a sequence of states. One major difference between linear temporal logic (LTL) and computation tree logic (CTL) is how the model expresses time. In the computation tree logic, the time branches into a tree-like structure for all related execution scenarios. Therefore, CTL explicitly introduces path quantifiers [18]. After defining the set of formal properties of the system, the model checker can verify whether the model satisfies the requirements. These studies hinted that the problem of schedulability analysis, i.e., determining whether a given task set meets its deadline constraints, can be achieved by the model checker toolbox.

Uppaal is a toolbox for modelling and verifying real-time systems that have been applied in schedulability analysis. The query language of UPPAAL, used to specify properties to be checked, is a subset of CTL (computation tree logic) [5]. To facilitate modelling and debugging, the UPPAAL model checker can automatically generate a *diagnostic trace* that explains why a property is (or is not) satisfied by a system description. The diagnostic traces generated by the model-checker can be graphically visualized using the simulator [26].

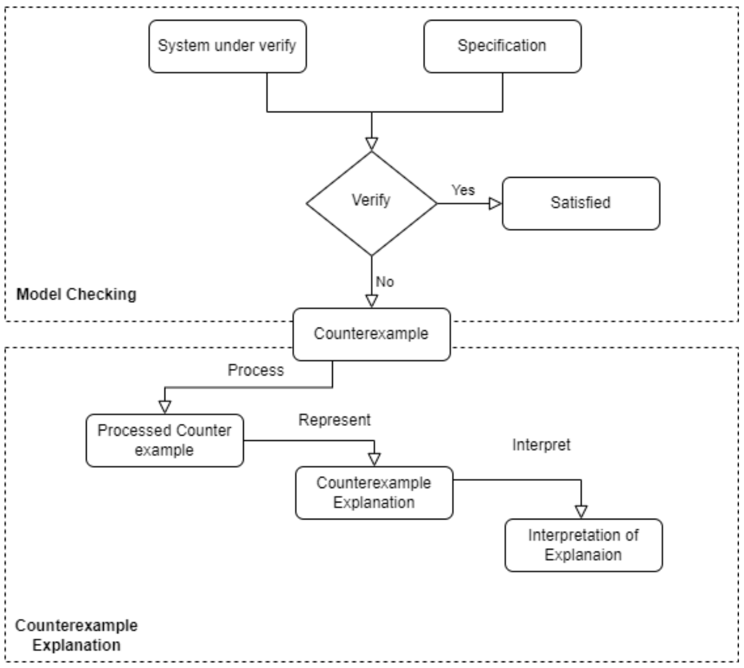

System under verify
Specification
Verify
Yes
Satisfied
No
Model Checking
Counterexample
Process
Processed Counter example
Represent
Counterexample Explanation
Interpret
Interpretation of Explanaion
Counterexample Explanation

Figure 2.1: An Overview of model checking and counterexample explanation.

Many study cases by applying UPPAAL strongly prove that UPPAAL model checker can be performed for schedulability analysis [28, 13, 23, 7, 8]. Al-Bataineh et al. proposed a comparative study of decision diagrams for RTS model checking. The experiment focuses on three different state-of-the-art real-time systems model checkers: UPPAAL, RED, and Rabbit. It is concluded that for timed systems with complex modelling details, UPPAAL is highly recommended as it has richer expressiveness in modelling systems than Rabbit and RED [2].

2.2 Reachability-based schedulability analysis of RTS

The schedulability analysis can be defined as reachability analysis on the state space of a real-time system [19]. In this section, we describe the different reachability-based mechanisms used to model the deadline missed scheduling problem.

2.2.1 Linear Hybrid Automaton

Linear Hybrid Automata (LHA) are finite-state automata extended with real-valued variables. Transitions between discrete states may be conditional on the values of these variables and may assign new values to variables. The LHA can be used to model a real-time system [39]. More specifically, Sun et al.[37] put forward LHA to represent the tasks and the scheduler. In their model, the counterexample can be discovered from the error locations in the automata where the jobs missed the deadline. However, when the number of sporadic tasks grows beyond 7, and for more than 4 processors, the tool developed by Sun et al.[37] cannot terminate on current desktop computers due to the increased complexity.

2.2.2 Timed Automata

To perform a schedulability test for Non-Preemptive task sets, Yalcinkaya et al. mapped the schedulability problem to the reachability problem in timed automata (TA) [3] by using TA extensions available in UPPAAL [40]. Regrading the result (TA for sporadic tasks) produced by Yalcinkaya et al., the counterexample can be found from the `Start` state to the `Miss` state. Although the counterexample is generated in this case, a considerable state of space is also created. Thus, it cannot solve big problems because it takes a very long time to reach the state where the deadline is missed. A further issue is that UPPAAL does not provide an interval of execution time and release time, so a job would still miss its deadline even if it is a sub-interval of the original intervals. Thus we are only able to obtain one counterexample generated by UPPAAL.

Other researchers [13, 28] have tried to implement a schedulability test in UPPAAL. For example, David et al. applied the stopwatch automata [11] in UPPAAL to solve the uncertainty of release time, starting time and the execution time of jobs. If the query is not schedulable, their model shows that a counterexample has been established in the over-approximation. The system, however, **may** still be schedulable. This ambiguity renders the TA reachability problem undecidable [40].

2.2.3 Schedule-abstraction Graph

Recently, Nasri et al. [29] [31] have made significant advances in *global non-preemptive* schedulability tests. The approach explores all possible schedules that may potentially happen at runtime. The experimental results showed that the schedule-abstraction graph was able to identify more task sets than the TA-based analyses within the one-hour time limit [40]. Further, Nasri et al. demonstrated the schedule-abstraction graph is able to scale to much bigger systems which can scale to 30+ periodic tasks with thousands of jobs [29], and this approach reduces the runtime. For example, it requires less than 10 seconds, while UPPAAL needed tens of minutes to reach the same result.[31]. The current schedule-abstraction graph will terminal when it reaches a deadline miss state. Therefore, it cannot provide a counterexample directly. As a result of its efficient, interval-based state-space exploration approach, which is adapted to the schedulability problem, unlike UPPAAL [40]. Building the counterexample solution on top of the salable schedule-abstraction graph should provide more than one counterexample.

2.3 Backward Chain

Research has tended to focus on model checking rather than counterexample explanation. The recent evidence [24] suggests that there is no general agreement on explaining counterexamples generated by model checker. Therefore the aim of our research is to extend current knowledge of generating a counterexample on the schedule-abstraction graph. The information presented in the counterexample should not overwhelm the system designer. A job set is marked unschedulable once an execution scenario with a deadline miss is found. Thus the path P from the initial state $v1$ to a deadline missed state v_p indicates the trace of counterexample.

As we discussed earlier, LTL and CTL require forward looking condition by investigating a given state, we can assert about what will happen in the future. However, a backward chain condition is a symmetric version where we investigate a certain state and we can guarantee something has eventually happened in the past. As mentioned by Rohit and Masesh [12], the counterexample checking algorithm proposed by Clarke et al. [14, 15] starts from the last state of the counterexample and proceeds backward. Karpas, Erez, et al. [25] developed a new reasoning mechanism for the system about what must be achieved and when it must be achieved. In their research, the goal propositions as the starting state for back chaining and stopped back chaining as soon as a known initial state was derived. Building a sequence of actions backwards until it reaches the initial state requires a deeper understanding of how *plan-space planing* works.

To address this question, we studied *plan-space planing*. This planning is considered as two independent options:

1. the choice of actions, and
2. the order of the chosen actions so as to achieve the goal.

The proposed concept of solution to the *plan-space planing* problem namely *partial-order plans* [20]. A partial plan can be used as any subset of job-dispatching sequence that leads to the deadline miss state. R Trinquart [38], an authority on Analyzing Reachability Within Plan Space presents a comprehensive structure developed to analyze reachability within the plan space using partial order planners. Thus, it is generally concluded that the partial order plan algorithm may generate a counterexample on the schedule-abstraction graph.

Summary: So far, the partial order plan algorithm has never been applied to generate a counterexample for the schedulability analysis. For a DAG schedule graph, building and explaining a valid counterexample can start from the last state and proceed backwards. This is because they are checking at each point whether any of the potential states can exhibit counterexample from that point onward.

Chapter 3

Implementation of the Schedule-abstraction graph

The following chapter explains the implementation of the current schedule-abstraction graph (SAG). This SAG implementation is aimed at providing a schedulability test for a set of non-preemptive jobs executed on a uni-processor/multiprocessor platform by a JLFP scheduling policy. The detailed installation and configuration of the tool can be found in appendix A.

3.1 Schedule-abstraction graph

In this section, we describe how the schedule-abstraction graph works. By following the breadth-first approach, Nasri et al. [29, 31] built the SAG. The schedule-abstraction graph is constructed in two phases: expansion and merging phases. The expansion phase expands the not-yet-completed paths in the graph by appending a new vertex for any job that might be *dispatched next* by the scheduling policy. A job Jj can be dispatched next after path P if its earliest start time EST_j is not later than its latest start time LST_j, i.e., if

$$EST_j \leq LST_j \tag{3.1}$$

Definition 3.1.1. (From [31]) Job J_j is a direct successor of v_p only if Inequality 3.1 holds.

However, the size of the graph may grow exponentially by considering all potential combinations of scheduling decisions. To avoid such a state-space explosion, Nasri et al.[31] introduced a merging phase that restricts the growth of the graph by combining the terminal vertices of paths that contains the same set of dispatched jobs and have intersecting availability intervals.

Definition 3.1.2. (From [31]) Two system states v_p and v_q can be merged to one system state if $J^P = J^Q$ and $\left[A_1^{\min}(v_p), A_1^{\max}(v_p)\right] \cap \left[A_1^{\min}(v_q), A_1^{\max}(v_q)\right] \neq \emptyset$.

The labels in the merge vertex are calculated by formula:

$$\left[A_1^{\min}(v_p), A_1^{\max}(v_p)\right] \leftarrow \left[A_1^{\min}(v_p), A_1^{\max}(v_p)\right] \cup \left[A_1^{\min}(v_q), A_1^{\max}(v_q)\right] \tag{3.2}$$

v_p and v_q are merged by updating v_p using Equation. After that, we redirect all incoming edges of v_q to v_p and remove v_q from the system state set.

3.2 High-level overview of the SAG tool

Nasri and Brandenburg [29] initially proposed the SAG tool in 2017 for non-preemptive task response-time analysis on single-core platforms. After that, the tool is extended to cover any global job-level fixed-priority on the multi-core systems [31]. The SAG performs both the *exact* schedulability analysis and response-time analysis. The analysis is an *exact* schedulability analysis. This means that there is no execution scenario in a job set that is deemed schedulable by the analysis that could result in a deadline miss[35]. The different types of options the tool supports are shown below:

Time mode: The time mode includes two options: (1) dense-time (2) discrete-time. In our experiment, we chose the discrete-time model.

Time limit: This mode provides an option for handling timeout. The user can specify the maximum CPU running time (sec). *Zero* means no limit time.

Precedence: This mode is to support precedence constraints.

Continue after deadline miss: We can set the default to off to end the analysis on the first missed deadline. This appears to be most appropriate to build a counterexample to schedulability.

Checking whether those time qualities are respected is one thing, but it is much more useful if the time properties are not respected, and we can provide some example to illustrate under what scenario the deadline miss happens. The goal of this thesis is to create a counterexample for the current SAG uni-core system. For the uni-core system, a non-preemptive scheduling policy establishes an efficient way to utilize the platform, requiring significantly less complexity to analyze the response time than a preemptive scheduling policy. Therefore, the scope of this project is find one counterexample based on single-core, job level fixed priority and non-preemptive and work-conserving scheduling policy.

Figure 3.1 illustrates a high-level overview of SAG (focusing on uni-core systems).

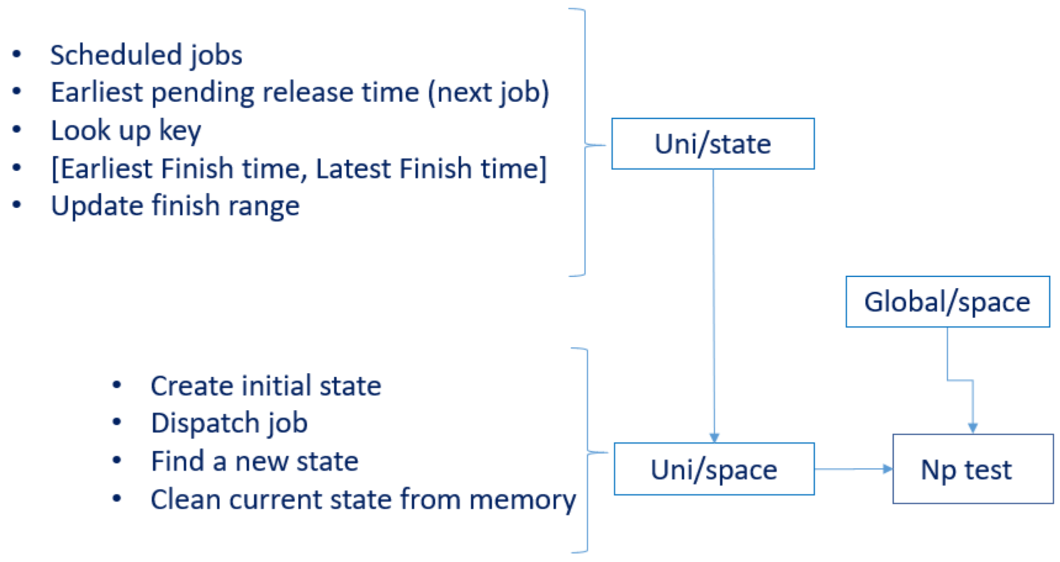


Figure 3.1: high-level overview of the SAG

3.3 Analysis results

To test the correctness of the code, we first applied the unit test suites as follows:

```
./runtests
```

The results are listed below:

```
[doctest] doctest version is "1.2.6"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases:     60 |     60 passed |      0 failed |      0 skipped
[doctest] assertions:    556 |    556 passed |      0 failed |
[doctest] Status: SUCCESS!
```

As we observed, all 60 cases were passed. Thus we are able to run the tool with the `fig1c.csv` example. The import data for the NP-schedulability-analysis tool has six columns, namely task ID, job ID, starting time, computation time, deadline, and priority.

Task ID	Job ID	Arrival min	Arrival max	Cost min	Cost max	Deadline	Priority
1	1	0	0	1	2	10	1
1	2	10	10	1	2	20	2
1	3	20	20	1	2	30	3
1	4	30	30	1	2	40	4
1	5	40	40	1	2	50	5
1	6	50	50	1	2	60	6
2	7	0	0	7	8	30	8
2	8	30	30	7	7	60	9
3	9	0	0	3	13	60	7

Table 3.1: The import CSV data for performing the schedulability-analysis

The corresponding command for running the tool with the `fig1c.csv` example is listed below:

```
./nptest examples/fig1c.csv
```

The NP Schedulability result of the `fig1c.csv` example is demonstrated in below list.

```
examples/fig1c.csv,  1,  9,  11,  11,  1,  0.000043,  4.628906,  0,  1
```

We can conclude that the example `fig1c.csv` is schedulable (i.e., the tool could prove the nine jobs in the job set have no deadline misses). Eleven states and edges were created to build the schedule abstraction graph during the analysis. The CPU running time is 0.000043 seconds, and the memory usage is 4.628906 megabytes.

By installing the graph tool, we are able to generate a dot graph to visualize the full schedule abstraction graph created during the analysis.

```
sudo apt install graphviz
```

Generating a graph requires a lot of CPU running time and memory occupation. The default setting is not producing the dot graph automatically. Thus it requires us to enable the depicting graph option in the `config.h` file first, and the dot graph for this example is depicted below, generated by the following command:

```
dot -Tps fig1c.dot -o fig1c.pdf
```

A part of the dot graph resulting from the analysis of the input job set *fig1c* is depicted in Figure B.1. In this graph, one of the node is labelled $S3 : [4, 15]$ and the incoming edge is labelled *T3 J9* meaning there is an execution scenario where the job J_9 of task T_3 finishes its execution within time interval $[4, 15]$.

ER stands for the earliest release time for the jobs that haven't been scheduled yet. In state S3, the jobs left to be scheduled are $J_7, J_2, J_3, J_4, J_8, J_5, J_6$ and for instance, J_7 is released at time 0. Thus $ER = 0$ as visible at state S3.

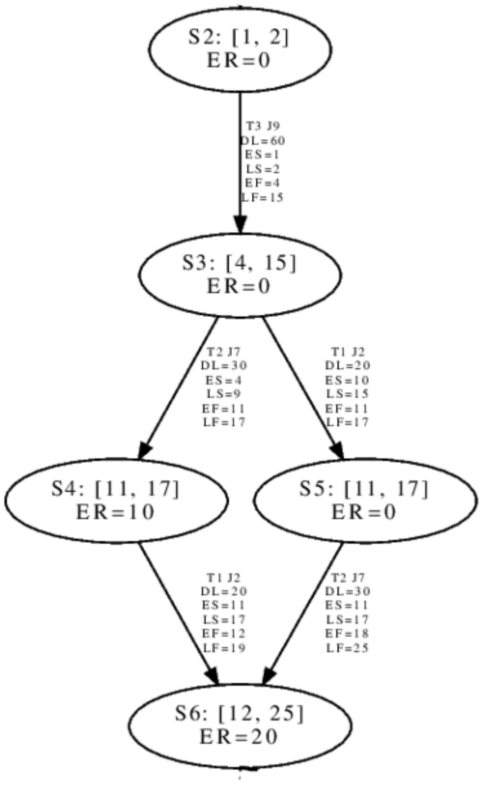


Figure 3.2: The part of dot graph of the example `fig1c.csv`, the full graph is in Appendix B.

3.4 Additional Challenges

The current schedule-abstraction graph algorithm developed by Nasri et al. [29] only keeps states on the front of the graph in memory. For instance, Example 1.2.2, generated from the schedule-abstraction graph shown in Figure 3.3.

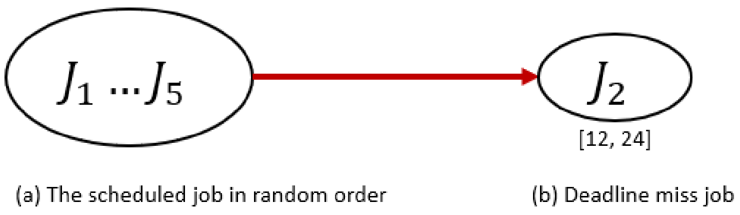


Figure 3.3: Part (a) is scheduled job set computed by the scheduling police. Part (b) is the final state which provides job J_2 's $[EFT, LFT]$.

Without the job dispatching sequence, we cannot set the ordering constraints between each action (job); consequently, the partial order plan approach is unlikely to work directly. Besides this, to compute the partial order plan, we need to save/compute extra features while building the schedule-abstraction graph

In the next section, we will discuss how we can overcome the stated challenges and hence how we can enhance and modify the schedule abstraction graph tool to produce counter-examples.

Chapter 4

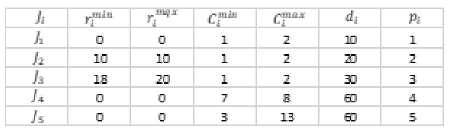
Build Counterexamples Using the Schedule-abstraction Graph

4.1 Partial Order Plan

Partial plan is a partially ordered plan, it is a tuple consisting of four components: a subset of the actions, a subset of the temporal constraints, a subset of rationale, and the subset of variable binding. The formal definition of the partial plans is demonstrated below:

Definition 4.1.1 (From[20])**.** A partial plan is a tuple $\pi = (A, \prec, B, L)$, where:

- $A = a_1, ..., a_k$ is a set of partially instantiated planning operators.
- $\prec$ is a set of ordering constraints on A of the form $(a_i \prec a_j)$.
- B is a set of binding constraints on the variables of actions in A of the form $x = y$, $x \neq y$, or $x \in D_x$, D_x being a subset of the domain of x.
- L is a set of causal links of the form $(a_i \xrightarrow{p} a_j)$, such that a_i and a_j are actions in A, the constraint $(a_i \prec a_j)$ is in $\prec$, proposition p is an effect of a_i and a precondition of a_j , and the binding constraints for variables of a_i and a_j appearing in p are in B.



J_i	r_i^{min}	r_i^{max}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	0	0	1	2	10	1
J_2	10	10	1	2	20	2
J_3	18	20	1	2	30	3
J_4	0	0	7	8	60	4
J_5	0	0	3	13	60	5

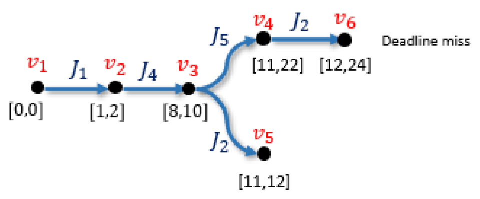


Figure 4.1: A simple example illustrating a unschedulable job set

Let us illustrate the partial plans corresponding the Example 1.2.2. The job dispatching sequence J_1, J_4, J_5, J_2 is claimed as unschedulable. The information generated by the tool is depicted in Figure 4.1. The set of partially instantiated planing operators is $A = a_1$, a_4, a_5, a_2, where a_i is the action of dispatching job J_i. The start step in the partial order plan is job J_1, and the finishing stage is job J_2, which misses its deadline $(A_1^{max}(v_6) = 24)$ as shown in Figure 4.2.

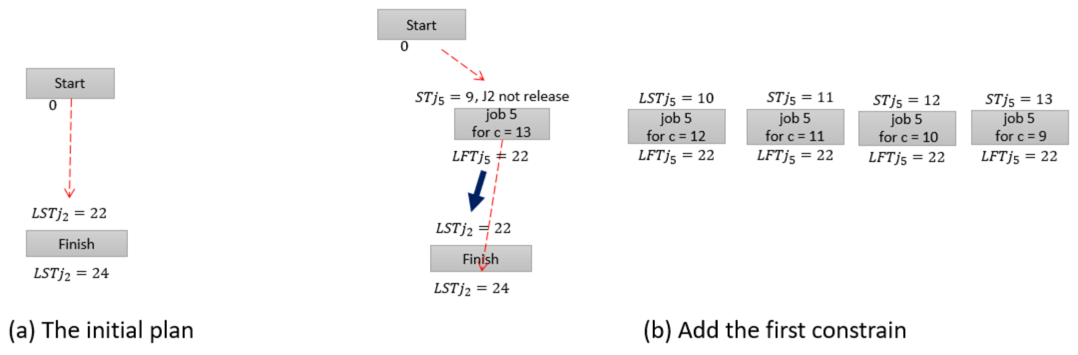


Figure 4.2: Part (a) is the *dummy* steps in the initial plan, the start step has the initial conditions and the finish step has the goal condition as its preconditions. Part (b) of the figure is we added one action, the reason for adding this action is to achieve the unsatisfied preconditions.

The computation interval of the job J_2 is $C_{J_2} \in [1, 2]$. We decide to use $C_{J_2} = 2$ here. The ordering constrain (red line) is shown in Figure 4.2, means job J_5 has to complete before job J_2 starts. The blue causal link in the same figure illustrates that we are going to use $LFT_{J_5} = 22$ to satisfy the precondition $LST_{J_2} = 22$ in the finish step. Note the computation interval of the job J_5 is $C_{J_5} \in [3, 13]$, however, only $C_{J_5} = 13$ which implies $LST_{J_5} = 9$ can be selected as the computation time of job J_5 in this case because of the schedule policy from the schedule-abstraction graph [29]. After adding all the actions, the counterexample is built as depicted in Figure 4.3. This plan is *complete* because every precondition of every step is satisfied by the effect of the previous steps.

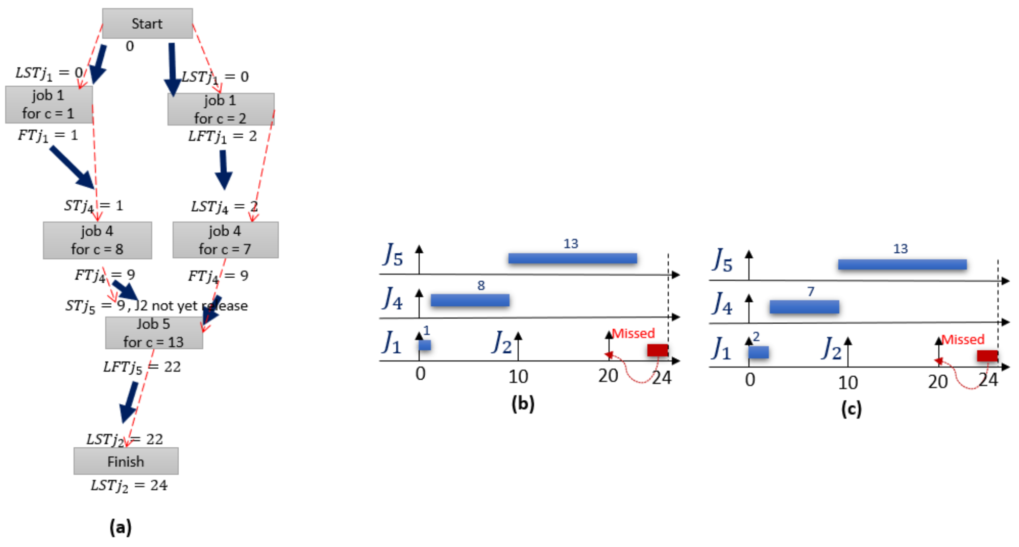
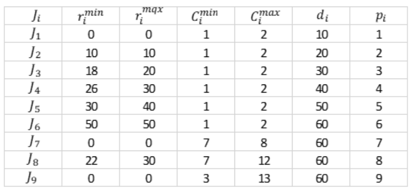


Figure 4.3: The complete partial plan is shown is part (a). Part (b) and part (c) in the figure are two counterexamples derive from the partial plan.

Example 4.1.1. The example shown in Figure 4.4 is more complicated to apply the partial order plan. The main challenges are:

1. The label of the vertex v_x is updated after the merge phase.
2. The core encounters an idle time.

Consider we know the job dispatching sequence $J_1, J_7, J_2, J_9, J_3, J_8, J_4$ is unschedulable shown in Figure 4.4.



J_i	r_i^{min}	r_i^{mqx}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	0	0	1	2	10	1
J_2	10	10	1	2	20	2
J_3	18	20	1	2	30	3
J_4	26	30	1	2	40	4
J_5	30	40	1	2	50	5
J_6	50	50	1	2	60	6
J_7	0	0	7	8	60	7
J_8	22	30	7	12	60	8
J_9	0	0	3	13	60	9

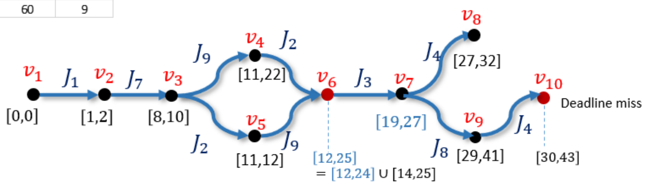


Figure 4.4: Scheduling-abstraction graph of Example 4.1.1. In this diagram, job J_4 may miss its deadline if the lower priority job J_8 executes first.

The *goal* of Example 4.1.1 is job J_4 missing its deadline, so we consider the finish time of J_4 is in the range $t \in [41, 43]$. The *merge* action (in `step 4`) is added in the partial plan shown in Figure 4.5. This action ensures the *effect* of job J_2 can be considered as the precondition of dispatching the next action job J_3 at $[12, 25]$. Further, when the core has idle time, the additional preconditions need to be obtained (displayed in blue colour) from the schedule-abstraction graph (Figure 4.4) to build the causal link from `step 3` to `step 2`. Different execution scenarios of the job J_4 missed the deadline can be found in Figure 4.6.

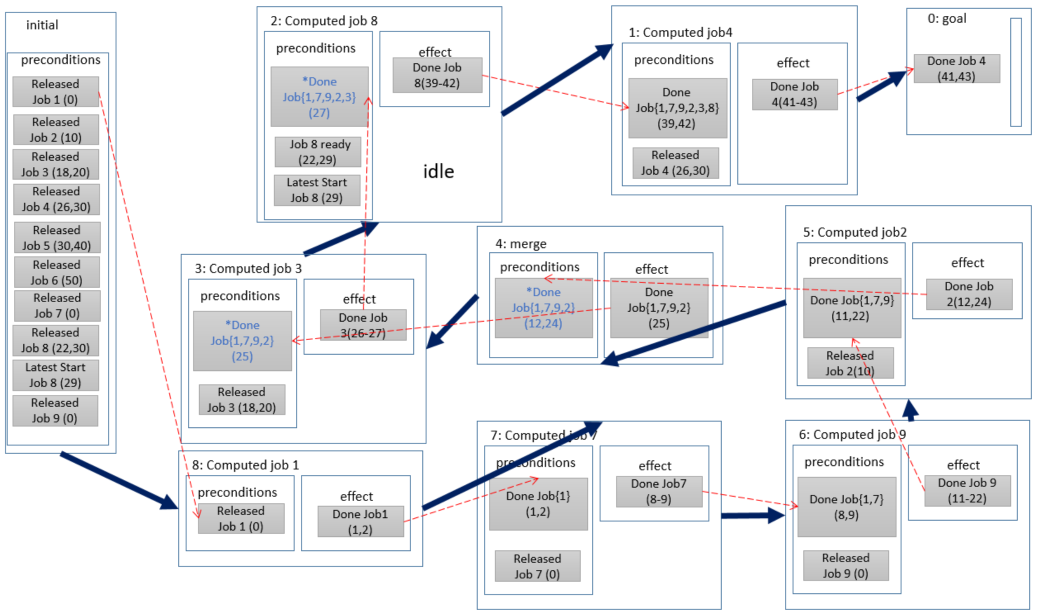


Figure 4.5: A partial plan of the Example 4.1.1. In this graph we skip some causal links from the preconditions in the initial to the remainder states.

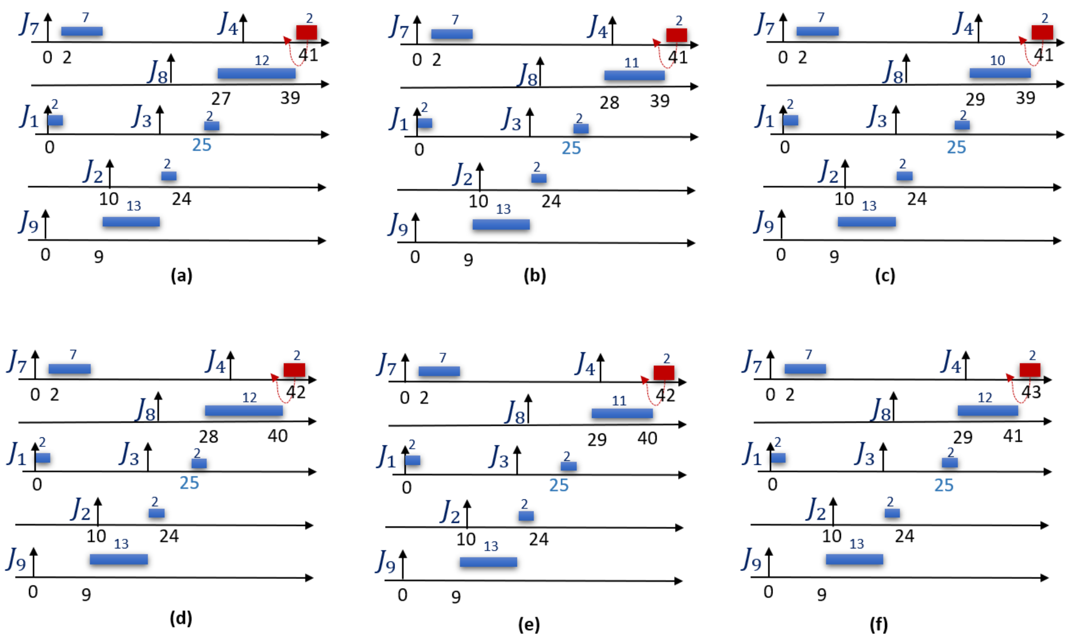

J_7 7 J_4 2 0 2 41 J_8 12 27 39 J_1 2 J_3 2 0 25 J_2 2 10 13 24 J_9 0 9 (a)
J_7 7 J_4 2 0 2 41 J_8 11 28 39 J_1 2 J_3 2 0 25 J_2 2 10 13 24 J_9 0 9 (b)
J_7 7 J_4 2 0 2 41 J_8 10 29 39 J_1 2 J_3 2 0 25 J_2 2 10 13 24 J_9 0 9 (c)
J_7 7 J_4 2 0 2 42 J_8 12 28 40 J_1 2 J_3 2 0 25 J_2 2 10 13 24 J_9 0 9 (d)
J_7 7 J_4 2 0 2 42 J_8 11 29 40 J_1 2 J_3 2 0 25 J_2 2 10 13 24 J_9 0 9 (e)
J_7 7 J_4 2 0 2 43 J_8 12 29 41 J_1 2 J_3 2 0 25 J_2 2 10 13 24 J_9 0 9 (f)

Figure 4.6: Part (a) - (f) are different execution scenarios built from the partial order graph of Figure 4.5.

4.2 Partial Order Plan Algorithm

This chapter explains how to generate one counterexample using a three-steps algorithm. Figure 4.7 provides a high-level overview of three steps. First the candidate counterexample job set is created using a modified version of the SAG algorithm by Nasri et al. [29]. The modifications to the original SAG algorithm will be presented in details in Section 4.3. On the other hand, if the system was deemed schedulable, the program stops as there is no possible execution scenario where a job misses its deadline that can be generated. If a candidate counterexample job set was generated, we use it to perform the second step of the algorithm, which will be explained details in Section 4.4. In the second step, a compressed version of the schedule abstraction graph is created. In the third step of the algorithm, we then used the compressed schedule abstraction graph for backward searching (see Section 4.5) to generate the counterexample.

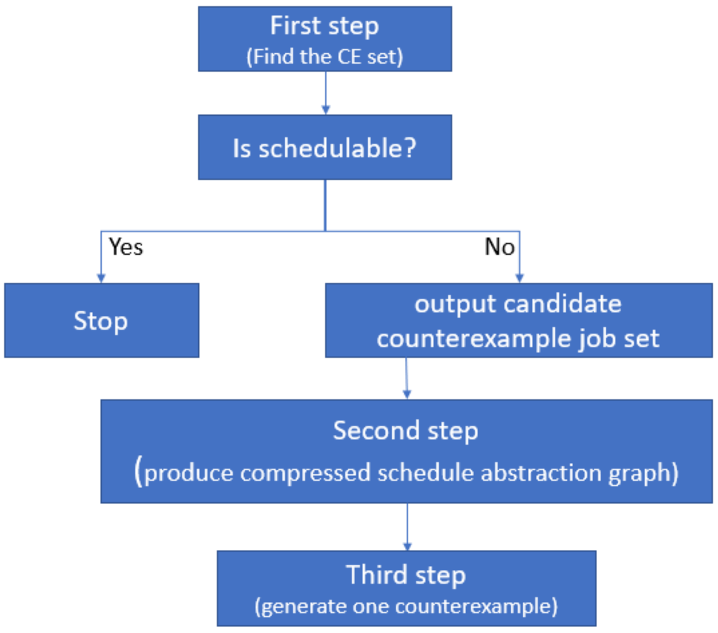

First step
(Find the CE set)
Is schedulable?
Yes
No
Stop
output candidate counterexample job set
Second step
(produce compressed schedule abstraction graph)
Third step
(generate one counterexample)

Figure 4.7: High-level overview of three runs

4.3 Creating the candidate counterexample job set

The initial solution for building a counterexample is by looking for which jobs are actually involved in a potential deadline miss. By limiting our attention to that reduced job set, we can avoid considering the whole job set $\mathcal{J}$ when building the counterexample and thus reduce the runtime and memory consumption of the algorithm. This subset of jobs, which is referred to as the candidate counterexample job set, must follow a certain number of properties:

1. the candidate counterexample job set must experience a potential deadline miss under the given scheduling policy;

2. all execution scenarios possible with the candidate counterexample job set must also be possible with the original job set with the given scheduling policy.

In other words, the schedule abstraction graph for the candidate reduction job set must be a subgraph of the schedule abstraction graph of the original job set.

To create this candidate counterexample job set, we use the property proven below in Lemma 4.3.1 and Corollary 4.3.2 that any job executed before an idle time will not impact the occurrence of a deadline miss happening after that idle time. To prove this property, we first define an idle state as in Definition 1, using the work-conserving property of our system model, that is to say that the processor only idle when there is no *ready jobs.*

Definition 4.3.1. A system state v_p is an idle state for an eligible successor job J_j if $A^{max}(v_p) < EST_j$.

An example illustrating the Definition 4.3.1 is shown in Figure 4.8. Since by definition $A^{max}(v_p)$ is the last time at which the processor will become available to execute the eligible successor job, and because EST_j is the earliest time the eligible successor job may start executing on the platform, the the platform will certainly be idle from $A^{max}(v_p)$ to EST_j.

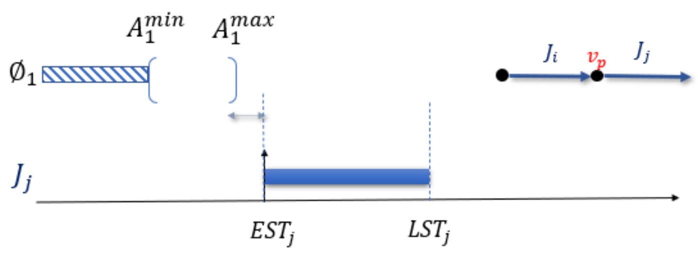


Figure 4.8: The idle state illustration

Lemma 4.3.1. The set of jobs J^{CE} scheduled after the idle state v_p is not impacted by the job set dispatched before state v_p.

Proof. Consider a job J_j executed right after the idle state v_p. The start time of J_j is within the interval $[EST_j(v_p), LST_j(v_p)]$ and because job J_j is eligible to be executed after v_p, we have $EST_j \leq LST_j$ (see Section 3.1) . Nasri et al. proved [31] that the earliest start time of job J_j is bounded by

$$EST_j = max\left\{r_j^{min}, A_1^{max}\right\} \tag{4.1}$$

Since v_p is an idle state, we have by Definition 4.3.1 that $A_1^{max} < EST_j$. Combining the latter equation with (4.1), we conclude that the earliest start time of job J_j depends on the job J_j its *possible ready* time r_j^{min}.

Nasri et al. [31] define the latest start time of the job J_j as follows

$$LST_j = min\left\{t_{wc}, t_{high} - 1\right\} \tag{4.2}$$

where, t_{high} is the earliest point in time from which on job J_j certainly is not the highest-priority ready job anymore and t_{wc} is an upper bound on the time at which there is certainly a ready job that will be executed by the work-conserving scheduler policy. Those are computed with Equation (4.3) and (4.4).

$$t_{\text{high}} \triangleq \min_{\infty}\left\{r_x^{\max} \mid J_x \in \mathcal{J}\backslash\mathcal{J}^P \wedge p_x < p_j\right\} \tag{4.3}$$

$$t_{wc} \triangleq \max\left\{A_1^{\max}\left(v_p\right), \min_{\infty}\left\{r_x^{\max} \mid J_x \in \mathcal{J}\backslash\mathcal{J}^P\right\}\right\} \tag{4.4}$$

□

Both (4.3) and (4.4) depend only on the set of of jobs $J \backslash J^P$, that is the set of jobs that have not been scheduled yet until reaching system state v_p and by extension the start time of the jobs executed after those. Therefore, we just proved that the set of jobs executed before an idle state v_p does not affect the start time of the job dispatched at state v_p. Therefore, by taking the state v_p as the *new* initial state, we can remove all jobs dispatched before the system state v_p.

Corollary 4.3.2. Any job executing before an idle state v_p will not contribute to the a deadline miss occurring after the idle state v_p.

Proof. It is a direct consequence from Lemma 4.3.1 □

Definition 4.3.2. The candidate counterexample job set is defined as $\mathcal{X}$

$$\mathcal{X} : \mathcal{J} \backslash \left(\mathcal{J}^{\mathcal{P}} \backslash \mathcal{J}^{\mathrm{CE}}\right) \tag{4.5}$$

Where $\mathcal{J}$ is the total job set, $\mathcal{J}^{\mathcal{P}}$ is the path that leads to the deadline miss and $\mathcal{J}^{\mathrm{CE}}$ is the set of jobs scheduled after the latest idle state v_p. As we can see from Example 4.3.1, $\left(\mathcal{J}^{\mathcal{P}} \backslash \mathcal{J}^{\mathrm{CE}}\right)$ is all jobs that have been executed before the last idle state. Hence, the candidate counterexample job set is the entire job set excluding all jobs executed before the last idle state.

To compute the set of jobs J^{CE} and thus the candidate counterexample job set $\mathcal{X}$, we modify the SAG algorithm. Each state v_i generated by the SAG during its expansion phase carries a set $CE(v_i)$ as a label that contains the set of jobs that were executed since the last idle state encountered before v_i i.e., $CE(v_i)$ is the set for system state v_i.

Definition 4.3.3. If J_i is a direct successor job of an idle state v_p, by dispatching J_i, a vertex v_n is created and added to the SAG by connecting it to v_p, the CE job set in the system state v_n is :

$$CE(v_n) = \{J_i\} \tag{4.6}$$

In the above scenario, J_i is a direct successor job of an idle state v_n, For any state v_n resulting from dispatching a job J_i after a non-idle system state v_m, we have:

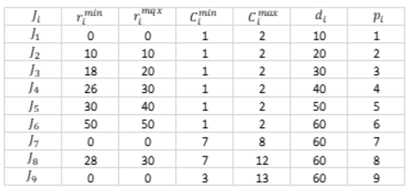
$$CE(v_n) = CE(v_m) \cup \{J_i\} \tag{4.7}$$

If two states v_m and v_n are merged into a new single state v_q during the generation of the SAG (see the explanation of the merge phase in Section 3.1 as a means to avoid state space explosion when building the SAG), we build the the set $CE(v_q)$ as follows

Definition 4.3.4. When merging system states v_m into a new system state v_q, the job set $CE(v_q)$ of v_q is given by

$$CE(v_q) \leftarrow CE(v_m) \cup CE(v_n) \tag{4.8}$$

Example 4.3.1. Consider the scheduling scenario of a job set $j = \{J_1, J_2, ..., J_9\}$, the schedule-graph of this job set is shown in the Figure 4.9. For that job set, job J_4 misses its deadline when reaching system state $v_1 0$. According to Definition 4.3.1, the state v_7 is the last idle state occurring before J_4 misses its deadline at $v_1 0$. This is because the earliest start time of job J_8 is equal to $r_8^{\min} = 28$ and the core certainly ready time at state v_7 is 27. Therefore, it holds that $r_8^{\min} > A_1^{max}$.



J_i	r_i^{min}	r_i^{max}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	0	0	1	2	10	1
J_2	10	10	1	2	20	2
J_3	18	20	1	2	30	3
J_4	26	30	1	2	40	4
J_5	30	40	1	2	50	5
J_6	50	50	1	2	60	6
J_7	0	0	7	8	60	7
J_8	28	30	7	12	60	8
J_9	0	0	3	13	60	9

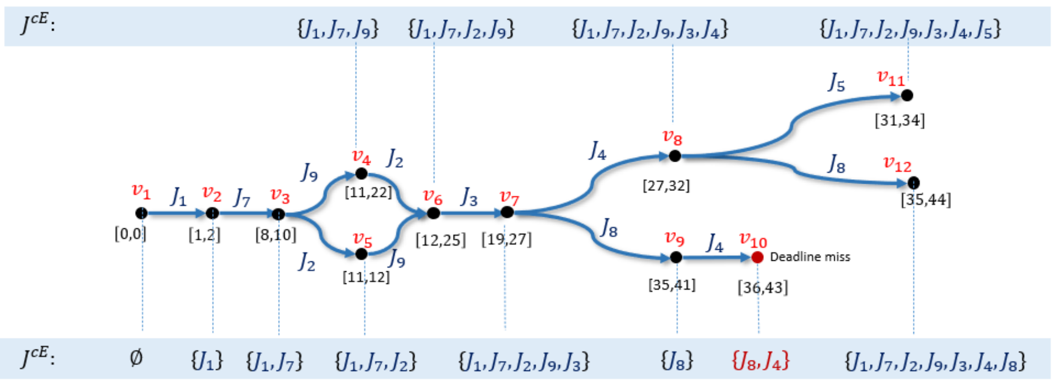
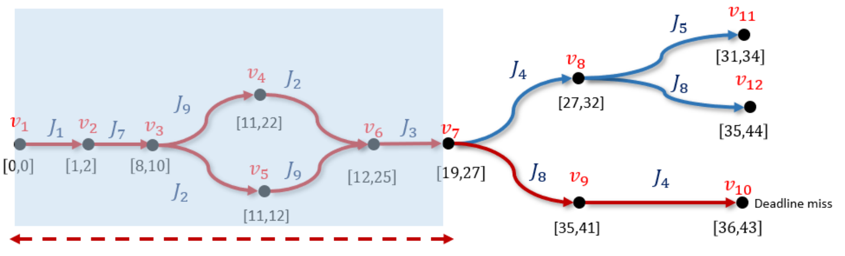
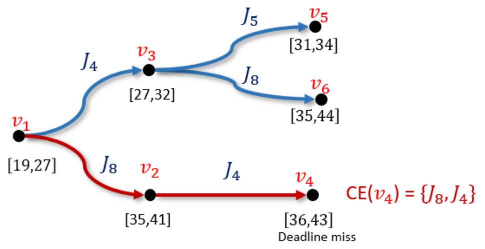

J^{cE}:
$\{J_1,J_7,J_9\}$
$\{J_1,J_7,J_2,J_9\}$
$\{J_1,J_7,J_2,J_9,J_3,J_4\}$
$\{J_1,J_7,J_2,J_9,J_3,J_4,J_5\}$
v_1 J_1 v_2 J_7 v_3 J_9 v_4 J_2 v_5 J_2 J_9 v_6 J_3 v_7 J_4 v_8 J_5 v_{11} J_8 v_{12} J_8 v_9 J_4 v_{10}
[0,0] [1,2] [8,10] [11,22] [11,12] [12,25] [19,27] [27,32] [31,34] [35,44] [35,41] [36,43]
Deadline miss
J^{cE}:
$\emptyset$
$\{J_1\}$
$\{J_1,J_7\}$
$\{J_1,J_7,J_2\}$
$\{J_1,J_7,J_2,J_9,J_3\}$
$\{J_8\}$
$\{J_8,J_4\}$
$\{J_1,J_7,J_2,J_9,J_3,J_4,J_8\}$

Figure 4.9: The modified version of the schedule graph and the label of the CE job set (J^{CE}) for each vertex are calculated using Definition 4.3.3 and 4.3.4.

According to Corollary 4.3.2, in this example, none of the jobs J_1, J_2, J_3, J_7 or J_9 participate to reaching the deadline miss state. Therefore, the job set, J^{CE} is at the deadline miss state v_10 contains only the jobs $\{J_4, J_8\}$. Finally, according Definition 4.3.2, the candidate counterexample job set is given by $\mathcal{X} = \{J_1, J_2...J_9\} - \{J_1, J_7, J_9, J_2, J_3, J_8, J_4\} + \{J_8, J_4\} = \{J_5, J_6, J_8, J_4\}$ (See Figure 4.10).


v_1 J_1 v_2 J_7 v_3 J_9 v_4 J_2 J_2 v_5 J_9 v_6 J_3 v_7 J_4 v_8 J_5 v_{11} J_8 v_{12} J_8 v_9 J_4 v_{10}
[0,0] [1,2] [8,10] [11,22] [11,12] [12,25] [19,27] [27,32] [31,34] [35,44] [35,41] [36,43]
Deadline miss
Non necessary jobs

(a) In the schedule-abstraction graph, the path that leads to the deadline miss state v_{10} is coloured with red


v_1 J_4 v_3 J_5 v_5 J_8 v_6 J_8 v_2 J_4 v_4
[19,27] [27,32] [31,34] [35,44] [35,41] [36,43]
Deadline miss
CE(v_4) = $\{J_8, J_4\}$

(b) After the reduction, the path that leads to the deadline miss state v_4

Figure 4.10: job set $\{J_8, J_4\}$ is the CE job set ($J^{\mathtt{CE}}$). Clearly, the SAG of the CE job set is the one we can found in the fig (b) which is indeed a subgraph of the original SAG shown in fig (a).

Algorithm 1 illustrates how to generate the CE job set J^{CE} for system state v_k. It follows the SAG construction algorithm of Nasri et al.[29], with modifications made to the creation and update of CE job sets at lines 7-13 and line 23. In summary, the CE job set J^{CE} is updated during each system state of SAG expansion and merge phase. If a job fails to meet its deadline, the program stops and return the candidate counterexample job set $\mathcal{X}$ for that deadline miss (lines 17 to 19).

The graph is initialized in line 1 with a root vertex v_1 labeled with the interval $[0, 0]$. The repeating expansion phase corresponds to lines 2-20. In the expansion phase, firstly, we determine whether system state v_p is an idle state (line 7), if so, a vertex v_k is created with the CE job set label equal to the direct successor job J_i (line 9). When system state v_p is not idle, it means that previous jobs dispatched before v_p may have contributed to the deadline miss. It is therefore necessary to remember every job on the path, in addition to the direct successor job J_i (line 11).

The path P is extended at line 16 by connecting the new vertex v_k to the previous vertex v_p by adding this new system state v_k to the path P. The algorithm then performs the merge phase at lines 22-26, this is the same process as the merge phase mentioned in Section 3.1, except that we also update the CE job set via the Definition 4.3.4.

When the system is deemed unschedulable, the program stops and the candidate counter-example job set is generated in line 18.

Algorithm 1 Algorithm for CE job set creation - idle state detection

```
    Input: job set $\mathcal{J}$
    Output: CE job set $CE(k)$, the candidate counterexample job set $\mathcal{X}$
 1: Initialize $G$ by adding a root vertex $v_1$ with interval $[0, 0]$
 2: while ($\exists$ path $P$ from $v_1$ to a leaf $v_p$ s.th. $|P| < |J|$) do
 3:     $P \leftarrow$ the shortest path from $v_1$ that ends in a leaf $v_p$
 4:     if there is no eligible job then
 5:         return unschedulable
 6:     end if
 7:     for each eligible successor job $J_j$ do
 8:         if $r_j^{min} > A_{max}(v_p)$ then
 9:             $CE(k) = \{J_j\}$
10:         else
11:             $CE(k) = CE(p) \cup \{J_j\}$
12:         end if
13:     end for
14:     Add a new vertex $v_k$ and CE(k) to V with label $[EFT_k, LFT_k]$
15:     Add an edge from $v_i$ to $v_k$ with label $J_j$
16:     Let path $P' = P + \langle v_k \rangle$
17:     if $J_j$ misses its deadline then
18:         $\mathcal{X} \leftarrow \mathcal{J} \backslash \left( J^{P'} \backslash CE(k) \right)$
19:         exit
20:     end if
21:     while ($\exists Q$ matches $P'$ ) do
22:         Update $[EFT_k, LFT_k] \leftarrow [EFT_q, LFT_q] \cup [EFT_k, LFT_k]$
23:         Update $CE(K) \leftarrow CE(Q) \cup CE(K)$
24:         Redirect all incoming edges of $v_q$ to $v_k$
25:         Remove $v_q$ from V
26:     end while
27: end while
```

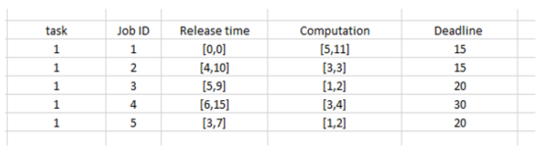
4.4 Build the compressed SAG for backward searching

Recall that the final objective of our algorithm is to generate a concrete execution scenario in which a job J_i misses its deadline. That is, we must find a set of which shows execution times $C^{\text{miss}} = <C_1, C_2, ..., C_m>$ and release times $r^{miss} = <r_1, r_2, ..., r_m>$ such that, for each job $J_i \in J$, $C_i \in [C_i^{min}, C_i^{\max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$, and J_i misses its deadline, i.e., finishes its execution at time t such that $t - r_i^{min} > D_i$. To reduce the search space that must be explored to find this execution scenario, we construct a compressed version of the schedule abstraction graph using the two following properties

Property 1. Any path such that J_i's latest finish time is smaller than its deadline must not be in the compressed SAG.

Property 2. The compressed SAG, should only contain jobs from J^{CE} as returned by Algorithm 1.

Example 4.4.1. Figure 4.11 shows the complete SAG of an unschedulable job set, where job J_2 may miss its deadline at time 15. This finish time t includes the possibility that $t > D_2 \rightarrow t > 15$. In this example, J_2 appears in several paths in the SAG. However, most of those path do not end up with J_2 missing its deadline. For instance, path $J_1 \rightarrow J_4 \rightarrow J_2$ is not the deadline missed path since the finish time of job J_2 is in the range $[14, 15]$. Similarly, the path $J_1 \rightarrow J_5 \rightarrow J_4 \rightarrow J_2$, $J_1 \rightarrow J_5 \rightarrow J_2 \rightarrow J_3$ ... $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_5$ are not the deadline missed path because J_2 completes before or at time 15 in all those paths.



task	Job ID	Release time	Computation	Deadline
1	1	[0,0]	[5,11]	15
1	2	[4,10]	[3,3]	15
1	3	[5,9]	[1,2]	20
1	4	[6,15]	[3,4]	30
1	5	[3,7]	[1,2]	20

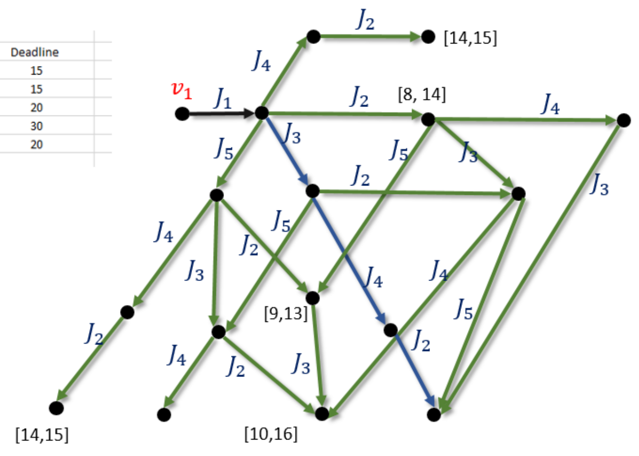


Figure 4.11: The original SAG

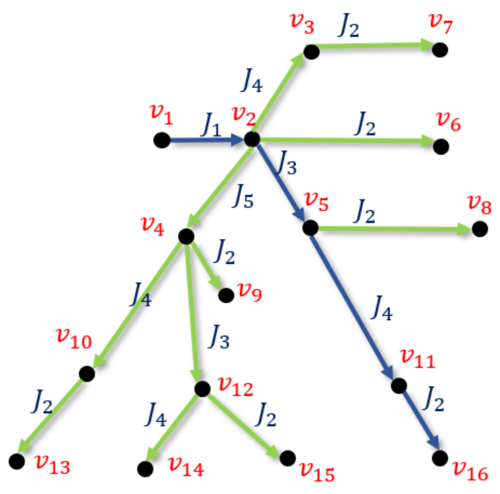


Figure 4.12: The smaller graph of the Example 4.4.1 job set

Accordingly, the second step algorithm will not save the edges where the finish time t of a deadline missed job J_m is less than the deadline of J_m and will produce the smaller graph shown in Figure 4.12 instead.

We proved in the previous section that an idle state v_p can take as the initial state, we can remove all jobs dispatched before the system state v_p. Additionally, the candidate counterexample job set (J^{CE}) must experience a potential deadline miss under the given scheduling policy. Therefore the compressed SAG returned by Algorithm 2 should only contain the jobs from CE job set J^{CE} (See Property 2). The Algorithm is explained in detail as follows:

In line 1 of Algorithm 2, we initialize the compressed SAG. During the while-loop, the algorithm first selects a direct successor job J_j from the CE job set and then it checks whether J_j is the deadline missed job (line 5). If so, we determine if J_j's latest finish time exceeds its deadline D_j (line 6). If that is the case, then we add it (v_k) directly after state v_p, through lines 7-9, after that, we will exit the program as we already have the deadline miss execution scenario (line 9).

Based on the Property 1, the deadline missed job J_j is not added to the edge if the latest finish time of J_j is less than its deadline, in order to prevent expanding non-deadline missed paths (line 11). In the case of any other non-deadline missed eligible jobs in the CE job set, we must include these jobs in the graph (lines 14-15) according to Property 2.

As soon as we begin the merge phase, it will operate in the same manner as the original SAG, as we explained in Section 3.1.

Algorithm 2 Algorithm for graph creation

```
    Input: Candidate counterexample job set $\mathcal{X}$
    Output: A condensed SAG includes a deadline missed path
 1: Initialize $G$ by adding a root vertex $v_1$ with interval $[0, 0]$
 2: while ($\exists$ path $P$ from $v_1$ to a leaf $v_p$, and leaf $v_p$ is eligible s.th. $|P| < |J^{CE}|$) do
 3:     $P \leftarrow$ the shortest path from $v_1$ that ends in a leaf $v_p$
 4:     for each eligible successor job $J_j$ is in the $J^{CE}$ do
 5:         if $J_j$ is the deadline missed job then
 6:             if $J_j$ misses its deadline then
 7:                 Add an edge from $v_i$ to $v_k$ with label $J_j$
 8:                 Let path $P' = P + \langle v_k \rangle$
 9:                 Exit
10:             else
11:                 set leaf vertex $v_k$ is not eligible
12:             end if
13:         else
14:             Add an edge from $v_i$ to $v_k$ with label $J_j$
15:             Let path $P' = P + \langle v_k \rangle$
16:         end if
17:         Execute the merge phase (see Section 3.1)
18:     end for
19: end while
```

4.5 Partial order plan for finding a counterexample (third step)

This section we introduce the third step algorithm for finding a counterexample execution scenario. According to the original SAG algorithm, it is a *state-space* algorithm, meaning it searches through the space of possible states attempting to identify all solutions to the problem. Furthermore, the SAG is based on a forward search from the initial state to the goal state. Unlike the first and second step algorithms, the third steps backward towards the initial system state. This third step algorithm is a *plan-space* search through the space of partial plans, which are sets of actions that are not necessarily ordered.

The complete Partial order plan construction procedure is given in Algorithm 3. First, we initialize the graph G with the condensed SAG return from Algorithm 2 (line 1), empty the branch list (line 2) and mark each graph node as an unexplored node (lines 3 - 4). By inserting the starting node of a graph in the stack we are able to begin exploring the graph in a depth-wise manner (lines 11-18).

During exploration, we first pop the node x from the stack and add it to the visited list (line 10). Whenever there is more than one adjacent node to a node x, this node x can become a branch point, indicating we made multiple job dispatching decisions at this system state. The branch point is then added to the list (lines 7-9). The node v can only be added to the linear plan if the eligibility condition is met. A detailed discussion of the eligibility conditions will follow.

Having checked all the nodes adjacent to node x, and still unable to find one which meets the eligibility condition, we must return to the most recent branch point y, and remove all nodes from the plan up to point y and then repeat the DFS (lines 19-24).

Algorithm 3 Partial order plan for finding a counterexample

```
Input: A condensed SAG
Output: Linear plan to achieve goal
1: Initialize G ← A condensed SAG
2: Initialize branchlist = ∅
3: for each node x ∈ $G$ do x.visited = false
4: end for
5: Initialize x ← the deadline missed job as start node
6: if x is not initial system state ($v_1$) then
7:     if size of G.Adj[x] > 1 then
8:         add x to the branchlist
9:     end if
10:    x.visited = true
11:    for each v ∈ G.Adj[x] do
12:        if v.visited == false then
13:            if $J_j$ is eligible job then
14:                add v to the linear plan
15:                DFS(G,v)
16:            end if
17:        end if
18:    end for
19:    if no eligible job in G.Adj[x] then
20:        pop x from branchlist
21:        y ← branchlist.front()
22:        Remove all the edges till this branch point y
23:        Execute the DFS (see lines 11-18)
24:    end if
25: else
26:    Exit
27: end if
```

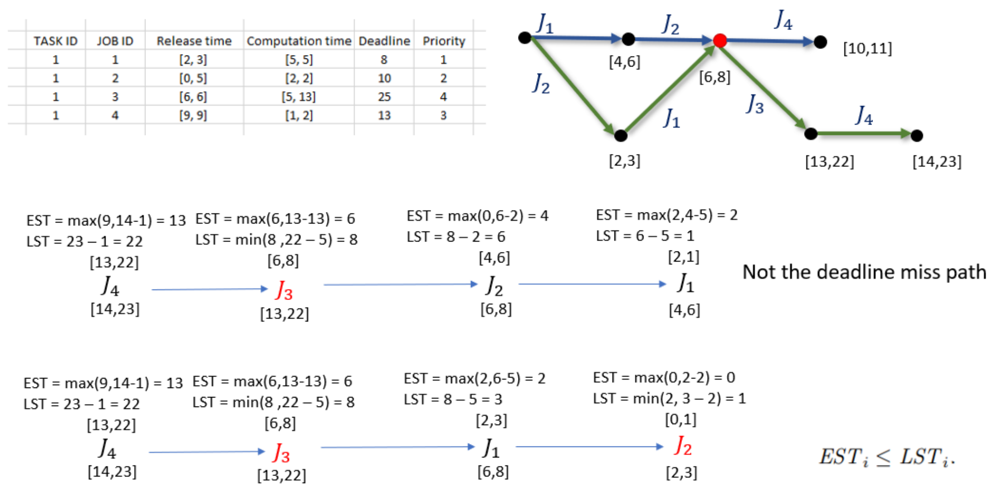
Partial Order Plan Eligibility Condition: A job J_i can be added in the linear plan during backward transition if its earliest start time EST_i^{DM} for deadline miss scenario is not later than its latest start time LST_i^{DM} for deadline miss scenario. i.e., if

$$EST_i^{DM} \leq LST_i^{DM} \tag{4.9}$$

Lemma 4.5.1. Job J_i is the corresponding action of v_p only if Inequality holds.

Proof. The upper bound of the time at which J_i is the corresponding action of v_p is defined by Equation (4.9) and (4.11). Therefore, if J_i cannot be dispatched by LST_i^{DM}, then it cannot be a direct successor of v_p. As EST_i^{DM} is the earliest time at which J_i can be dispatched after v_p. When the condition fails ($EST_i^{DM} > LST_i^{DM}$), the algorithm should move back to the most recent nondeterministic choice point and select another job not yet tried. □

Example 4.5.1. Figure 4.13 shows an example of unschedulable job set $\mathcal{J} = \{J_1, J_2, J_3, J_4\}$, where the finish time of deadline missed job J_4 is in the range $[14, 23]$. Thus our goal is to finish J_4 at $[14, 23]$, and the finish step has the goal conditions as preconditions, namely that J_4 begins at interval $[13, 22]$. To satisfy this precondition, we keep track of using causal links to add the job J_3 in the path. The effect of job J_2 and job J_1 can both satisfy the precondition of J_3. Hence we will have two possible deadline missed paths, the first one is path $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$, the second one is path $J_2 \rightarrow J_1 \rightarrow J_3 \rightarrow J_4$. Nevertheless, we noticed that the start time of J_1 in the path $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$ does not hold for the eligible condition $EST_i^{DM} < LST_i^{DM}$. Therefore, the deadline missed path is $J_2 \rightarrow J_1 \rightarrow J_3 \rightarrow J_4$.



TASK ID	JOB ID	Release time	Computation time	Deadline	Priority
1	1	[2, 3]	[5, 5]	8	1
1	2	[0, 5]	[2, 2]	10	2
1	3	[6, 6]	[5, 13]	25	4
1	4	[9, 9]	[1, 2]	13	3

Figure 4.13: The earliest start time and latest start time of job J_1 in the path $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$ is 2 and 1, respectively. Since the $EST_2^{DM} > LST_2^{DM}$, means J_2 is not the successor in this path. For the correct deadline miss path $J_2 \rightarrow J_1 \rightarrow J_3 \rightarrow J_4$, it illustrates that every precondition of every job is satisfied by the effect of some other job.

First, we make the initial plan for the third step. This initial plan contains the finish/goal step. The goal condition is the preconditions for this finish step. After that, we iterate until the plan is complete. We keep track of causal relationships by using causal links from the output of the second step. This will ensure that every precondition of every job is satisfied by the effect of some other job. The counterexample will continue to be developed until it reaches the initial state and all the preconditions are met.

In this section, we decided the eligibility condition $EST_i^{DM} \leq LST_i^{DM}$ to perform backward transition. If this eligibility condition is false, we must go back to the most recent branch to select a different option, from among those that have not yet been visited. If they've all been visited, we must go back to the next most recent branch, and keep exploring.

In the next section we then explain how to compute $EST_i^{DM} \leq LST_i^{DM}$ of J_i.

4.6 Earliest and Latest Start Times for deadline missed job

For each job $J_i \in J^{CE}$, we aim to prove a lower bound $EST_i^{DM}(v_p)$ and an upper bound $LST_i^{DM}(v_p)$ on the time at which J_i may start executing in v_p (Equations (4.9) and (4.11), respectively).

$$\mathrm{EST}_{J_i}^{DM} = \begin{cases} \max\left\{r_i^{min}, EFT_{J_i}^{DM} - C_i^{max}, EFT_{J_i} - C_i^{min}\right\} & \text{if } J_i \text{ misses its deadline} \\ \max\left\{r_i^{min}, EFT_{J_i}^{DM} - C_i^{max}\right\} & \text{otherwise} \end{cases} \tag{4.10}$$

$$\mathrm{LST}_{J_i}^{DM} = \min\left\{t_{high}^{DM} - 1, LFT_{J_i}^{DM} - C_i^{max}\right\} \tag{4.11}$$

$$\mathrm{t}_{high}^{DM} = \begin{cases} \infty & \text{if } J_i \text{ misses its deadline} \\ r_x^{max} | \mathrm{Depth}\left(J_x\right) = n + 1 \cap p_x < p_j & \text{otherwise} \end{cases} \tag{4.12}$$

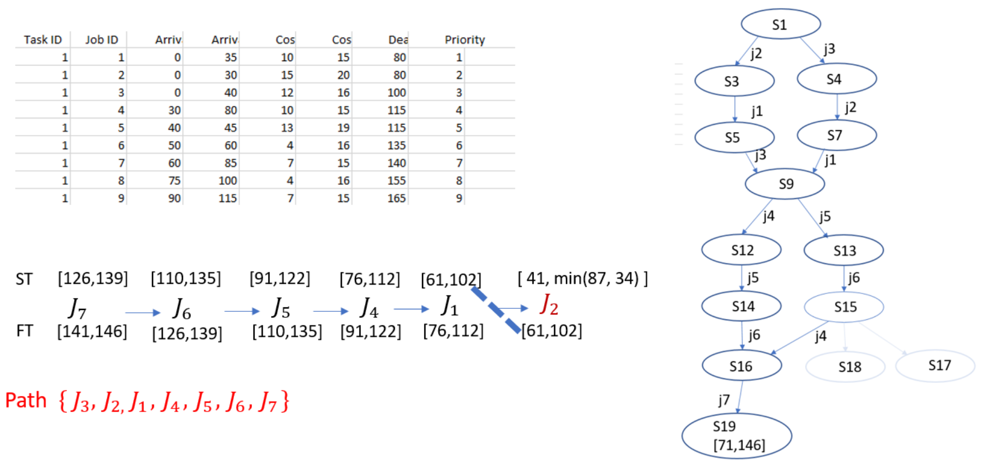
Lemma 4.6.1. Job $J_i \in CE$ can not start executing (as a successor of state v_p) before $EST_{J_i}^{DM}$.

Proof. Job $J_i \in J^{CE}$ cannot start executing prior to the earliest time at which it may release, i.e., r_i^{min}, nor can it start executing before the earliest time at which a core may become available, which is given by $EFT_{J_i}^{DM} - C_i^{max}$. For the deadline miss job, it cannot start before EST_i which defined by $EST_i = EFT_i - C_i^{min}$. Combining the facts, we observe that J_i is the certain job on the deadline miss path only if it starts later than $EST_{J_i}^{DM}$ shown in equation (4.9).

The latest start time of J_i on the deadline missed path is determined by the following: the scheduler follows a JLFP scheduling policy. t_{high}^{DM} define in equation (4.12) is the earliest point in time from which on J_i certainly is not the highest-priority ready job anymore. Since the deadline missed job is the final job that has been released on the path, so when job J_i misses its deadline, t_{high}^{DM} is infinity. The t_{high}^{DM} for a job that completed before its deadline depends on the latest release time of the higher priority job that dispatched after it. Taking into account the following facts: $LST_{J_i}^{DM} \leq t_{high}^{DM} - 1$ and $LST_{J_i}^{DM} \leq LST_{J_i}^{DM} - C_i^{max}$. We decided that the latest start time of J_i is shown in equation (4.11).

□

Now, consider the example in Figure 4.14 where J_2 's EST_2^{DM} is larger than LST_2^{DM} in path $J_3 \rightarrow J_2 \rightarrow J_1 \rightarrow J_4 \rightarrow J_5 \rightarrow J_6 \rightarrow J_7$. As a result, this is not the deadline missed path.



Task ID	Job ID	Arriv	Arriv	Cos	Cos	Dea	Priority
1	1	0	35	10	15	80	1
1	2	0	30	15	20	80	2
1	3	0	40	12	16	100	3
1	4	30	80	10	15	115	4
1	5	40	45	13	19	115	5
1	6	50	60	4	16	135	6
1	7	60	85	7	15	140	7
1	8	75	100	4	16	155	8
1	9	90	115	7	15	165	9

ST [126,139] [110,135] [91,122] [76,112] [61,102] [41, min(87, 34)]
$J_7 \to J_6 \to J_5 \to J_4 \to J_1 \to J_2$
FT [141,146] [126,139] [110,135] [91,122] [76,112] [61,102]
Path $\{J_3, J_2, J_1, J_4, J_5, J_6, J_7\}$
S1
j2
j3
S3
S4
j1
j2
S5
S7
j3
j1
S9
j4
j5
S12
S13
j5
j6
S14
S15
j6
j4
S16
S18
S17
j7
S19 [71,146]

Figure 4.14: An example showing a scenario with the high priority job J_1 dispatched later than the lower priority job J_2. Hence the $t^{DM}_{high}(J_2) = r^{max}_1 = 35$. The $LST^{DM}_2 = \min(34, 102 - 15) = 34$.

4.7 Earliest and Latest Finish Times for deadline missed job

Our goal is generating one potential example of execution scenario that leads to the deadline missed job J_i finishing at the interval which is later than job J_i's deadline. Consequently, the earliest and latest finish time of job J_i are defined as follows.

$$EFT^{DM}_i = \text{deadline}(J_i) + 1 \tag{4.13}$$

$$LFT^{DM}_i = LFT_i \tag{4.14}$$

The Figure 4.15 below illustrates the finish time range of the deadline missed job.

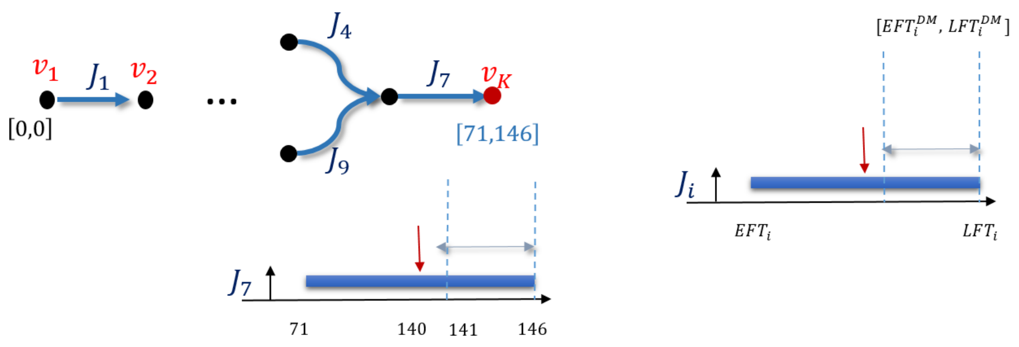


Figure 4.15: An example where job J_7 is the deadline missed job. The job J_7's deadline is 140 and it finishes its execution at time t such that $t > 140$. Thus the earliest finish time of job j_7 is $EFT^{DM}_{J_7} = 140 + 1 = 141$ and latest finish time of job J_7 is $=LFT^{DM}_{J_7} = LFT_{J_7} = 146$.

4.8 Completeness of the Proposed Solution

This section establishes the correctness of our analysis by showing that, for any unschedulable job set, there exists a linear plan created by Algorithm 3 that represents the schedule of specific jobs to reach a deadline miss execution scenario.

Theorem 4.8.1. For any deadline miss execution scenario such that a job $J_i \in \mathcal{X}$ finishes its execution at time t such that $t - r_i^{min} > D_i$, there must exist a path in the linear plan generated by Algorithm 3.

Proof. Our proof for this property is based on contradiction. We assume that the system misses a deadline, but the three steps algorithms do not return a counterexample. Based on this assumption, we can give two hypotheses as follow:

- H_0: There is no state where the eligibility condition (equation 4.9) holds.
- H_1: In one state, it is held for the eligibility condition. However, in our third step, our backward search algorithm never reaches this state.

For the hypothesis H_0, it implies that for all states (generated during the first or second step) we have $EST_i^{DM} > LST_i^{DM}$. For first step and second step algorithms, we search the space by building the SAG, when we dispatch a job from state v_p to state v_p', the eligibility condition $EST_i(v_p) \le LST_i(v_p')$ must hold. Since $EST_i^{DM}(v_p)$ is a subset of the $EST_i(v_p)$ and $LST_i(v_p) = LST_i^{DM}(v_p)$, then we can reject the hypothesise H_0.

We want to make a decision of hypothesis H_1 by using induction structure, assume that the path $P = \{v_p, v_p', ..., v_2, v_1\}$ respects the claim for jobs added in the plan before J_i in the execution scenario that led J_i to start at time t. Notice that each state in the plan represents a single step in the plan. An edge represents the transformation of one plan to another, such as adding a new step between existing steps. The base case is goal state v_p which is the interval $[EFT_i^{DM}, LFT_i^{DM}]$ of the deadline missed job J_i. Here we set a requirement, for each job $J_x \in J^P$, the interval $[EST_x^{DM}, LST_x^{DM}]$ safely lower- and upper-bounds (i.e., it contains) the start time interval of J_x.

In the induction step, we want to prove whether there exists a vertex v_p' that is directly connected to v_p with an edge labeled J_i. That requirement holds for state v_p' and the interval $[EST_x^{DM}, LST_x^{DM}]$ contains the start time of J_i. According to Lemma 4.9, algorithm 3 adds any job that can possibly be added before v_p, hence also for J_i. Moreover, equation (4.9) and equation (4.11) provide a lower and upper bound on J_i's start time. The requirement follows from the assumption that the interval $[EST_x^{DM}(v_p), LST_x^{DM}(v_p)]$ computed for every job J_x added before J_i in a state reached prior to $v_p$is correct.

Algorithm 3 initializes the deadline missed job j_i as the first job in the plan for any state v_p' that is a direct successor of the initial system state v_p. Thus, we reject hypothesis H_1.

Having rejected all hypotheses, we conclude that if there is a deadline miss job in the system, we will calculate the counterexample execution scenario using the three steps algorithm.

□

4.9 Correctness of the Proposed Solution

This section we want to demonstrate that the counterexample we generated from the three steps algorithm has valid bound.

Theorem 4.9.1. For any deadline miss execution scenario such that a job $J_i \in \mathcal{X}$ finishes its execution at time t, there exists a path $P = \{v_p, v_p', ..., v_2, v_1\}$ in the linear plan generated by

Algorithm 3 such that J_i is the label of the edge from the state v_p to the state v'_p and $t \in [EST_i^{DM}, LST_i^{DM}]$.

Proof. It has been proved that for path $P = \left\{v_1, v_2, ..., v'_p, v_p\right\}$ generated by first and second step algorithms, a vertex v_p has an edge labeled J_i and the start time of job J_i has the start time interval $[EST_i, LST_i]$. From the equation , clearly, the EFT_i^{DM} is always larger than the EST_i and LST_i^{DM} is equal to the LST_i generated from the SAG. Due to the fact that the interval for the path leading to the deadline missed state is a subset of the interval for the $[EST_i, LST_i]$. Therefore, the bound generated by the three steps algorithm is safe and correct.

□

Chapter 5

Implementation of Three Runs

This chapter discusses the implementation of the three-steps algorithm. Each step is implemented as a different program. The two first steps are two different modifications of the original schedule abstraction graph analysis available at `https://github.com/gnelissen/np-schedulability-analysis`. The third step is implemented as an entirely new C++ program. Technical details concerning the first step, second step, and third step are provided in Section 5.1, section 5.2 and section 5.3, respectively. These details include the software design choices made and the associated outputs.

5.1 First step implementation

The class diagram for the first step implementation is depicted in Figure 5.1. Nasri and Brandenburg [29] developed a breadth-first, state-space search algorithm, in which all possible scheduling decisions are implemented in the explore class. Each of these decisions adds a new system state and an edge to the graph. To implement the first step of our three-steps algorithm, we modified the original SAG program implemented by Brandenburg et al. [29]. Figure 5.1 shows the procedures we added to the original implementation in red. Our first step takes place during the expansion phase of the SAG. As soon as an idle state is detected, we modify the `CE_SET` in the state. Furthermore, this `CE_SET` is updated during the states merge phase. The program stops if a job's finish time t exceeds its deadline d, in which case the candidate counterexample job set (the input for the second step) and the identifier of the job that missed its deadline are saved as a csv file.

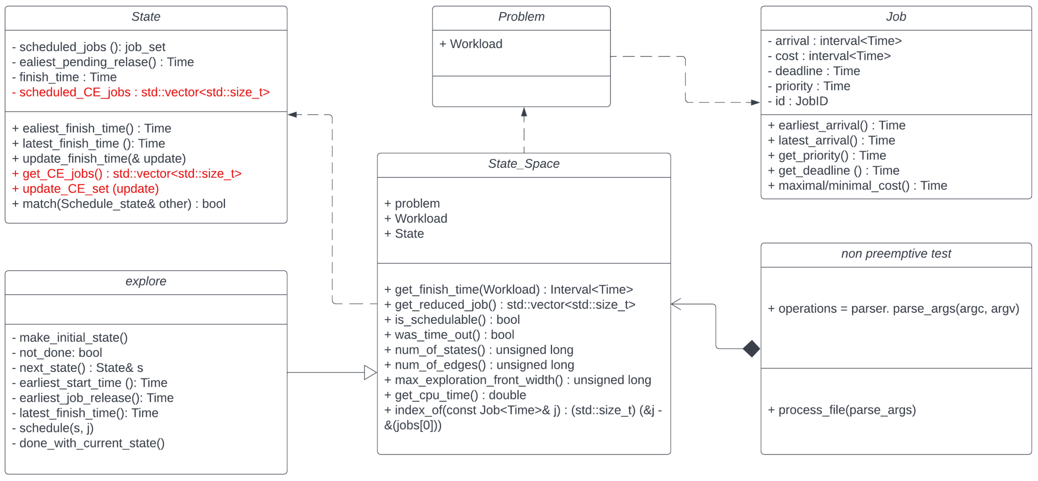

State
- scheduled_jobs (): job_set
- ealiest_pending_relase() : Time
- finish_time : Time
- scheduled_CE_jobs : std::vector<std::size_t>
+ ealiest_finish_time() : Time
+ latest_finish_time (): Time
+ update_finish_time(& update)
+ get_CE_jobs() : std::vector<std::size_t>
+ update_CE_set (update)
+ match(Schedule_state& other) : bool
Problem
+ Workload
Job
- arrival : interval<Time>
- cost : interval<Time>
- deadline : Time
- priority : Time
- id : JobID
+ earliest_arrival() : Time
+ latest_arrival() : Time
+ get_priority() : Time
+ get_deadline () : Time
+ maximal/minimal_cost() : Time
State_Space
+ problem
+ Workload
+ State
+ get_finish_time(Workload) : Interval<Time>
+ get_reduced_job() : std::vector<std::size_t>
+ is_schedulable() : bool
+ was_time_out() : bool
+ num_of_states() : unsigned long
+ num_of_edges() : unsigned long
+ max_exploration_front_width() : unsigned long
+ get_cpu_time() : double
+ index_of(const Job<Time>& j) : (std::size_t) (&j - &(jobs[0]))
explore
- make_initial_state()
- not_done: bool
- next_state() : State& s
- earliest_start_time (): Time
- earliest_job_release(): Time
- latest_finish_time(): Time
- schedule(s, j)
- done_with_current_state()
non preemptive test
+ operations = parser. parse_args(argc, argv)
+ process_file(parse_args)

Figure 5.1: UML class diagram for *first step* implementation

An example of output produced by the first step of our three-steps algorithm is shown in table 5.1

Task ID	Job ID	r min	r max	c min	c max	deadline	Priority	deadline miss
1	1	0	35	10	15	80	1	0
1	2	0	30	15	20	80	2	0
1	3	0	40	12	16	100	3	0
1	4	30	80	10	15	115	4	0
1	5	40	45	13	19	115	5	0
1	6	50	60	4	16	135	6	0
1	7	60	85	7	15	140	7	1
1	8	75	100	4	16	155	8	0
1	9	90	115	7	15	165	9	0

Table 5.1: An example output CSV data from the first step. This table format can be compared with the original input table 3.1. The first step output table has one extra column, namely the deadline miss column, which indicates job J_7 missed its deadline

5.2 Second step implementation

In the same way as the first step configuration, the second step implementation is built upon the original SAG analysis implemented by Brandenburg et al. [29]. All our modifications take place in the expansion of the SAG. A ccompressed SAG is generated when making scheduling decisions that lead to a deadline miss. A compressed SAG = (V, E) is created during the SAG exploration. Figure 5.2 shows the UML class diagram for the second step of the program.

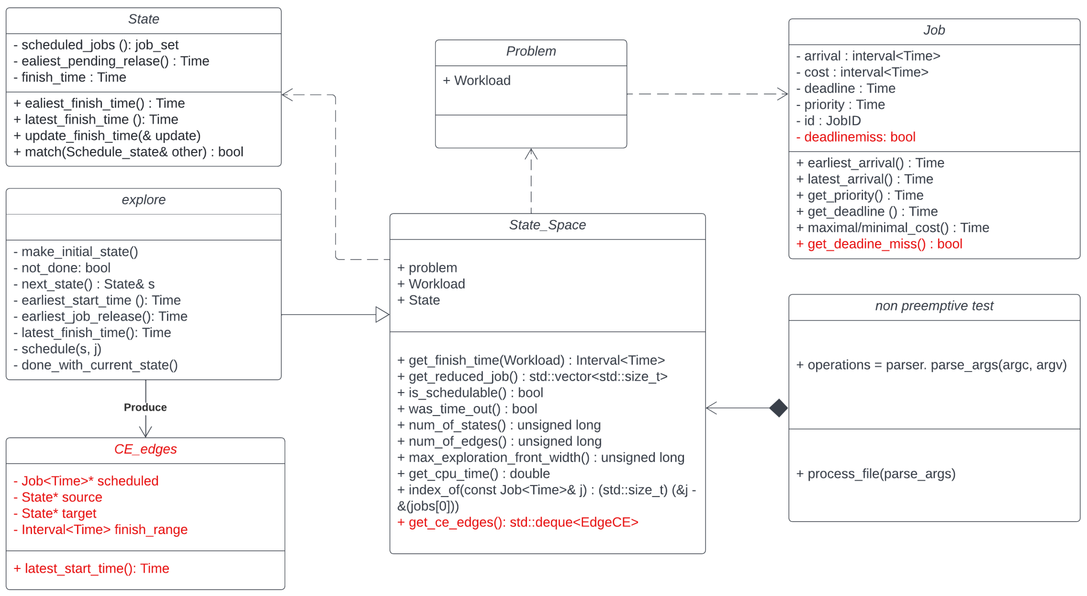

State
- scheduled_jobs (): job_set
- ealiest_pending_relase() : Time
- finish_time : Time
+ ealiest_finish_time() : Time
+ latest_finish_time (): Time
+ update_finish_time(& update)
+ match(Schedule_state& other) : bool
Problem
+ Workload
Job
- arrival : interval<Time>
- cost : interval<Time>
- deadline : Time
- priority : Time
- id : JobID
- deadlinemiss: bool
+ earliest_arrival() : Time
+ latest_arrival() : Time
+ get_priority() : Time
+ get_deadline () : Time
+ maximal/minimal_cost() : Time
+ get_deadine_miss() : bool
explore
- make_initial_state()
- not_done: bool
- next_state() : State& s
- earliest_start_time (): Time
- earliest_job_release(): Time
- latest_finish_time(): Time
- schedule(s, j)
- done_with_current_state()
State_Space
+ problem
+ Workload
+ State
+ get_finish_time(Workload) : Interval<Time>
+ get_reduced_job() : std::vector<std::size_t>
+ is_schedulable() : bool
+ was_time_out() : bool
+ num_of_states() : unsigned long
+ num_of_edges() : unsigned long
+ max_exploration_front_width() : unsigned long
+ get_cpu_time() : double
+ index_of(const Job<Time>& j) : (std::size_t) (&j - &(jobs[0]))
+ get_ce_edges(): std::deque<EdgeCE>
non preemptive test
+ operations = parser. parse_args(argc, argv)
+ process_file(parse_args)
Produce
CE_edges
- Job<Time>* scheduled
- State* source
- State* target
- Interval<Time> finish_range
+ latest_start_time(): Time

Figure 5.2: UML class diagram for *second step* implementation. The `CE_edges` as a structure contains the job, the source of this job and the target of this job. We can use the `finish_range` produced in the second step as part of the eligible condition in the third step implementation.

Table 5.3 is an example of the output produced by our implementation of the second step of our algorithm. It has five columns in total. In the output table, the deadline missed job J_7 (task 1) only appears once, satisfying the second step algorithm we designed in the previous chapter.

from	to	task	job	latest start time
0	1	1	1	30
0	2	1	2	30
0	3	1	3	29
1	4	1	2	45
1	5	1	3	29
2	4	1	1	50
2	6	1	3	34
2	7	1	4	34
3	5	1	1	45
3	6	1	2	34
4	8	1	3	65
4	9	1	4	39
5	8	1	2	60
6	8	1	1	54
6	10	1	4	34
7	9	1	1	49
8	11	1	4	81
8	12	1	5	79
9	11	1	3	64
10	11	1	1	49
11	13	1	5	96
12	13	1	4	98
12	14	1	6	79
13	15	1	6	115
14	15	1	4	95
14	16	1	8	79
15	18	1	8	84
15	17	1	7	131

Table 5.2: Above is an example of compressed SAG from the second step. The last row in a CSV file always represents a missed deadline job. As we can see, the deadline missed job J_7 has been dispatched from state v_{15} to state v_{17}.

5.3 Third step implementation

The third step of our three-steps algorithm is implemented as an entirely new C++ program. We use the outputs from the first and second step as inputs for the third step. In contrast to the first and second steps, the third step uses a depth first approach to implement the plan state searching algorithm. When the program finds one deadline missed execution scenario, it will stop and produce a CSV file recording that execution scenario as the final output. The UML class diagram for the third step is illustrated in the Figure 5.3. As part of the third step, we need to import two CSV files into the program (that is taken care of by the two procedures in the Problem class). The first is the candidate counterexample job set generated from the first step, and the second is the output from the second step. The third step output is shown in table 5.3.

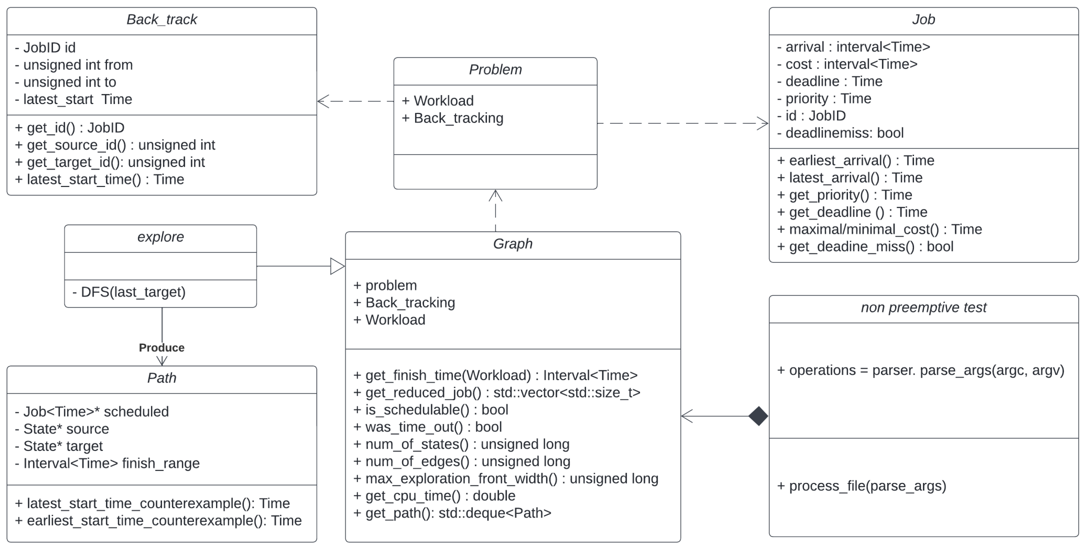

Back_track
- JobID id
- unsigned int from
- unsigned int to
- latest_start Time
+ get_id() : JobID
+ get_source_id() : unsigned int
+ get_target_id(): unsigned int
+ latest_start_time() : Time
Problem
+ Workload
+ Back_tracking
Job
- arrival : interval<Time>
- cost : interval<Time>
- deadline : Time
- priority : Time
- id : JobID
- deadlinemiss: bool
+ earliest_arrival() : Time
+ latest_arrival() : Time
+ get_priority() : Time
+ get_deadline () : Time
+ maximal/minimal_cost() : Time
+ get_deadine_miss() : bool
explore
- DFS(last_target)
Graph
+ problem
+ Back_tracking
+ Workload
+ get_finish_time(Workload) : Interval<Time>
+ get_reduced_job() : std::vector<std::size_t>
+ is_schedulable() : bool
+ was_time_out() : bool
+ num_of_states() : unsigned long
+ num_of_edges() : unsigned long
+ max_exploration_front_width() : unsigned long
+ get_cpu_time() : double
+ get_path(): std::deque<Path>
Produce
Path
- Job<Time>* scheduled
- State* source
- State* target
- Interval<Time> finish_range
+ latest_start_time_counterexample(): Time
+ earliest_start_time_counterexample(): Time
non preemptive test
+ operations = parser. parse_args(argc, argv)
+ process_file(parse_args)

Figure 5.3: UML class diagram for *third step* implementation

Below is the table that contains a set of deadline miss execution scenarios, the execution sequence that leads to a deadline miss is $J_7 \rightarrow J_6 \rightarrow J_5 \rightarrow J_4 \rightarrow J_3 \rightarrow J_2 \rightarrow J_1$, For each job in that sequence, we also output the minimum and maximum execution time, and the earliest and latest release time of the job such that a deadline miss occurs.

Task ID	Job ID	r min	r max	cost min	cost max	earliest start time	latest start time
1	7	60	85	7	15	126	131
1	6	50	60	4	16	110	115
1	5	40	45	13	19	91	96
1	4	30	80	10	15	76	81
1	3	0	40	12	16	60	65
1	2	0	30	15	20	40	45
1	1	0	35	10	15	25	30

Table 5.3: The third step result.

For instance, in the example of table 5.3, J_7 will miss its deadline in the execution scenario of Figure 5.4(a) where J_7 is released at 85 and executes for 15, but also in an execution scenario as shown in Figure 5.4(b) where J_7 is released at 70 and executes for 14.

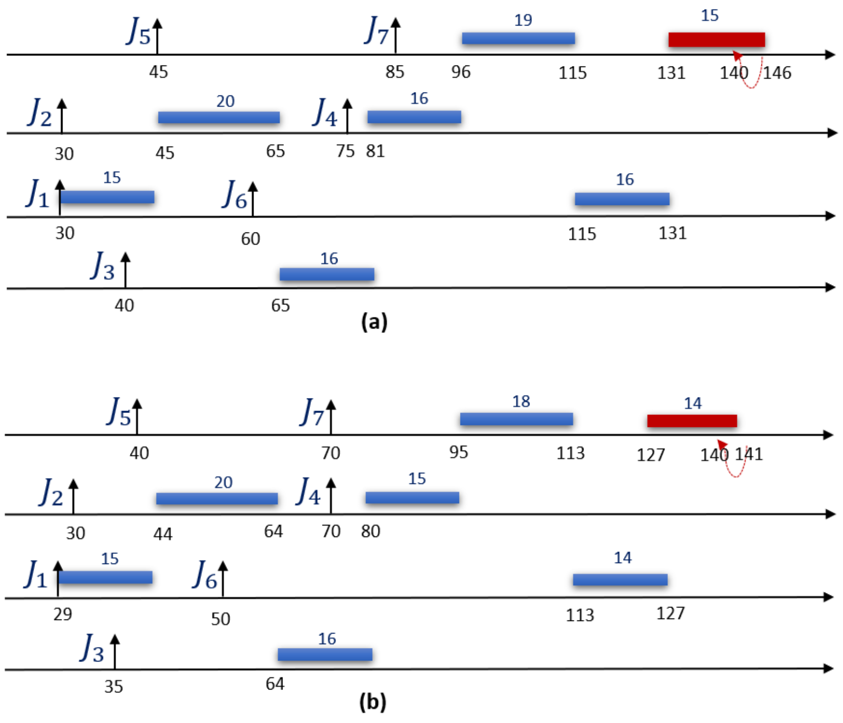


Figure 5.4: Based on the table 5.3, different execution scenarios were created

Chapter 6

Empirical Evaluation

In this chapter, we performed a large-scale analysis to evaluate the accuracy and run-time for the implementations of three steps algorithms.

We generated task sets by modifying the established method of Stafford's RandFixSum algorithm [17]. This algorithm is implemented as a python script, which requires the python 2.7 environment. The task mode generated by Stafford's RandFixSum algorithm is sporadic task model. A sporadic taskset consists of n tasks that have the following properties: period or minimum interarrival time T_i, worst-case execution time C_i and deadline D_i. Task utilisation is expressed as $U_i = C_i/T_i$. The taskset parameters that are used to understand and analyze the behavior of scheduling algorithms are

- taskset cardinality n
- total taskset utilisation u.

As a result of this algorithm, utilisation values of U_i are chosen, and a sum for n tasks is used to achieve the target total utilisation ($\sum_{i=1}^{n} U_i = u$). By using the Python script, this can be applied directly to the problem of task utilisation generation with a chosen constant total taskset utilisation. After periods have also been generated, worst-case execution times can be calculated by $C_i = U_i T_i$

To minimize difference in the number of schedulable tasksets at lower utilisation values over the different taskset sizes, a log-uniform distribution of task periods can be used, with tasksets generated for different ratios of the minimum (T_{min}) to the maximum (T_{max}) task period [17].

We generated 20 task sets for each configuration of release jitter (with upper limit is 25% of the period), execution-time variation(with upper limit of offset is period -1), and Nine different utilization values from 0.1 to 0.9. To avoid situations where the hyperperiod is unpractically long due to incompatible task periods, we chose periods following a log-uniform distribution in the range [1000, 10000].

Experiments were conducted on the Dutch SURFsara Cartesius cluster (Snellius). We implemented the analysis in C++ as a single-threaded program. The measured metrics are defined as follows:

- **CPU run-time**: The CPU run-time indicates the time that elapses between the moment the job set is started and then is finished. This is measured by seconds. The purpose of this parameter is to verify the optimization performance of our three-step algorithm. The improved results can be obtained by dividing the CPU time required for the original analysis by the CPU time required for our implementation.

- **Memory Consumption**: The memory consumption is measured the memory usage for each job sets. Here, memory usage is measured as a peak (divided by 1024). This reports as megabytes. We intend to optimize the memory consumption by our three-steps algorithm.
- **The number of jobs in the job set**: The first step is limiting our attention to a reduced job set, therefore we can avoid considering the whole job set J. The first step performance can be measured by the number of job in the job set.
- **The number of edges that were discovered**: The CPU run-time and Memory Consumption are influenced by the number of edges generated on the process. Therefore, it is a different way to calculate how much memory and run-time will be required.

6.1 First step analysis

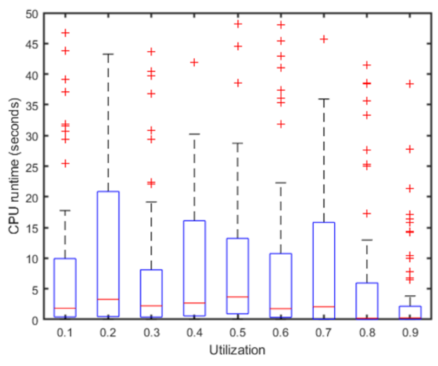
The CPU run-time for the first step of the modified SAG implementation is depicted in Figure 6.1 (a).

Observation 6.1.1. The higher utilization, i.e., $u = 0.8$ and $u = 0.9$, show a significant difference in runtime compared to the other utilization.

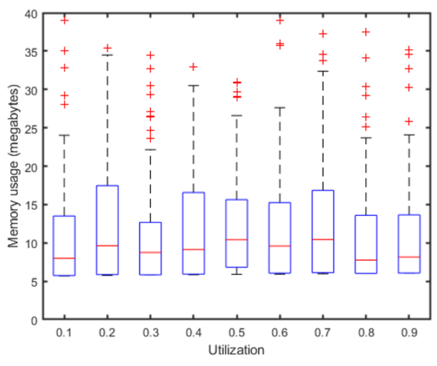
This performance can be explained by the fact that the higher the utilization, the task set is deemed unschedulable since an execution scenario containing a deadline miss was encountered. As soon as a deadline miss occurs, our analysis of this task set is stopped. Consequently, lower utilization generates mostly scheduleable task sets, which tend to increase runtime. Furthermore, the maximum run time produced by $u = 0.5$ is 562.31. The other maximum run time given by $u = 0.4$ is 322.73.

Observation 6.1.2. The higher utilization, i.e., $u = 0.8$ and $u = 0.9$, has a low memory consumption compared to the rest.

The memory usage box-plot can be found in Figure 6.1 (b), $u = 0.5$ consumed considerable amount of memory (122.69 megabytes), which makes sense as it took a large amount of run-time. On the contrary, $u = 0.8$ and $u = 0.9$ of the first step implementation have a low memory consumption. Because the job set is deemed unschedulable in the higher utilization, an execution scenario of J leads to a job exhibiting a response time exceeding its deadline. In other words, with a further increase in utilization, runtime and memory consumption of the analysis decrease since more job sets become unschedulable early in the process.



(a) CPU run-time for Task = 20, first step

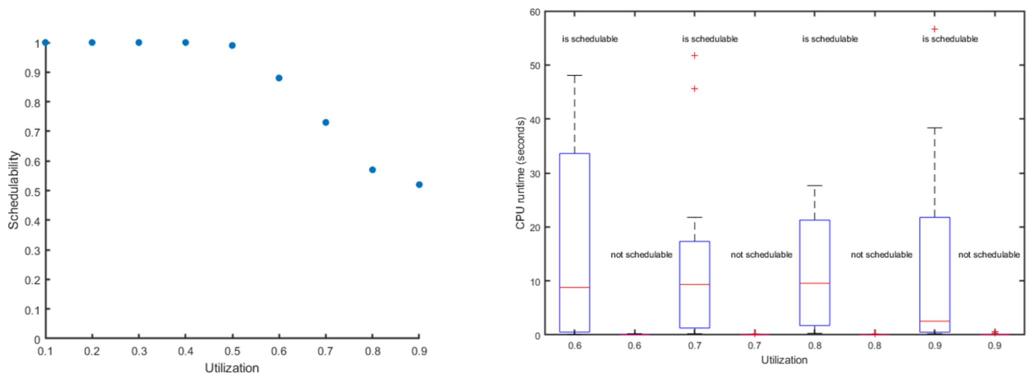


(b) Memory usage for Task = 20, first step

Figure 6.1: Run-time and memory consumption results for the first step.

Observation 6.1.3. With a further increase in utilization (start at $u = 0.6$), more job sets become unschedulable.

By conducting the first step, we can find the impact of utilization on the schedulability ratio (i.e., the percentage of task sets deemed schedulable by the analysis). Whenever a job set experience a potential deadline miss under the given scheduling policy, it generates the candidate counterexample job set as CSV file. Figure 6.2 (a) shows the result of schedulability ratio relative to the change in utilization. It can be seen the schedulability ratio decreased with a further increase in utilization, and the schedulability ratio of $u = 0.1$, $u = 0.2$, $u = 0.3$, $u = 0.4$ are equal to 1, meaning all the job sets are deemed to deemed schedulable, while schedulability ratio of $u = 0.8$, $u = 0.9$ is close to 50% (table 6.1) due to high system utilization.



(a) Schedulability results

(b) Run time comparison of schedulable and not schedulable job sets, first step

Figure 6.2: Schedulability ratio result

Utilization	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
# non-schedulable	0	0	0	0	1	12	27	43	48
# tests	100	100	100	100	100	100	100	100	100
Schedulability ratio	100%	100%	100%	100%	99.00%	88.00%	73.00%	57.00%	52.00%

Table 6.1: The schedulability ratio for 20 tasks when system utilization varies.

Observation 6.1.4. The job observes a deadline miss always happens at an early stage of the system.

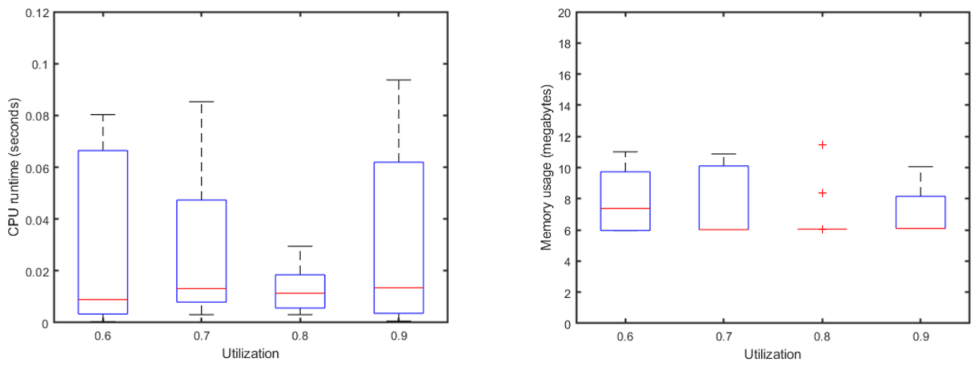
Our modification of the original schedule abstraction graph analysis checks whether timing requirements are respected at every system state of the exploration. Whenever an execution scenario containing a deadline miss occurs, our system immediately stops the exploration and marks the job set as unscheduled. This results in a decrease in runtime as more job sets are found to be unscheduled at an early system stage. Comparing the two results of the first step implementation, *schedulable* and *unschedulable*, it is expected that the *schedulable* result would require significantly larger run-time than the *unschedulable* depicted in figure 6.2 (b).

6.2 Second step analysis

For the unschedulable job set, the second step produces the compressed SAG. Taking into account the schedulability ratio discussed in the previous section, we are only interested in the system utilization starting at $u = 0.6$.

Observation 6.2.1. The runtime and memory consumption vary slightly with an increase in utilization.

Figure 6.3 shows how run time and memory usage vary from system utilization for the second step algorithm. The maximum run-time is produced by $u = 0.6$. This may support our finding in the first step algorithm that the lower utilization requires a longer run-time. Figure 6.3 also shows the run-time for $u = 0.8$ and $u = 0.9$ is almost equal to the run-time for $u = 0.7$. This is interesting, since, in terms of utilization variation, these run-time differ significantly for the first step.



(a) CPU run-time for Task = 20, second step

(b) Memory usage for Task = 20, second step

Figure 6.3: Run-time and memory consumption results for the second step.

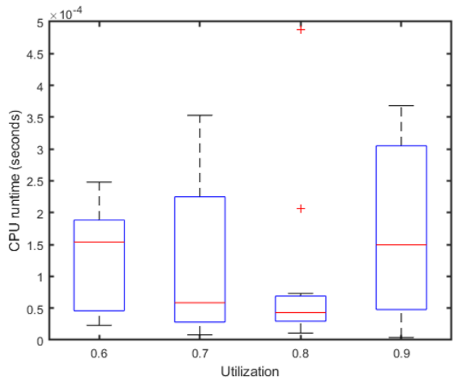
6.3 Third step analysis

The third step algorithm illustrates under which execution scenario the deadline miss occurs.

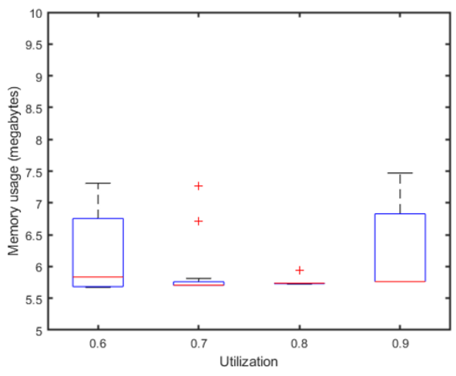
Observation 6.3.1. The runtime and memory consumption of the third step is significantly decreased compared with the two first steps algorithm (two different modifications of the original schedule abstraction graph analysis)

To implement plan-space algorithm (third step), we need to save extra features while building the schedule-abstraction graph. These features might consume a lot of memory or significantly increase the current schedule-abstraction graph's run time. Therefore, we optimize the process by dividing it into three steps instead of trying to do everything at once. Each step executes a different algorithm. They manipulate and compute different data. At the end of the third step, a scheduling example as a linear plan leading to a deadline miss is produced.

This finding appears that we have successfully optimized the process. In the third step, $u = 0.6$ also achieves maximum run-time and memory usage (11.0012 megabytes).



(a) CPU run-time for Task = 20, third step



(b) Memory usage for Task = 20, third step

Figure 6.4: Run-time and memory consumption results for the third step.

6.4 Three-steps comparison results

In this section, we show the results obtained for the three-steps algorithm. For the first step data in the plots of Figure 6.5, we only employ the unschedulable job set. In the first step, $u = 0.9$ yields 8048 of edges, while the second step yields 6547 of edges and the third step results in 100 of edges. As a result, for $u = 0.9$, the edges in the first step are 80 times greater than those in the third and 1.23 times greater than those in the second step.

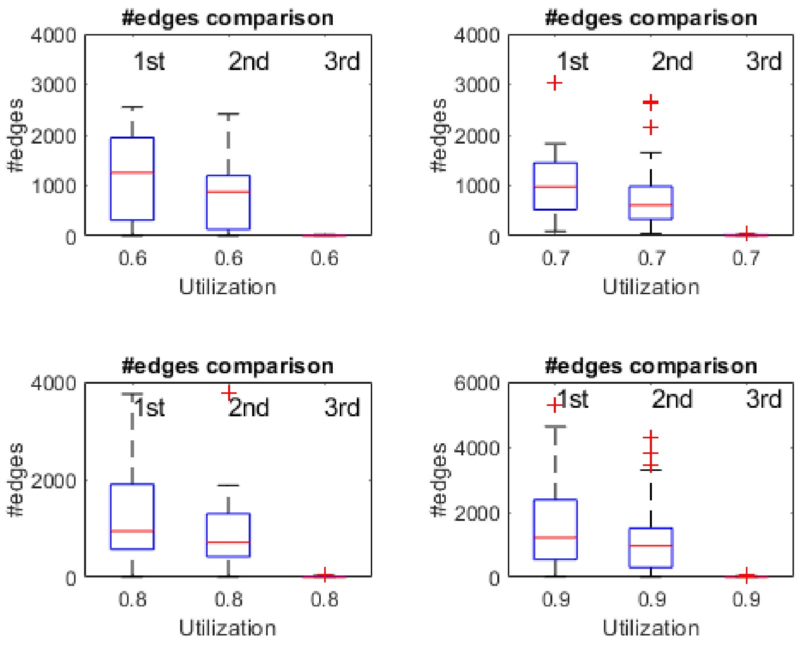
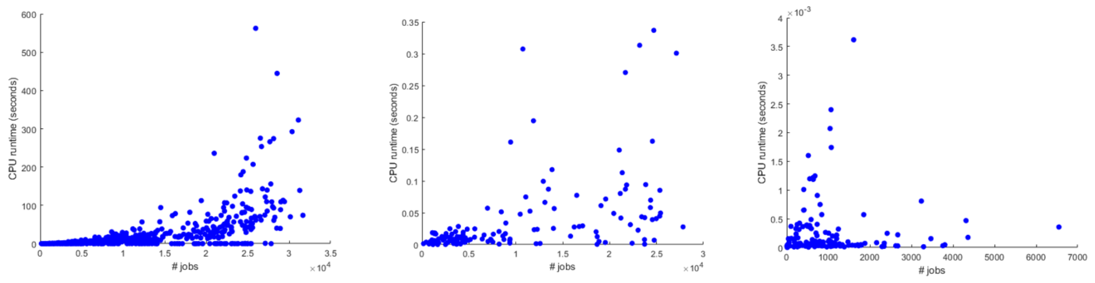


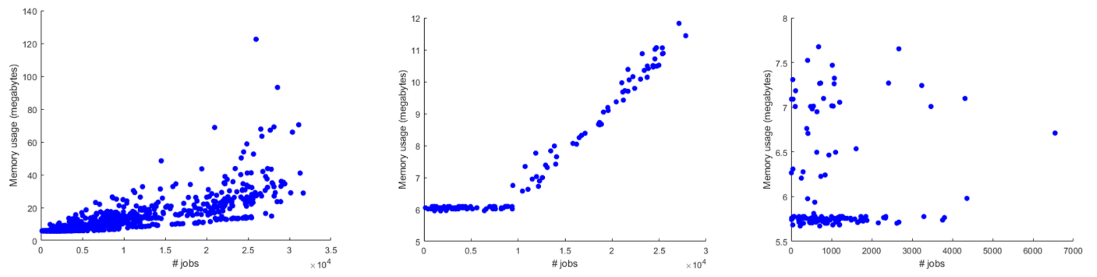
Figure 6.5: Number of edges for each implementation.

Figure 6.6 shows the impact of the number of jobs on runtime for the three-steps algorithm. The first step analysis remains sufficient for large job sets scheduled on platform. The number of jobs from the second step is not significantly reduced compared with the first step, however, the run-time has been drastically shortened. In addition to these improvements, it is very interesting to note that in the third step, both the number of jobs and the run-time significantly decreased. These trends were the same for the memory consumption shown in Figure 6.7.



(a) CPU run-time for for # jobs, first step (b) CPU run-time for for # jobs, second step (c) CPU run-time for for # jobs, third step

Figure 6.6: Three-steps experimental results, the impact of the number of jobs on runtime.



(a) Memory consumption for for # jobs, first step (b) Memory consumption for for # jobs, second step (c) Memory consumption for for # jobs, third step

Figure 6.7: Three-steps experimental results, the impact of the number of jobs on memory consumption.

Chapter 7

Conclusions and Future work

7.1 Conclusions

In this report, we introduced the concept of a partial order planning algorithm for generating the counterexample on the schedule-abstraction graph. The schedule-abstraction graph (SAG) aims to perform a reachability-based response-time analysis. The approach of SAG entails exploring the space of possible decisions that a scheduler can make for a set of jobs. However, the existing SAG will terminate the analysis as soon as a deadline miss is encountered and does not return any information that may help explain what led to such a deadline miss. Therefore, this thesis aims to present an approach that can structurally build a counterexample that provides a detailed execution scenario, including the release time, starting time and execution time of the jobs involved in the deadline miss scenario.

A partial order plan algorithm can start from the deadline missed state and proceed backwards, checking at each point whether any of the potential states can achieve the casual link onward to reach the initial state eventually. To implement this plan-space algorithm, we developed a three-steps algorithm to build counterexamples using partial order planning. Each step executes a different algorithm. According to the first step algorithm, a subset of the entire job set was identified as the candidate counterexample job set. This job set must experience a potential deadline miss under the given scheduling policy, and all execution scenarios possible with the candidate counterexample job set must also be possible with the original job set with the given scheduling policy. Then for the second step algorithm, we constructed a compressed version of the schedule abstraction graph which reduce the search space that must be explored to find deadline missed execution scenario for the third step algorithm. At the end of the third step, a scheduling example that leads to a deadline miss is then produced.

Returning to the research questions RQ1 to RQ4 listed in Section 1.3, we then presented conclusions derived from this graduation project that are summarized as answers to the research questions as follows:

- **RQ 1** How can we identify a set of paths in the schedule abstraction graph that leads to a deadline miss?

  The solution we provide involved two steps. As a first step, we proved that only jobs executed after the last idle time before the deadline actually participates to the deadline miss. Therefore, the first step of our three-steps algorithm solution identifies which jobs participated to the deadline miss (see Section 4.3). And then, in our second step algorithm (see Section 4.4), we used this limited job set to build a new graph that only contains paths (i.e., job execution orderings) that can lead to a deadline miss.

- **RQ 2** How can we generate the counterexample from a set of paths in the schedule abstraction that lead to a deadline miss?

  We used plan-space, backward searching algorithm mention is Section 4.5 and implemented in Section 5.3 to successfully build a valid counterexample which start from the last state and proceed backwards. The plan-space backward searching algorithm is implemented as the third step of our three steps algorithm. To limit the search space of the backward searching algorithm, it take as input the compressed graph generated at second step algorithm (Section 4.4).

- **RQ 3** Is there already enough information available in the current implementation of the SAG framework to build a counterexamples? If not, what is the minimal amount of information that should be added to the graph?

  The first answer is "No". Since original SAG is only kept states on the front of the graph in memory.

  Due to the lack of information on the original SAG for building the counterexample, therefore, for each new system state generated in the SAG, we had to save the set of jobs executed since the last idle state encountered in the SAG (Lemma 4.3.1). This can then be used in a second step to build a compressed version of the SAG that contains only path that may lead to a deadline miss (Section 4.4).

- **RQ 4** What techniques can be used to efficiently implement the counterexample generation feature to the existing implementation of the schedule-abstraction graph framework?

  Instead of trying to build a single program that would both do the analysis as the original SAG analysis tool and generate counterexamples at the same time, we divided the process in three steps. This allowed us to optimize the implementation and limit the amount of information that needs to be saved and manipulated at each step of the algorithm.

  Each step executed a different algorithm. The first step algorithm created a candidate candidate counterexample job set which is a subgraph of the schedule abstraction graph of the original job set. The second step produced a compressed SAG which only contain jobs returned by first step algorithm. As a result of the third step, an example of an execution scenario in which timing requirement is violated when it deems a system unschedulable is then produced.

  According to the Empire Evaluation in Chapter 6, our three-steps algorithm has proven itself to be a successful and promising technique for finding the counterexample for schedule-abstraction graph based analyses.

7.2 Future work

The table generated from the third step algorithm contains a set of deadline miss execution scenarios for the job J_i that missed its deadline D_i. However, the amount of information may overwhelm a system designer. Future work should attempt to present information in a more digestible manner, for instance showing an concrete schedule as a Gantt chart. It would help identify issues more easily and thus help system designers understand why timing violations arise and what changes to the system are required to meet associated timing requirements.

In addition, once a job J_i fails to meet its deadline, the program stops, so we only find counterexamples associated to that single job missing its deadline. This may lead to a user focusing on solving that specific deadline miss to only find out that other job may also miss their deadline later when running the analysis again. That said, the current implementation of SAG has an option that enables the system to continue if it detects that a deadline has been missed by the system. This mode may result in a number of jobs can have a deadline miss. Therefore, it would be useful for future work to find all the deadline missed jobs and the corresponding execution scenarios.

In the future, our three-steps algorithm could be extended to more complex systems. For instance, finding counterexamples when the tasks execute on multi-core platforms, uses locking protocols to protect access to shared resources, or when jobs have precedence constraints.

Acknowledgement

This work made use of the Dutch national e-infrastructure with support of the SURF Cooperative under grant agreement no EINF-3525.

Bibliography

[1] Erika Ábrahám, Bernd Becker, Christian Dehnert, Nils Jansen, Joost-Pieter Katoen, and Ralf Wimmer. Counterexample generation for discrete-time markov models: An introductory survey. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 65–121. Springer, 2014. 13

[2] Omar Al-Bataineh, Mark Reynolds, and David Rosenblum. A comparative study of decision diagrams for real-time model checking. In *International Symposium on Model Checking Software*, pages 216–234. Springer, 2018. 14

[3] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994. 15

[4] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal*, 8(5):284–292, 1993. 1, 2

[5] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. *Formal methods for the design of real-time systems*, pages 200–236, 2004. 14

[6] Enrico Bini, Giorgio C Buttazzo, and Giuseppe M Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, 2003. 1

[7] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 106–114, 2008. 14

[8] Abdeldjalil Boudjadar, Alexandre David, Jin Hyun Kim, Kim G Larsen, Marius Mikučionis, Ulrik Nyman, and Arne Skou. Hierarchical scheduling framework based on compositional analysis using uppaal. In *International Workshop on Formal Aspects of Component Software*, pages 61–78. Springer, 2013. 14

[9] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011. 1

[10] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. a survey. *IEEE transactions on Industrial Informatics*, 9(1):3–15, 2012. 1

[11] Franck Cassez and Kim Larsen. The impressive power of stopwatches. In *International Conference on Concurrency Theory*, pages 138–152. Springer, 2000. 15

[12] Rohit Chadha and Mahesh Viswanathan. A counterexample-guided abstraction-refinement framework for markov decision processes. *ACM Transactions on Computational Logic (TOCL)*, 12(1):1–49, 2010. 15

[13] Franco Cicirelli, Angelo Furfaro, Libero Nigro, and Francesco Pupo. Development of a schedulability analysis framework based on ptpn and uppaal with stopwatches. In *2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, pages 57–64. IEEE, 2012. 14, 15

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000. 15

[15] Edmund Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 19–29. IEEE, 2002. 15

[16] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007. 1, 2

[17] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010. 47

[18] E Allen Emerson and Joseph Y Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986. 13

[19] A.N. Fredette and R. Cleaveland. Rtsl: a language for real-time schedulability analysis. In *1993 Proceedings Real-Time Systems Symposium*, pages 274–283, 1993. 14

[20] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice.* Elsevier, 2004. 16, 23

[21] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 263–272. Springer, 2007. 1

[22] Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536–546, 2011. 1

[23] Pujie Han, Zhengjun Zhai, Brian Nielsen, Ulrik Nyman, and Martin Kristjansen. Schedulability analysis of distributed multicore avionics systems with uppaal. *Journal of Aerospace Information Systems*, 16(11):473–499, 2019. 14

[24] Arut Prakash Kaleeswaran, Arne Nordmann, Thomas Vogel, and Lars Grunske. A systematic literature review on counterexample explanation. *Information and Software Technology*, page 106800, 2021. 13, 15

[25] Erez Karpas, David Wang, Brian C Williams, and Patrik Haslum. Temporal landmarks: What must happen, and when. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*, 2015. 15

[26] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1):134–152, 1997. 14

[27] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973. 1

[28] Marius Mikučionis, Kim Guldstrand Larsen, Jacob Illum Rasmussen, Brian Nielsen, Arne Skou, Steen Ulrik Palm, Jan Storbank Pedersen, and Poul Hougaard. Schedulability analysis using uppaal: Herschel-planck case study. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 175–190. Springer, 2010. 14, 15

[29] Mitra Nasri and Bjorn B Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23. IEEE, 2017. 1, 2, 5, 15, 17, 20, 24, 27, 31, 41, 42

[30] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. A response-time analysis for non-preemptive job sets under global scheduling. In *30th Euromicro Conference on Real-Time Systems*, pages 9–1, 2018. 1, 2

[31] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. Response-time analysis of limited-preemptive parallel dag tasks under global scheduling. In *31st Conference on Real-Time Systems*, pages 21–1, 2019. 1, 2, 6, 15, 17, 18, 28

[32] Mitra Nasri, Geoffrey Nelissen, and Gerhard Fohler. A new approach for limited preemptive scheduling in systems with preemption overhead. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25–35. IEEE, 2016. 1

[33] Suhail Nogd, Geoffrey Nelissen, Mitra Nasri, and Björn B Brandenburg. Response-time analysis for non-preemptive global scheduling with fifo spin locks. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 115–127. IEEE, 2020. 2

[34] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977. 13

[35] Sayra Ranjha. Partial-order reduction in reachability-based response-time analyses. 2021. 2, 18

[36] Sayra Ranjha, Mitra Nasri, and Geoffrey Nelissen. Work-in-progress: Partial-order reduction in reachability-based response-time analyses. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 544–547. IEEE, 2021. 2

[37] Youcheng Sun and Giuseppe Lipari. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor global fixed-priority scheduling. *Real-Time Systems*, 52(3):323–355, 2016. 1, 14

[38] Romain Trinquart. Analyzing reachability within plan space. In *Proc. of the*, 2003. 16

[39] Steve Vestal. Modeling and verification of real-time software using extended linear hybrid automata. In *NASA CONFERENCE PUBLICATION*, pages 83–94. NASA; 1998, 2000. 14

[40] Beyazit Yalcinkaya, Mitra Nasri, and Björn B Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1228–1233. IEEE, 2019. 1, 2, 15

Appendix A

Installation and Configuration

The tool is able to work on Linux system, thus we installed the Linux Mint (Ubuntu Focal), version 20.3 as the operating system for the laptop computer. The following command to complete the C++ compiler:

```
sudo apt install clang
```

The correct command for installing the Intel Thread Building Block (TBB) library and parallel runtime:

```
sudo apt install libtbb-dev
```

To compile the tool, we first enter the build directory:

```
cd build
```

Using the command in below generates the `Makefile` with `cmake`:

```
cmake ..
```

Finally we can apply the appropriate command to build the np-schedulability-analysis

```
 make -j
```

Appendix B

Dot graph

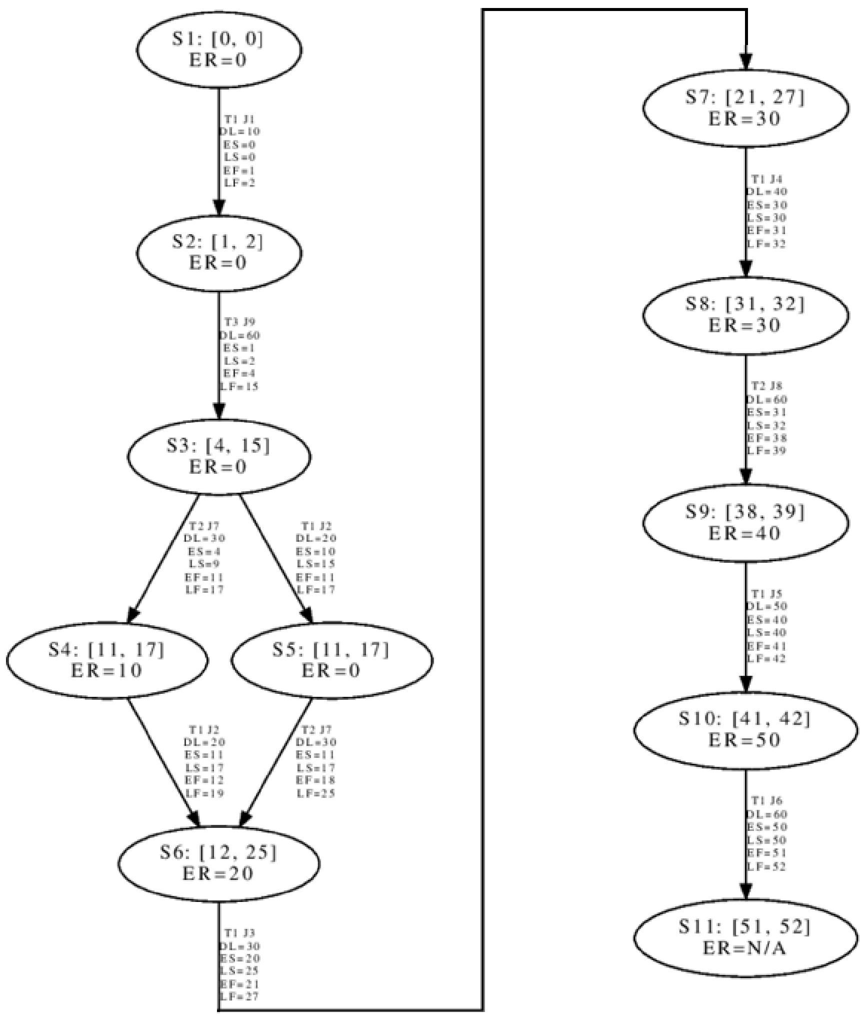


Figure B.1: The dot graph of the example `fig1c.csv`