Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2025)

Analyzing Launch Operations Using the Spaceport Throughput Analysis Resource (STAR)

Richard Levinson[1], **Vijaykumar Baskaran** [1], **Jeffrey Brink**[2], **Jeremy Frank**[3]

[1] KBR, Intelligent Systems Division, NASA Ames Research Center
[2]Spaceport Management and Integration, NASA Kennedy Space Center
[3]Intelligent Systems Division, NASA Ames Research Center

Abstract

We describe the development of the Spaceport Throughput Analysis Resource (STAR), which evaluates Kennedy Space Center spaceport launch throughput. STAR integrates simulation and limited rescheduling, using a constraint programming model to check constraints and reschedule events. The outputs of STAR are launch delays, and the constraint violations leading to those delays, that can be used to inform infrastructure investments to reduce future delays. At STAR's core is a constraint program representing a limited horizon scheduling problem used to identify resource constraint violations, and revise schedules in the presence of unexpected events that disrupt the launch schedule. We describe the design and implementation of STAR using illustrative examples, and describe performance results on use cases showing how increased launch rates stress KSC infrastructure.

1 Introduction

NASA's Kennedy Space Center (KSC) and Cape Canaveral Space Force Station (CCSFS) are the world's preeminent multi-user spaceport, providing facilities and launch capabilities to the agency, NASA's commercial partners, and other government agencies. KSC ensures an environment in which NASA's programs and other users can safely and effectively carry out their operations. Ninety-six launches took place in 2024, with more than 150 launches expected in 2025 [1]. KSC meets the needs of customers who request launches based on their own schedules, but who may not be aware of KSC-wide resource limitations and external constraints on launch operations. KSC-wide resources include telecommunications, tracking, commodities such as Helium and Nitrogen, and special equipment needed to support launches. Constraints include seasonal operations limitations and scheduled maintenance. Since launch operations are complex and uncertain, unexpected events can also cause delays in launch and supporting operations.

KSC stakeholders need to answer the following question: what KSC-wide constraints cause rescheduling, and how often? The Spaceport Throughput Analysis Resource (STAR) assesses whether a proposed set of launches and associated activities can be performed given the resources KSC currently has available, and external constraints imposed on KSC operations. STAR integrates short-horizon scheduling and simulation of launches and unexpected events that disrupt the launch schedule, driven by user-configurable probabilities of different classes of event outcomes, including delays and worst-case use of resources. STAR records the *constraint violations* leading to delays, informing stakeholders of key infrastructure limitations preventing customers from being able to perform their missions as desired, and giving insight into how to improve spaceport throughput.

The rest of the paper is organized as follows. In Section 2 we give an overview of KSC spaceport operations. In Section 3 we formally describe the scheduling problem ingredients. In Section 4 we describe how manifests are simulated. In Section 5 we describe the constraints problem that is solved when rescheduling. In Section 6 we empirically evaluate STAR run-time on KSC-provided benchmark problems. In Section 7 we describe related work. Finally, in Section 8, we conclude and describe future work.

2 Overview

We begin with describing KSC spaceport operations. A *manifest*, or launch manifest, consists of a set of *launches* of different launch vehicles. Each launch may have an associated *supporting event* preceding the launch. We refer to launches and supporting events collectively as *events*. An *initial manifest* assigns launch events to notional launch dates and times. If present, supporting events precede the launch event by a minimum duration based on the launch vehicle type. These initial manifests are desired launch dates generated by *customers*, and represent the ability of a customer to launch their own vehicles according to their internal scheduling constraints, as opposed to those imposed by KSC. Events use different *resources*, and are subject to a variety of additional *constraints*, which will typically require rescheduling the launches to resolve conflicts in the initial manifest, and as a result of delays due to unexpected events.

KSC's resource use and constraints are quite complex, so the problem is akin to the Resource Constrained Project Scheduling Problem (RCPSP) (Blazewicz, Lenstra, and Kan 1983). However, STAR's job is not to generate schedules, but rather to *identify KSC infrastructure constraints* that lead to delays during the execution of manifests. For this reason,

[1]https://www.orlandosentinel.com/2025/01/29/space-coast-launch-schedule/

STAR interleaves the simulation of scheduled events, potentially resulting in delays of those events, and re-scheduling over a myopic time horizon to satisfy constraints. STAR minimizes schedule makespan when rescheduling after a constraint violation. We distinguish between the manifest and the *run*, which records events the day and hour that they occur, along with unexpected events and constraint violations that cause delays. Delays, reflected by the difference between the initial manifest date and actual event date, translate to lack of customer satisfaction. Constraints are divided into customer-centric *minimum-spacing* constraints, which are enforced during rescheduling but are not implicated as causing launch delays, and *infrastructure constraints*, which are tracked as the primary cause of a launch delay.

2.1 Resources

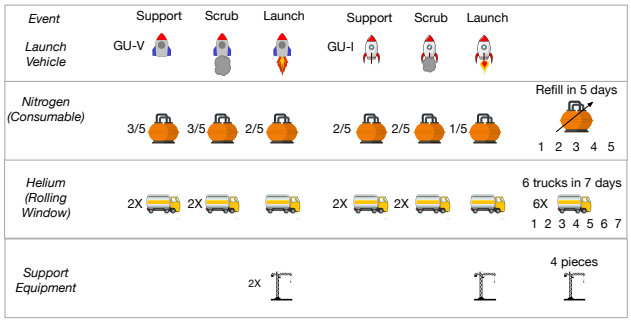


Figure 1: Hypothetical launch and supporting events, resources, and resource usage used to illustrate STAR.

As is typical in scheduling problems, resources come in many different varieties. Some KSC-wide resources are *reusable*; they are used and then returned. An example is special equipment, available in limited numbers, that must be used during a launch and subsequently reconfigured between launches. For instance, there may be 4 pieces of such equipment, and each piece of equipment may require 4 days to reconfigure after a launch is successfully completed.

Some resources are *consumable*; events use these resources, and they are replenished at a fixed daily rate. For example, events may use between $\frac{1}{5}$ and $\frac{3}{5}$ of KSC's stored Nitrogen, which can be fully replenished in 5 days.

Lastly, some resources have *rolling limits*; that is, they can be replenished up to a fixed amount over a fixed number of days. These rolling limits are an abstraction of a maximum rate of replenishment. For instance, Helium may be resupplied at a rate of at most 6 truckloads every 7 days. All 6 truckloads may be delivered any time (including all on the same day!) during the rolling 7 day period. But once 6 trucks have arrived, no more Helium can be delivered until enough time has passed. Multiple rolling limits may apply to the same resource; Helium may be resupplied at a rate of at most 6 trucks every 7 days, and at most 12 trucks every 20 days. An additional Yearly Rolling Limit constrains the maximum number of times a rolling limit can be reached in a year. For example, the 12 trucks every 20 days limit can be reached at most once a year. Once 12 trucks are required in a 20 day rolling period, at most 11 trucks can be used in every 20 day period thereafter for the remainder of the year.

We use two hypothetical launch vehicles in examples throughout the paper. Events use pieces of equipment to support launches, stored Nitrogen, and truckloads of Helium with a single rolling limit. As shown in Figure 1, one launch vehicle is a Go-Upper-One (GU-I). Each launch consumes $\frac{1}{5}$ of the Nitrogen storage tank capacity and 1 truck worth Helium, and uses 1 piece of supporting equipment. The supporting event uses $\frac{2}{5}$ of the Nitrogen storage tank capacity, and 2 trucks worth Helium, and occurs one day before the launch. The second vehicle is a Go-Upper-Five (GU-V)[2]. The launch uses $\frac{2}{5}$ of the Nitrogen tank and 1 truck worth Helium, and 2 pieces of supporting equipment. The supporting event uses $\frac{3}{5}$ of the Nitrogen tank and 2 trucks worth Helium, and occurs 3 days before the launch.

2.2 Scrubs

Due to uncertainties involved in spaceport operations, both launches and supporting events may be delayed or *scrubbed*. A scrub may use resources; scrubbed launches and supporting events use the same amount of resources. Scrub resource use is usually higher than that of either a supporting event or a launch. To continue with our example, Figure 1 shows the GU-V launch uses $\frac{2}{5}$ of the Nitrogen tank and 1 truck worth of Helium, and 2 pieces of supporting equipment. A scrub uses $\frac{3}{5}$ of the Nitrogen tank and 2 trucks worth of Helium (the same amount of Helium and Nitrogen as the supporting event). Events in the manifest are assumed to use the worst-case resource usage. That means in our previous example, checking Nitrogen consumption for a GU-V launch would assume usage of $\frac{3}{5}$ of the Nitrogen tank and 2 trucks worth of Helium. Once the event has been deemed successful, the actual resource consumption of the launch ($\frac{2}{5}$ of the Nitrogen) is tracked in the run. The tracking of scrubs in the manifest is shown for the consumable Helium resource, and protecting against scrubs in the manifest, is also shown in Figure 2 (left); the dotted line shows how much more resource would be used if a scrub that uses resources occurs instead. When a scrub occurs, the scrub is recorded in the run, and the event is attempted again until it succeeds.

2.3 Resource Examples

In this section we explain resource behavior using example manifests and runs. Consider the Go-Upper Five's Helium consumption. The rolling Helium limit of 6 tankers in 7 days described in the previous paragraph ensures KSC can only launch a GU-V or a GU-I every rolling 7 day period, as shown in Figure 2 (left). More precisely, since events use 2 trucks in the worst case, and the limit is 6 trucks in 7 days, any 7 day period can contain 3 events (launch or supporting) in any order. However, a total of 4 events are needed to perform 2 launches. An example of how the launches, separation of launch and supporting events, and resource consumption and replenishment of the consumable Nitrogen resource

[2]With love and respect to Randall Munroe's magnificent Up-Goer-Five xkcd web comic, which can be found at https://xkcd.com/1133/

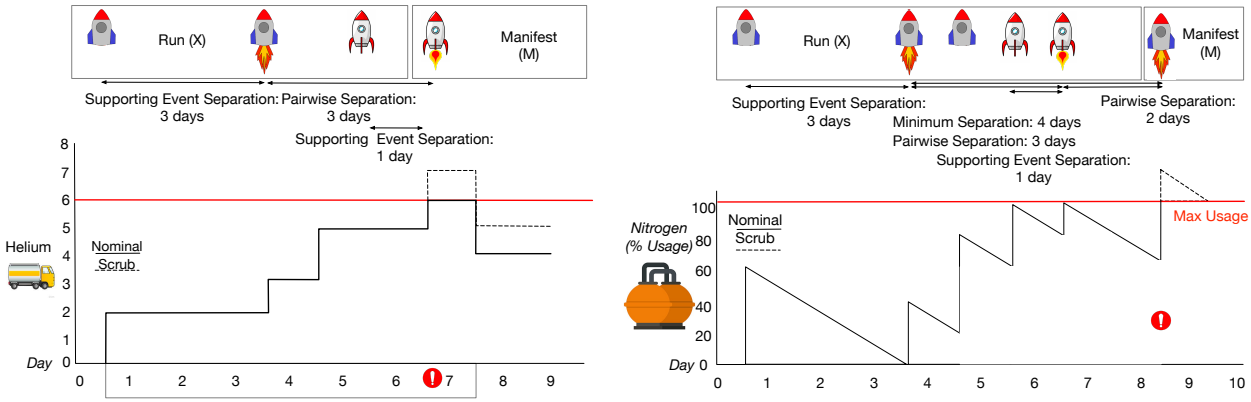


Figure 2: Tracking Rolling limits (left): Two launches are shown, a GU-V and a GU-I. The rolling limit is 6 tankers in a 7 day period; the first period is indicated at the bottom of the figure starting on day 1. The separation between main and supporting events of the GU-V is 3 days. The run shows a GU-V launch on Day 4. The GU-I supporting event on Day 6 uses 2 tankers, and is followed by its launch on Day 7. We must conservatively assume the GU-I launch uses 2 trucks if there is a scrub; the required number of Helium trucks (shown by the dotted line) violates the rolling limit on Day 7. Tracking Consumable Resources (right): Three launches are shown, one GU-I and two GU-Vs. The second GU-V launch is on day 9. The worst-case Nitrogen resource use (also shown by dotted line) of the GU-V launch on day 9 in the manifest violates the limit.

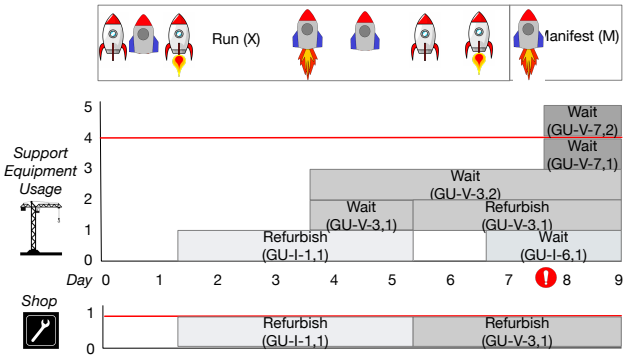


Figure 3: Tracking Reusable Resources. The run shows a GU-I launch on day 2, a GU-V launch on day 4, both of which are recorded in the run, and a GU-V launch on day 7 in the manifest. Each GU-I uses one piece of launch support equipment, while the GU-V uses two. Since the shop can only refurbish one piece of equipment at a time, by the time we get to the GU-V launch in the manifest, not enough pieces of equipment have been refurbished to support it. We label each wait and refurbish task with a launch vehicle and launch day, and an arbitrary index referring to the piece of support equipment used by that launch.

work, is shown in Figure 2 (right). Resource replenishment is continuous, at a fixed rate, while each event uses large amounts of the resource instantly.

2.4 Temporal Constraints

Launches may have customer-provided individual *launch windows* that must be respected. These windows only constrain the launch event; the supporting event can fall outside these windows. Similarly, KSC may have facility-wide launch *blackout periods* when no launches may take place. Supporting events can occur during these periods. Launch windows are specified by customers, while facility-wide launch blackouts are KSC-wide infrastructure constraints.

Temporal constraints arising due to the need to manage infrastructure across all of KSC during any launch are referred to as *KSC pairwise constraints*. Pairwise constraints propagate between launch events are coupled with constraints linking launches to the supporting events (further described in Section 3 (Formalism)). Customer-driven *minimum spacing* constraints drive the time needed to reconfigure a launch pad between launches. Pairwise temporal separation durations are *order dependent*. That is, if a GU-V is scheduled to launch before a GU-I, the required separation between launches is 3 days; if the GU-I is scheduled to launch before the GU-V, the separation is only 2 days. Examples of these pairwise constraints are also shown in Figure 2, such as the 3 day separation constraint between the first GU-V launch on day 4, and the GU-I launch on day 7. Delays due to scrubs, as well as due to rescheduling when constraints are violated, can propagate to the customer-specific minimum spacing constraints. The resulting minimum-spacing delays are referred to as *ripple effect* delays. Delays due to rescheduling are tracked separately from ripple effect delays, as discussed further in Section 4.

3 Formalism

Formally, we start with the manifest, M, containing a set of launches L. An element of the manifest consists of a launch event l_i or a supporting event s_i, and the time at which each event is scheduled, $t(l_i)$ or $t(s_i)$. We denote a generic event by e_i and the scheduled time of the event by $t(e_i)$.

We denote the set of rolling limit constrained resources by O, the set of consumable resources by C, and the set of reusable resources by R. Each event uses some amount of the specified resources. All resources are consumed at the

hour on the date the event occurs. We denote the amount of resource used by event e_i as follows: $o_j(e_i)$ denotes use of rolling resource o_j, $c_j(e_i)$ denotes use of consumable c_j, and $r_j(e_i)$ denotes the use of reusable resource r_j. We will describe the notation of worst-case usage of an event prior to simulation, and the actual usage of events as a result of simulation outcome in Section 4 (Manifest Simulation). Consumable resources are replenished at an hourly production rate, up to a maximum capacity. We denote by c_i^m the maximum capacity of consumable resource $c_i \in C$, and $\delta(c_i)$ the hourly rate of replenishment of resource c_i.

Limits on rolling resources constrain the amount of resource that can be used in a rolling daily period. Recall resource o_i can be subject to multiple rolling limits. Each such limit is denoted $o_{i,j}$, and consists of $\texttt{win}_j(o_i)$, the j^{th} rolling time window size, and $o_i w_j^m$, the maximum resource use limit on o_i associated with the j^{th} window. Yearly limits are expressed as a number of times a specific rolling limit can be reached (not exceeded) in a year. We denote by $o_i y_j^m$ the number of times the j^{th} rolling limit on resource o_i can be reached in a year. When the j^{th} rolling limit is reached $o_i y_j^m$ times, $o_i w_j^m$ is reduced to $o_i w_j^{m'}$ until the next year. We abuse notation and say $o_{i,j} \in O$. Some rolling limits only apply when one of a specific set of launches $L_j(o_i) \subset L$ has taken place in the window.

Reusable resources are used for a period of time, and then returned. The duration of resource use is a function of the schedule. We denote by r_i^m the maximum capacity of reusable resource r_i. Reusable resources in STAR are complicated due to the need to refurbish each piece of equipment between launches. A single shop refurbishes these reusable resources, and it can refurbish a limited number of pieces of support equipment at a time (i.e. the shop is itself a reusable resource of limited capacity). Modeling reusable resource usage and refurbishment is accomplished using an additional set of tasks. We denote reusable resource r_1 as the pool of launch support equipment, and resource r_2 as the shop. If l_i uses $r_1(l_i)$ pieces of support equipment, we create $r_1(l_i)$ refurbishment jobs and associated constraints. Each refurbishment job for the j^{th} piece of support equipment used for launch l_i consists of a wait task w_{ji} and the refurbishment task itself, b_{ji}. Both tasks use r_1, as does the launch itself, but the b_{ji} task only uses r_2, the shop. The set of wait tasks w_{ji} for l_i is denoted W_i and the set of refurbishment tasks b_{ji} is denoted B_i. The duration of b_{ji} is fixed to a known constant bd. The duration of the w_{ji} needs to be computed given the constraints and an objective. We bundle the wait task duration with the launch hour itself, so wait task w_{ji} starts at $t(l_i)$ and ends when the shop starts to service this piece of support equipment. Refurbish task b_{ji} starts at the time w_{ji} ends, lasts bd days, and uses the shop until it is finished. Both resources r_1 and r_2 are returned when b_{ji} ends, as is normal for reusable resources. A run and manifest tracking the Reusable resources is shown in Figure 3.

We denote by K the set of KSC infrastructure pairwise constraints; there may be several such constraints between a pair of launches l_i and l_j, so each constraint is denoted $k_h(l_i, l_j)$. The separation time between launches is order dependent, i.e. $t(l_j) > t(l_i) \Rightarrow t(l_j) - t(l_i) \geq k_{hji}$ and $t(l_j) < t(l_i) \Rightarrow t(l_i) - t(l_j) \geq k_{hij}$.

Let P be the set of customer-provided pairwise constraints. At most one such constraints exist between a pair of launches. These constraints can also be order dependent. If such a constraint exists, it is denoted by $p(l_i, l_j)$ and has a similar form to $k_h(l_i, l_j)$.

The set of constraints defining the separation between a launch event and supporting event is denoted D; the separation for an individual launch event l_i and its supporting event s_i is denoted d_i and has the form $t(l_i) - t(s_i) \in [d_{i,l}, d_{i,u}]$. Let N be the set of customer-provided windows for each launch. The set of launch windows for l_i is denoted n_i. Since a launch may have multiple windows, the j^{th} window for launch l_i is denoted n_{ji} with bounds $[n_{ji,l}, n_{ji,u}]$. The constraint has the form $\vee_j(t(l_i) \in [n_{ji,l}, n_{ji,u}])$.

Denote scrubs by a_i, and resource usage of the scrubs $o_i(a_i)$ or $c_i(a_i)$ as appropriate. Reusable resources are not used during scrubs. The time a scrub occurs is denoted $t(a_i)$. A scrub is just another event, so e_i can refer to a scrub, and $t(e_i)$ can refer to the time of a scrub.

We denote the run by X. The run contains the simulated manifest events, the times of those events, and related resource use generated during the simulation of M.

Let Z refer to the set of KSC-wide blackout periods. Each blackout period z_i has bounds $[z_{i,l}, z_{i,u}]$. No launches are permitted during these periods. Finally, let H refer to the earliest and latest date of any launch in M.

4 Manifest Simulation

Simulation begins with an initial manifest M of fixed times $t(e_i)$ and $t(s_i)$ (i.e. an initial schedule). This manifest is guaranteed not to violate any of the pairwise constraints in H, N, P, D. Other constraints (pairwise constraints in K or resource constraints) may be violated by the initial manifest. STAR simulates the manifest, generates unexpected events, and records delays and the reasons for them in the run.

We use the convention that the current event from M being simulated is always indexed by n; we refer to this event as e_n. If e_n is a supporting event s_n, then l_n refers to the launch in the manifest associated with this supporting event. Also by convention, if l_n is the current event in the manifest that is being simulated, then s_n refers to the supporting event (by definition in the run, since it has already been simulated) associated with this launch. Events in the manifest are indexed by i, thus, e_i.

STAR processes events in M in chronological order imposed by the manifest. The order in the initial manifest is modified by delays and rescheduling. As described in the introduction, STAR simulates the current event e_n to determine whether the event is scrubbed, or occurs at $t(e_n)$. The consumable and rolling window resources are consumed when there is a scrub, but the pad support equipment is not used, meaning these resources don't need to be refurbished, and are available for subsequent launches. The times of scrubs and associated resource use are added to the run X; the times of scrubs which do not incur resource use are also included in the run to aid in computing delays. The scrubbed

event is delayed some number of days into the future, and is inserted back into the manifest at the new time resulting from the delay. Events violating constraints are rescheduled, also resulting in delays.

At the time at which e_n is simulated, we must assume this event is a scrub that uses resources. Recall that supporting events can scrub and use resources. Thus, when checking for infrastructure violations and subsequent rescheduling, rolling resource $o_j(e_n)$ and consumable $c_j(e_n)$ are assumed to use their worst-case values, i.e. the consumption in the event of a scrub. When an event succeeds and is inserted into run X, we store the nominal resource usage in X. Thus, $o_j(a_i)$ and $c_j(a_i)$ refer to scrubbed event resource use, and $o_j(s_i)$, $c_j(s_i)$, $o_j(l_i)$, $c_j(l_i)$ refer to nominal resource use.

For rolling limits o_i that only apply when one of a specific set of launches $L_j(o_i)$ has taken place in the window, we can use the following calculation of the resource used between $t(e_n) - \texttt{win}_j(o_i)$ and $t(e_n)$. Denote by $\texttt{amt}(o_j, t(i))$ the amount of resource consumed during this window. Let $\mathcal{W}$ be the set of events occurring in this window. If $L_j(o_i) \cap \mathcal{W} = \emptyset$ then we set $\texttt{amt}(o_j, t(i)) = 0$, otherwise we use $\texttt{amt}(o_j, t(i))$ as specified. Consumable resource replenishment is assumed to occur hourly, up to c_i^m. Denote the amount of resource c_i that has been consumed and not replenished at the time e_i was processed d by $\texttt{amt}(c_i, t(i))$. If the last event occurred at $t(e_i)$ and the current event in the manifest is scheduled at $t(e_n)$, the new amount is $\texttt{amt}(c_i, t(n)) = \max((\texttt{amt}(c_i, t(i)) + \delta(c_i)(t(e_n) - t(e_i)), c_i^m)$.

If executing event e_n at $t(e_n)$ would violate some infrastructure constraint, it is rescheduled. However, only events in the run X and e_n itself are used to check for constraint violations. If a violation is found, only e_n can be rescheduled, and it is always rescheduled without regard to constraints involving future events in M. The current event is only constrained by pairwise constraints with previously executed events, (worst case) resource consumption of e_n, and resource consumption due to previously executed events or scrubs that use resources. Due to orbital mechanics, when there is a scrub, the launch time the next day is 30 minutes earlier; we approximate this as a (multiple of) 24 hour scrub, so e_n is constrained to move integer days into the future. Scrub delays can be up to 7 days in the future.

Some events in the manifest are never rescheduled as a result of infrastructure violations. These events impose a special form of blackout before and after the event, and are an exception to the general rule above that future events are not considered when rescheduling. The set of such events are denoted M_U. For an event $e_m \in M_U$, at time $t(e_m)$, an associated blackout $uz_m \in Z$ ensures no other event can be scheduled in the blackout period. This means if e_n is rescheduled, $t(e_n)$ is prevented from occurring in the interval $[uz_{m,l}, uz_{m,u}]$. Unlike other blackouts, if the associated event moves because of weather or technical delays, then $t(e_m)$ and the associated uz_m also moves. These blackout violations are also not recorded as infrastructure violations.

When events are rescheduled or delayed, a new event time $t^*(e_n)$ is generated, and the event is inserted back into manifest M in chronological order. This means events can be processed in a very different order than they appeared in the initial manifest. It also means launches can occur much later than their supporting events if they are delayed or rescheduled after the supporting event has been executed.

Algorithm 1: SimulateManifest

Input : Manifest M
Input : Blackouts Z
Input : KSC Pairwise K
Input : Resources O, C, R
Input : Unmovables M_U
Input : 'Customer' constraints D, P, N
Input : Horizon H
Output: Run X

```
X ← ∅;
while M ≠ ∅ do
    e_n ← getNextEvent(M, e_n);
    b ← checkInfrastructureSatisfied(X, e_n);
    if (b == false) then
        recordViolations(X, e_n);
        t(e_n) ← rescheduleEvent(X, e_n);
        propagateSpacing(X, e_n, M, D, P, N);
        recordDelays(e_n, M);
    else                                   // (b == true)
        UpdateYearlyRollingLimitCount(X, e_n);
        u_n ← getEventOutcome(e_n);
        if (u_n == success) then
            getEventUsage(e_n);
            X ← X ∪ (t(e_n), e_n);
            M ← M − (e_n);
        else                               // (u_n ≠ success)
            if (u_n == reseuse) then
                getEventUsage(a_n);
                X ← X ∪ (a_n, t(e_n));
            t(e_n) ← t(e_n)+ getScrubDelay(a_n);
            propagateSpacing(X, e_n, M, D, P, N);
return violations;
```

To ensure that the manifest always respects the 'customer' constraints in D, P, N, each time events are delayed or rescheduled, all of these constraints are propagated before the next item in M is processed. The difference $d_{n,u} - d_{n,l}$ is referred to as the *supporting event buffer*, and allows for a small amount of delay of supporting events prior to a launch, without provoking further revision of the manifest. The constraints in D are propagated if $t(s_n)$ changes, and are enforced as follows: If $t(l_n) - t(s_n) \leq d_{n,l}$ then we reassign $t(l_n) = t(s_n) + d_{n,u}$, otherwise $t(l_n)$ remains unchanged.

Algorithm 1 summarizes the manifest simulation algorithm. We highlight only the most important inputs to the various functions in this algorithm. For instance, checkInfrastructureSatisfied(X,e_n) does not take as input the manifest, but propagation of spacing requires the manifest, run, time of the current event, and the set of 'customer' constraints in D, P, N. Blackouts, minimum spacing and pairwise constraints, the resource types and limits, and the

horizon, are all used by most of the functions.

STAR logs event times, scrubs, delays and resource usage. The logs allow STAR users to compute metrics from each run, including the percentage of launches delayed by vehicle type, year, or other vehicle property, the percentage of delays caused by each resource or KSC-wide pairwise constraint class in K, customer spacing constraint propagation caused by resource or K induced rescheduling (ripple effect), and the number and length of delays, including delays due to unexpected events, direct delays due to rescheduling, and ripple effect delays caused by other delays in conjunction with minimum spacing constraints.

5 Constraint Checking and Rescheduling

The problem STAR uses, both for checking infrastructure violations and for rescheduling, consists of:

1. The run X, consisting of simulated manifest events and times they occur (e_i,$t(e_i)$); these can be launches (l_i,$t(l_i)$), supporting events (s_i,$t(s_i)$) or scrubs (a_i,$t(a_i)$).
2. Current manifest event and event time (e_n,$t(e_n)$)
3. Inequality constraints of the form $t^*(e_n) \geq x_i$, derived from the minimum spacing and pairwise constraints, and as we will see below, from the consumable resources.
4. Rolling Limit Resource usage of events and scrubs, $o_i(e_i)$ $o_i(e_n)$, and constraints $\texttt{win}_j(o_i)$ with limits $o_i w_j^m$, $o_i y_j^m$
5. Reusable Refurbishment tasks w_{ji} and b_{ji} and associated constraints, Reusable Resource usage $r_i(o_i)$, $r_i(w_{ij})$, $r_i(b_{ij})$, and Reusable Resource capacities r_i^m
6. Disjunctions of inequalities to handle Z

When checking infrastructure violations, e_n is constrained to occur at $t(e_n)$ in the manifest. All prior events are constrained to occur at the times recorded in X. When checking constraint violations, most of the problem described above involves evaluating the constraints given a fixed assignment. The only free variables are the start times and durations of the reusable refurbishment tasks w_{ji} and b_{ji}, not just for the current launch but others in X, which may be scheduled after $t(e_n)$. Even here, we only need to check whether reusable resource limits r_i^m are violated given the fixed start times of the launches in the run, and the time $t(e_n)$ from the manifest. All minimum spacing and KSC pairwise constraints devolve to simple inequalities, because every launch in X is totally ordered, and every launch in X precedes the current event e_n, both when checking constraints and when rescheduling.

For each event, the violations caused by resource limitations, blackouts, or the pairwise constraints in K must be recorded, along with the associated delay durations. If the reusable wait and refurbishment jobs are collectively infeasible, we only report that the support equipment constraint is violated, not whether there is an insufficient amount of support equipment or that the shop capacity is insufficient.

When rescheduling, we must find a new time for e_n that does not violate any infrastructure constraints; as described in the Overview, the problem is to minimize $t^*(e_n)$ subject to all the constraints. Instead of checking constraints, we pose and solve the problem of moving the current event to satisfy all constraints. The consumable resource constraints can be simplified into inequalities. Consider some consumable resource c_i. Because all prior events and scrubs are fixed in the simulated manifest, and we know how to compute the total replenished resource using the hourly rate $\delta(c_i)$, we know how to compute the available resource at $t(e_n)$, which is the current scheduled time for event e_n. Denote the amount of resource c_i that has been used and not replenished at $t(e_n)$ by $\texttt{amt}(c_i, t(n))$. If $c_i(e_n) + \texttt{amt}(c_i, t(n)) > c_i^m$, we can compute how far into the future e_n needs to be scheduled in order not to violate the resource constraint, namely, $w = \frac{c_i(e_n)+\texttt{amt}(c_n,t(e_n))-c_i^m}{\delta(c_i)}$. We then add constraint $t^*(e_n) \geq x_i = t(e_n)+w$.

Each rolling limit constraint $o_{i,j}$, characterized by window size $\texttt{win}_j(o_i)$ and limit $o_i w_j^m$, can be expressed as a cumulative constraint $Cum(S, D, R, b)$ (Caseau and Laburthe 1996; Beldiceanu and Carlsson 2002), which requires that a set of tasks given by start times S, durations D, and resource requirements R, never require more than a global resource bound b at any one time. The insight is that any event using o_i 'counts against' the constraint for $\texttt{win}_j(o_i)$ days, after which it is outside the window. Start times include the new event $t^*(e_n)$, and all prior event times in the run. The duration of resource usage equals $\texttt{win}_j(o_i)$ and resource usage for each event is indicated by $o_i(e_i)$. Recall that events in runs could be scrubs, and that we must protect for a scrub of the current event, so the resource usage for the current event e_n equals Finally, the global resource bound is $o_i w_j^m$ or $o_i w_j^{m'}$ depending on whether $co_i w_j \leq o_i y_j^m$. There is one such constraint per rolling window; each can be written $Cum(\mathbf{T}, t(e_n)^*; \mathbf{W}; \mathbf{O}; \mathbf{o_i w_j^m})$ where $\mathbf{T}$ is the set of all known event times e_i in the run, $\mathbf{W}$ is the set of all durations starting at those times, $\mathbf{O}$ is the set of all rolling window resource use of all events ($o_i(e_i)$ or $o_i(a_j)$ depending on the event). Bold font indicates constants; the only free variable in each such constraint is the new time for the current event.

We keep track of how many times $\texttt{win}_j(o_i)$ is reached (not exceeded) in a year using $co_i w_j$ and when this number equals $o_i y_j^m$, we reduce $o_i w_j^m$ to $o_j w_i^{m'}$, and use the same constraint of rolling limits above.

The reusable resource constraints on support equipment are expressed as a *pair* of cumulative constraints. The first constraint ensures all wait tasks w_{ji} and refurbishment tasks b_{ji} using the launch support equipment resource r_1 respect limit r_1^m. The cumulative constraint over tasks using r_1 (launch support equipment) is thus written $Cum(B_s, W_s; B_d, W_d; \mathbf{R_1}; \mathbf{r_1^m})$ where B_s, W_s are the start times of the wait and refurbishment jobs, B_d, W_d are the durations of those jobs, $\mathbf{R_1}$ is the set of reusable resource usages of r_1 of those jobs. The second constraint ensures the refurbishment tasks using the shop resource r_2 respect the shop limit r_2^m; this constraint is written $Cum(B_s; B_d; \mathbf{R_2}; \mathbf{r_2^m})$ where $\mathbf{R_2}$ is the set of reusable resource usages of r_2 of those jobs.

As final note, the minimum makespan schedule may push a launch outside the last date of the scheduling horizon. This does not constitute a constraint violation, but such 'failed'

launches are recorded as part of STAR's metrics.

The declarative version of the rescheduling problem follows. The declarative form of the constraint checking problem for violation recording is similar, except that $t(e_n)$ is fixed, and we minimize the latest end time of the schedule.

$$\min t(e_n)^*$$

$$\text{s.t. } \mathbf{x_i} \leq t(e_n)^* \qquad \forall c_i \in C \quad (1)$$

$$\vee_i \left(\mathbf{z_{i,l}} \leq t(e_n)^* \leq \mathbf{z_{i,u}}\right) \qquad \forall z_i \in Z \quad (2)$$

$$\mathbf{k_{hin}} \leq t(l_n)^* - \mathbf{t(l_i)} \qquad \forall k_h(l_i, l_n) \in K \quad (3)$$

$$Cum(\mathbf{T}, t(e_n)^*; \mathbf{W}; \mathbf{O}; \mathbf{o_i w_j^m}) \qquad \forall O_i, \forall o_{i,j} \in O_i \quad (4)$$

$$w_{ji,s} = \mathbf{t(e_i)} \qquad \forall W_i, \forall w_{ji} \in W_i \quad (5)$$

$$w_{jn,s} = t^*(e_n) \qquad \forall w_{jn} \in W_n \quad (6)$$

$$w_{ji,s} + w_{ji,d} = w_{ji,e} \qquad \forall W_i \cup W_n, \forall w_{ji} \in W_i \quad (7)$$

$$b_{ji,s} + b_{ji,d} = b_{ji,e} \qquad \forall B_i \cup B_n, \forall b_{ji} \in B_i \quad (8)$$

$$b_{ji,s} = w_{ji,e} \qquad \forall B_i \cup B_n, \forall b_{ji} \in B_i \quad (9)$$

$$b_{ji,d} = \mathbf{bd} \qquad \forall B_i \cup B_n, \forall b_{ji} \in B_i \quad (10)$$

$$Cum(B_s, W_s; B_d, W_d; \mathbf{R_1}; \mathbf{r_1^m}) \quad (11)$$

$$Cum(B_s; B_d; \mathbf{R_2}; \mathbf{r_2^m}) \quad (12)$$

Bold font indicates constants. Universal quantification indicates multiple constraints of each type. Constraints 1 represent consumable resource imposed delays. Constraints 2 are due to blackouts. Constraints 3 are due to pairwise infrastructure constraints. Constraints 4 are due to rolling resource window constraints. Constraints 5 - 6 constrain the start of the wait jobs for the reusable resources to launch times. Constraints 7 - 12 are the linked reusable resource constraints for the wait and refurbishment jobs.

6 Evaluating KSC Launch Operations

In this section, we describe the scale of KSC's launch operations, and evaluate STAR performance when used to analyze potential future KSC operations. KSC supports 20 distinct types of launch vehicles. There are 400 individual constraints, divided into 6 classes, between pairs of launch vehicles. Launches are separated by durations ranging from a few hours to a few days, and most are asymmetric (i.e. depend on the launch order). There are also 25 customer-imposed minimum spacing constraints. KSC-wide resources include one consumable, one rolling resource with three rolling window limits and no yearly limit, one rolling resource with four rolling windows and one yearly limit, and the two inter-linked reusable resources constraining pad support equipment and the shop. Analyses typically contain multiple infrastructure-prescribed blackout windows and customer launch windows. A typical manifest could contain 100 launches per year; a large manifest could contain $175-200$ launches. Analyses can run multiple years. Weather and technical scrub probabilities depend on each launch vehicle type (newer vehicles scrub more frequently in a year) and time of year (poor weather causes more scrubs).

STAR uses a combination of MiniZinc constraint modeling (Nethercote et al. 2007) and CBC constraint solver (Forrest and Lougee-Heimer 2014)[3]) and direct implementation of some constraint checking during manifest simulation in Python. MiniZinc features a simple, powerful modeling language, and the ability to pose and solve many problems easily via a Python API. As noted previously, STAR must assess the actual KSC infrastructure constraint violations resulting from unexpected events that lead to rescheduling; these constraints naturally map to constraints we pose using MiniZinc, as described in Section 5. STAR incorporates a manifest generation capability that takes as inputs the number of launches of specific types, manifest horizon [4]. Finally, STAR allows specification of probabilities and impacts of unexpected events. STAR was evaluated using a HP EliteBook 640 14 inch G9 Notebook PC, with an I5 1600 MHz processor and 16GB of RAM.

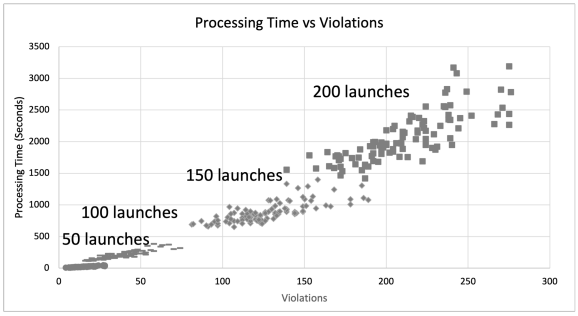
Figure 4a shows STAR run-times in a scatterplot, with the y axis indicating run-time in seconds for each manifest, and the x axis measuring the number of violations. Data from 100 manifests with $50, 100, 150$ and 200 launches each are shown. For each manifest, we see that run-time is (roughly) linear in the number of violations. Since only the current launch in the manifest can move, the number of free variables in our MiniZinc model is limited to those pertaining to that launch, plus those support equipment jobs from recent launches, and is thus weakly dependent on the number of launches in a manifest. Figure 4b is a box-and-whisker plot of the same data in Figure 4a, and shows super-linear growth in mean and median runtime with increasing launches; however, growth is almost linear after 100 launches. Since launch support equipment jobs can be left pending by launches that occur within a few days of the current launch, denser manifests with more launches will have more launches using that equipment, and thus lead to increased rescheduling time. Since there is a limited pool of such equipment, it is expected that this effect will diminish after enough launches are present to ensure the maximum number of pieces of equipment are always in use.

Recall that STAR tracks direct delays due to infrastructure violations that cause rescheduling separately from ripple effect delays, which are propagated from the direct delays to the customer constraints on launches due to pairwise constraints. Figure 4c is a box-and-whisker plot of the direct delays due to infrastructure violations for 100 manifests with $50, 100, 150$ and 200 launches. As with processing time, we see super-linear growth in the number of such direct delays, but growth is almost linear above 100 launches. Figure 4d is a box-and-whisker plot of the ripple effect delays due to infrastructure violations for 100 manifests with $50, 100, 150$ and 200 launches. We see a very sharp increase in ripple effect delays at 200 launches, indicating that for at least some sets of launches, there is a threshold beyond which essentially any launch delay from KSC translates to significant delays due exclusively to customer constraints.

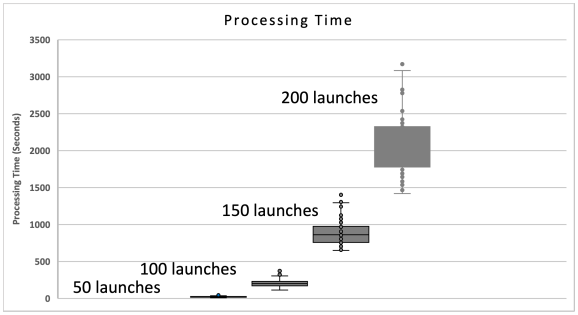
The scheduling problem solved by STAR contain multiple cumulative constraints linked by precedences, release times and deadlines, and is therefore most likely an instance of the Resource Constrained Project Scheduling Problem (RCPSP)

[3]https://pubsonline.informs.org/doi/pdf/10.1287/educ.1053.0020

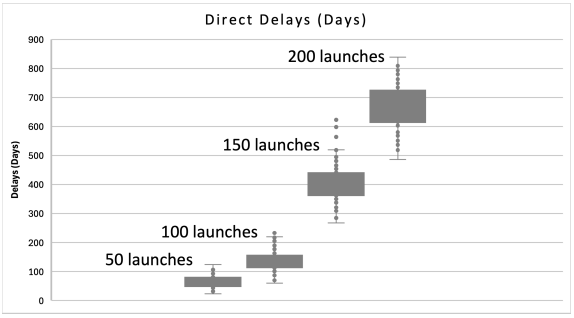
[4]Manifest generation employs custom algorithms, and its description is beyond the scope of this paper.



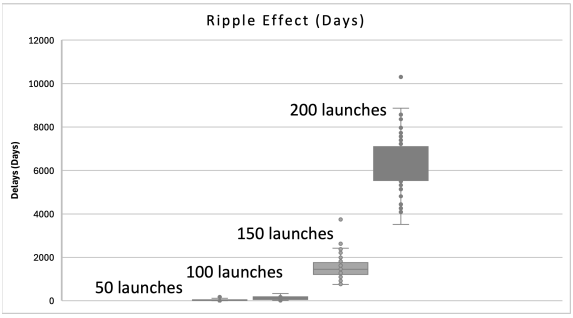
(a) STAR Runtime vs violations (of all classes) scatterplot.



(b) STAR Runtime statistics vs number of launches.



(c) Direct Delay statistics vs number of launches.



(d) Ripple Effect Delay statistics vs number of launches.

Figure 4: STAR runtime and performance statistics for 100 manifests with $50, 100, 150, 200$ launches.

which is $\mathcal{NP}$-Hard (Blazewicz, Lenstra, and Kan 1983)). However, Figure 4b appears to grow only polynomially with higher numbers of launches for our sample problems.

7 Related Work

The problem faced by KSC is similar to the problem of identifying *bottlenecks* that prevent scheduling and impact customer satisfaction. A typical approach to bottleneck analysis is to use Discrete Event Simulations (DES) (Lai, Che, and Kashef 2021). A challenge of DESs is that event transitions may not capture scheduling decisions without significant modeling effort. Direct analysis of optimal schedules (Wang et al. 2016) is also used. By contrast, the approach taken in STAR is to record constraint violations found during the simulated execution of the schedule, and to explicitly reschedule when violations are found. (Zhu, Zhou, and Che 2022) describes integrating scheduling and simulation to handle scheduling problems in the presence of uncertainty. Such approaches do not perform bottleneck analysis, which is the problem STAR addresses. Similarly, controllability approaches (Combi et al. 2019; Micheli 2017) explicitly handle temporal constraints and uncertainty, but also don't identify the sources of bottlenecks. Some approaches both identify and relax constraints to solve complex planning and scheduling problems. Hauser (Hauser 2014) describes the minimum constraint removal problem to enable robots to operate safely in an environment. The controllability work of (Yu, Fang, and Williams 2015) investigates how to relax constraints to ensure a controllability problem (probabilistic temporal constraints) can be controlled. They use relaxation costs to solve optimization problems. Eifler et al. (Eifler, Frank, and Hoffmann 2022) show how to explain why plans cannot simultaneously achieve pairs of goals, or properties. They automatically determine minimal relaxations of time and resource constraints that allow both (or sets of) properties to be achieved. While these approach finds relaxations of constraints to solve over-constrained problems, they may be too limited to address the problem STAR needs solved.

We chose to use a complete solver (CBC) to reschedule after constraint violations. Alternative approaches, e.g. those employing learning (Shyalika, Silva, and Karunananda 2020) or metaheuristcs such as local search (Beck, Feng, and Watson 2011) or large neighborhood search (Pisinger and Ropke 2019) could be used instead, providing they generate valid schedules with no constraint violations.

8 Conclusions and Future Work

STAR is a novel application, interleaving random generation of schedule disrupting events and rescheduling to evaluate bottlenecks in a complex facility. It also offers an interesting use case for knowledge engineering for scheduling applications. We have described these challenges for a specific application, our design and approach and a standard MiniZinc models and solver technology. Our lessons learned are useful for future projects requiring the integration of simulation, scheduling, and tracking resource violations as opportunities for investing in new infrastructure. We expect the same general methodology can be applied to other bottleneck analysis problems, e.g. at airports, factories, or port facilities. We expect that solver time can be further reduced by removing variables and constraints known to be fixed in the run.

Acknowledgements

We acknowledge the contributions of Kyle Booth, who designed and developed the first version of STAR, and our interns Sabrina Yazarda Noor and Ian Naidel, who helped design the UI and early versions of the manifest generator.

References

Beck, J. C.; Feng, T.; and Watson, J.-P. 2011. Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing*, 23(1): 1–14.

Beldiceanu, N.; and Carlsson, M. 2002. A New Multi-Resource *cumulatives* Constraint with Negative Heights. *Proceedings of the 8th International Conference on the Principles and Practices of Constraint Programming*.

Blazewicz, J.; Lenstra, J.; and Kan, A. 1983. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1): 11–24.

Caseau, Y.; and Laburthe, F. 1996. Cumulative Scheduling with Task Intervals. 363–377.

Combi, C.; Posenato, R.; Viganó, L.; and Zavatteri, M. 2019. Conditional Simple Temporal Networks with Uncertainty and Resources. *JAIR*, 64: 931 – 985.

Eifler, R.; Frank, J.; and Hoffmann, J. 2022. Explaining Soft-Goal Conflicts through Constraint Relaxations. In *Proceedings of the* 31^{st} *International Joint Conference on Artificial Intelligence*, 4621 – 4627.

Forrest, J.; and Lougee-Heimer, R. 2014. CBC User Guide. INFORMS TutORials in Operations Research.

Hauser, K. 2014. The minimum constraint removal problem with three robotics applications. *The International Journal of Robotics Research*, 33(1): 5–17.

Lai, J.; Che, L.; and Kashef, R. 2021. Bottleneck Analysis in JFK Using Discrete Event Simulation: An Airport Queuing Model. In *2021 IEEE International Smart Cities Conference (ISC2)*, 1–7.

Micheli, A. 2017. Disjunctive temporal networks with uncertainty via SMT: Recent results and directions. *Intelligenza Artificiale*, 11(2): 155–178.

Nethercote, N.; Stuckey, P.; Becket, R.; Brand, S.; Duck, G.; and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the* 13^{th} *International Conference on Principles and Practice of Constraint Programming*, 529—543.

Pisinger, D.; and Ropke, S. 2019. *Large Neighborhood Search*, 99–127. Cham: Springer International Publishing.

Shyalika, C.; Silva, T.; and Karunananda, A. 2020. Reinforcement Learning in Dynamic Task Scheduling: A Review. *SN Computer Science*, 1: 306.

Wang, J.-Q.; Chen, J.; Zhang, Y.; and Huang, G. Q. 2016. Schedule-based execution bottleneck identification in a job shop. *Computers and Industrial Engineering*, 98: 308–322.

Yu, P.; Fang, C.; and Williams, B. 2015. Resolving Over-constrained Probabilistic Temporal Problems through Chance Constraint Relaxation. In *Proceedings of the* 29^{th} *National Conference on Artificial Intelligence*, 3425 – 3431.

Zhu, M.; Zhou, C.; and Che, A. 2022. Simulation-Optimization Approach for Integrated Scheduling at Wharf Apron in Container Terminals. In *2022 Winter Simulation Conference (WSC)*, 1944–1955.