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

Richard Levinson[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
Authors listed in alphabetical order.

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 results of STAR are constraint violations leading to delays that can be used to make investments to reduce future delays. We describe the modeling, rescheduling problem formulation and algorithm development, and testing of STAR.

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 works to ensure an environment in which NASA's programs and other users can safely and effectively carry out their operations. Approximately 100 launches are expected to take place in 2024 [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. Since space vehicles are complex and launch operations are uncertain, unexpected events can also cause delays in operations.

The Spaceport Throughput Analysis Resource (STAR) assesses whether a set proposed 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 using a monte-carlo approach driven by configurable probabilities of different classes of event outcomes, including delays and worst-case use of resources. The results of STAR are the constraint violations leading to delays, inform stakeholders of key constraints preventing customers from being able to perform their missions as desired, and give insight into how to improve spaceport throughput. STAR has been deployed at KSC.

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 describe the specific problem solved for KSC. In Section 7 we describe the challenges of knowledge engineering for STAR.

2 Overview

We begin with describing KSC spaceport operations, and the problem STAR is intended to address. A *manifest*, or launch manifest, consists of a set of *launches* of different 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 launches to notional launch dates and times. A supporting event always precedes the launch event by some 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.

Finding a schedule to resolve constraint violations resembles a Job-Shop Scheduling Problem (Brücker 1998), a well studied problem in computer science. However, resource use and constraints are quite complex, so STAR does not fit nicely into a single job-shop paradigm. STAR's job is to *identify KSC infrastructure constraints* that lead to delays during the execution of manifests. 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. We distinguish in STAR 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 dates in the initial manifest not being met, translate to lack of customer satisfaction. Constraints are divided

[1]https://www.nasa.gov/centers-and-facilities/kennedy/kennedy-space-center-looks-ahead-busy-2024/

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 when they are the primary cause of a launch delay. The key question STAR answers is which KSC-wide constraints cause rescheduling, and how often.

2.1 Resources

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*; launches use these resources, and they are replenished at a fixed daily rate. For instance, KSC may have storage for Nitrogen that can be fully replenished in 5 days, with individual events using between 20% and 60% of the storage capacity.

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 of Helium 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. For instance, 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 class of rolling limit constraint describes the maximum number of times a rolling limit can be reached in a year. An example of this second class of constraint is that 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 will refer to these as Yearly Rolling Limits

Consider two hypothetical launch vehicles. We assume events use individual pieces of equipment to support launches, stored Nitrogen, and truckloads of Helium with a single rolling limit. One of our launch vehicles 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 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, and the supporting event occurs 3 days before the launch. The supporting event uses $\frac{3}{5}$ of the Nitrogen tank and 2 trucks worth Helium.

[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/

2.2 Scrubs

Due to the uncertainties involved in spaceport operations, launches or supporting events may be delayed or *scrubbed* for numerous reasons. Both launches and supporting events can be scrubbed. A scrub may use resources; scrubbed launch and supporting events use the same amount of resources (if any). Scrub resource use is usually higher than that of either a supporting event or a launch. To continue with our example, 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). Resource constraints prior to the actual launch are checked assuming the worst-case resource usage, i.e. STAR assumes that the event in the manifest scrubs and uses resources. That means in our previous example, checking Nitrogen consumption for a GU-V launch would assume $\frac{3}{5}$ of the Nitrogen tank and 2 trucks worth of Helium. Once the event has been deemed successful, however, the resource consumption of the launch is used instead. The tracking of the scrubs in the manifest is shown for the consumable Helium resource, and protecting against scrubs in the manifest, is shown in Figures 1; the dotted line shows how much more resource would be used if a scrub that uses resources occurs instead. When a scrub using resource occurs, the event is attempted again until it succeeds. The run records the usage of the scrub.

2.3 Resource Examples

To see how rolling limits work, and also see how the characteristics of each vehicle bump up against KSC-wide infrastructure constraints, consider the Go-Upper Five's Helium consumption. The 6 tankers in 7 days rolling Helium limit described in the previous paragraph ensures KSC can only launch one GU-V or a GU-I every rolling 7 day period, as shown in Figure 1 (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 of the 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 Helium resource work, is shown in Figure 1 (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.

In addition to constraints on individual launches, *pairs* of launch events must respect a wide variety of temporal constraints. Pairwise constraints propagate from launch events

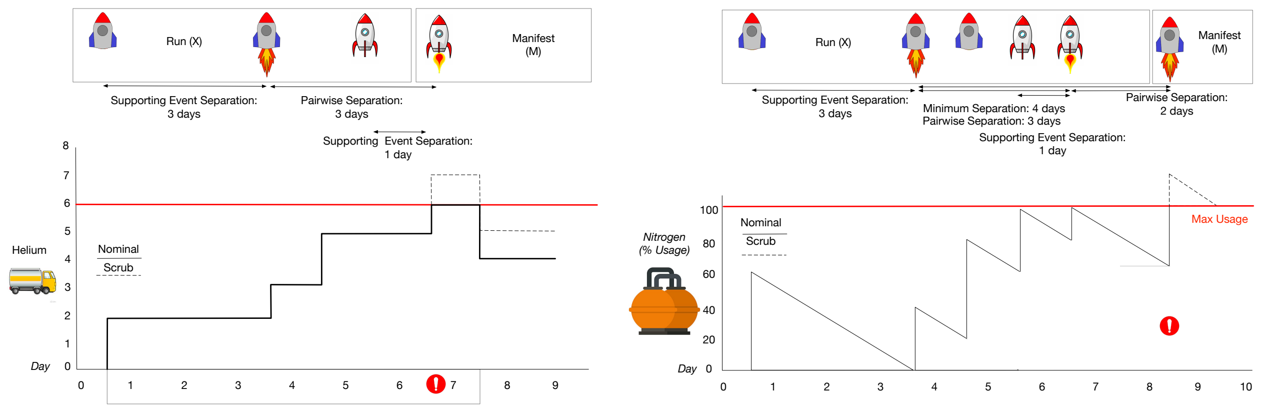

Figure 1: Tracking Rolling limits (left): Two launches are shown, a GU-V and a GU-I. The GU-V supporting event uses 2 tankers while the launch uses 1 tanker. The rolling limit is 6 tankers in a 7 day period; the first period is indicated at the bottom of the figure. The separation between main and supporting events of the GU-V is 3 days. The run shows a successful GU-V launch on Day 3. The GU-I supporting event on Day 5 is followed by its launch on Day 6. We must conservatively assume the GU-I launch uses 2 trucks if there is a scrub; the accumulated use of Helium violates the rolling limit on Day 6. Tracking Consumable Resources (right): This manifest has three launches, one GU-I and two GU-Vs. The second GU-V launch is on day 8. Nitrogen is a consumable resource used by all launches, and the tank can be refilled in 5 days. The worst-case Nitrogen resource use of the GU-V launch is on day 8 in the manifest violates the limit.

to the supporting events, so the only pairwise constraints we need to describe are those between launches. Many constraints arise due to the need to manage infrastructure across all of KSC during any launch, and are referred to as *KSC pairwise constraints* Customer's constraints and processes drive the time needed to reconfigure a launch pad between launches. For instance, it may take 4 days between successive GU-V launches and 3 days between successive GU-I launches. We refer to these as *minimum spacing* constraints. .

Some of these 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, whereas if the GU-I is scheduled to launch before the GU-V, the separation is only 2 days. Examples of both minimum spacing and KSC pairwise constraints are also shown in Figure 1 such as the 3 day separation constraint between the first GU-V launch on day 3, and the GU-I launch on day 6, and the violated 2 day separation constraint between the GU-I launch and the second GU-V 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, over 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 $t(s_i)$. It is sometimes convenient to remain neutral about whether the event is a launch or supporting event, in which case we denote the event by e_i and the time of the event by $t(e_i)$.

We denote the set of rolling limit constrained resources O, the set of consumable resources C, and the set of reusable resources R. Each event uses some amount of the specified resources. All resources are consumed at the hour on the date the event occurs. Consumable resources are replenished at a fixed rate. Rolling and reusable resources availability is discussed further below.

We denote the amount of resource used by event e_i as follows: $o_i(e_i)$ denotes use of rolling limit resource o_i, $c_i(e_i)$ denotes use of consumable limit c_i, and $r_i(e_i)$ denotes the use of reusable resource r_i. 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 are expressed in terms of the maximum rate of replenishment in a rolling daily period. Recall there can be multiple rolling limits on each such resource o_i; each such limit is denoted $o_{i,j}$. We denote by $\texttt{window}_j(o_i)$ 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 $o_i y_j^m$ as 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 fixed period of time and

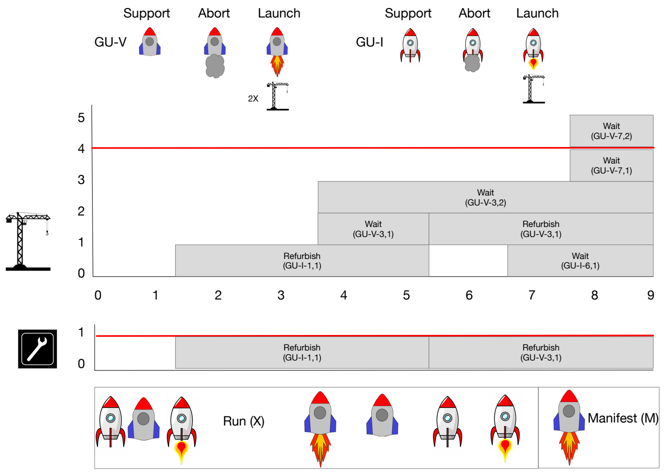

Figure 2: Tracking Reusable Resources. The run shows a GU-I launch on day 1, a GU-V launch on day 3, both of which are recorded in the run X, and a GU-V launch on day 6 in the manifest. Each GU-I uses one piece of launch support equipment, while the GU-V uses two. Since the shop can only refurbish one 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.

then returned. We denote by r_i^m the maximum capacity of reusable resource r_i. The reusable resources in STAR are complicated due to the need to refurbish each piece of equipment between launches; The shop needed to refurbish these reusable resources are also constrained, and can refurbish a limited number of pieces of support equipment at a time.

Modeling the usage and refurbishment is accomplished using an additional set of tasks. We refer to 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 b_{ji} tasks is denoted B_i.

The duration of b_{ji} is fixed to some duration bd. The duration of the wait tasks 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. The set of temporal constraints on wait times and refurbishment tasks associated with l_i is denoted BT_i. A run and manifest tracking the Reusable resources is shown in Figure 2.

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 somtimes depends on which vehicle launches first, so in general such constraints take the form

$$k_h(l_i, l_j) = \begin{cases} t(l_j) > t(l_i) \Rightarrow t(l_j) - t(l_i) \geq k_{hji} \\ t(l_j) < t(l_i) \Rightarrow t(l_i) - t(l_j) \geq k_{hij} \end{cases}$$

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 by X the simulated manifest events, the times of those events, and scrubs, scrub times, 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

We are initially presented 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 manifest. STAR simulates the manifest, generates unexpected events, and records delays and the reasons for them in the run.

STAR processes events in M in chronological order imposed by the manifest, but is modified by delays and rescheduling. As described in the introduction, STAR simulates events to determine whether launches are scrubbed, or occur at $t(e_i)$. In STAR, 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.

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 run are indexed by i, thus, e_i; we do not need to refer events in the manifest other than e_n.

At the time at which e_n is simulated, we must assume there will be 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. In general, $o_j(a_n)$ $o_j(s_n)$ $c_j(s_n)$ are all distinct. When an event succeeds and is inserted into run X, we store the nominal resource usage in X. When referring to resource use in X, we use an index other than n; $o_j(a_i)$ and $c_j(a_i)$ refer to scrubbed event resource use, and $o_j(l_i)$ and $c_j(l_i)$ 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 $t(l_n) - \mathtt{window}_j(o_i)$ and $t(e_n)$. Denote by $\mathtt{amt}(o_j, t(i))$ 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 $\mathtt{amt}(o_j, t(i))$= 0, otherwise we compute the amount as usual.

Consumable resource replenishment is assumed to occurs 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 $\mathtt{amt}(c_i, t(i))$. 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 $\mathtt{amt}(c_i, t(n))$= $\max((\mathtt{amt}(c_i, t(i)) + \delta(c_i)(t(e_n) - t(e_i)), c_i^m)$.

If executing the event e_n at $t(e_n)$ would violate some infrastructure constraint, it is rescheduled. However, only events in X and e_n itself are used to check for constraint violations. Only e_n can be rescheduled, and it is always moved 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. The new launch date and time are constrained to move integer days into the future, not hours. Because of orbital mechanics, when there is a scrub, the launch time the next day is 30 minutes earlier and we approximate as a 24 hour scrub.

Some events in the manifest are never be rescheduled as a result of infrastructure violations. These events impose a special form of blackout before and after the event. The set of such events are denoted M_U. For an event $e_i \in M_U$, at time $t(e_i)$, an associated blackout $uz_i \in Z$ ensures no other event can be scheduled in the blackout period. In general, $t(e_i) \in [uz_{i,l}, uz_{i,u}]$. Unlike other blackouts, if the associated event moves because of weather or technical delays, then $t(e_i)$ and the associated uz_i also moves.

Algorithm 1: SimulateManifest

```
Input  : Manifest $M$
Input  : Blackouts $Z$
Input  : Pairwise $K$
Input  : Rolling $O$
Input  : Consumables $C$
Input  : Reusable $R$
Input  : Unmovables $M_U$
Input  : Horizon $H$
Output: Run $X$
$X \leftarrow \varnothing$;
while $M \neq \varnothing$ do
    $e_n \leftarrow$ getNextEvent$(M, e_n)$;
    $b \leftarrow$ checkInfrastructureSatisfied$(X, e_n)$;
    if $(b == false)$ then
        recordViolations$(X, e_n)$;
        $t(e_n) \leftarrow$ rescheduleEvent$(X, e_n)$;
        propagateSpacing$(X, e_n, M)$;
        recordDelays$(e_n, M)$;
    else                                   // $(b == true)$
        UpdateYearlyRollingLimitCount$(X, e_n)$;
        $u_n \leftarrow$ getEventOutcome$(e_n)$;
        if $(u_n == success)$ then
            getEventUsage$(e_n)$;
            $X \leftarrow X \cup (t(e_n), e_n)$;
            $M \leftarrow M - (e_n)$;
        else                               // $(u_n \neq$ success$)$
            if $(u_n == reseuse)$ then
                getEventUsage$(a_n)$;
                $X \leftarrow X \cup (a_n, t(e_n))$;
            $t(e_n) \leftarrow t(e_n)+$ getScrubDelay$(a_n)$;
            propagateSpacing$(X, e_n, M, Z)$;
return violations;
```

When events are rescheduled or delayed, a new event time $t^*(e_n)$ is generated, and the event is inserted back into manifest, in chronological order. This means events can be processed in a very different order than they appeared in the initial manifest M. 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.

To ensure that the manifest always respects 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 hilight 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 and time of the current event. Blackouts, minimum spacing and pairwise constraints, the resource types and limits, and the horizon, are all used by most of the functions. As noted, delays in unmovable events can lead to changes in the blackouts due to these events, so the set of blackouts is passed to propagateSpacing(X,e_n,M,Z).

STAR logs event times, scrubs and scrubs, delays and failed launches. 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

For each launch, the violations caused by resource limitations, blackouts, or the pairwise constraints in K must be recorded, along with their durations. In the event that the reusable wait and refurbishment jobs are collectively infeasible, we only need to report that the support equipment constraint is violated, not whether there is insufficient amount of support equipment or that the shop capacity is insufficient. The job-shop-like 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 $\texttt{window}_j(o_i)$ with limits $o_i w_j^m$, $o_i y_j^m$
5. Disjunctions of inequalities to handle Z
6. Reusable Refurbishment tasks w_{ji} and b_{ji}
7. Reusable Refurbishment task constraints BT_i, BT_n
8. Reusable Resource usage $r_i(o_i)$, $r_i(w_{ij})$, $r_i(b_{ij})$
9. Reusable Resource capacities r_i^m

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. This means that, when checking constraint violations, almost all of the job-shop-like problem described above does not require solving a constraints problem at all, but merely 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 extend past $t(e_n)$. Even here, we only need to check whether the r_i^m limits 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.

When rescheduling, we find a new time for e_n that does not violate any infrastructure constraints; the problem is to minimize $t^*(e_n)$ subject to all the constraints. Instead of checking constraints, we now pose and solve the problem of moving the current event to satisfy all constraints. We now proceed to show how many of the resource constraints are simplified into inequalities, and formalize the remaining constraints into a mathematical programming problem.

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 $\texttt{window}_j(o_i)$ and limit $o_i w_j^m$, can be expressed as a cumulative constraint (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. Start times include the new event $t^*(e_n)$, and all prior event times in the manifest. The duration of resource usage equals $\texttt{window}_j(o_i)$ (abbreviated $w_j(o_i)$ for brevity below), and resource usage for each event is indicated by $o_i(e_i)$. Recall that events in runs could be scrubs. Recall we must protect for a scrub of the launch so the resource usage for the current event e_n is designated $o_i(a_n)$. Finally, the global resource bound is $o_i w_j^m$ or $o_i w_j^{m'}$ depending on whether $co_i w_j \leq o_i y_j^m$. Each such constraint, then, can be written $Cum(\mathbf{t(e_i)}, t^*(l_n), \mathbf{w_j(o_i)}, \mathbf{w_j(o_n)}, \mathbf{o(e_i)}, \mathbf{o(a_n)}, \mathbf{o_i w_j^m})$, where bold font indicates a constant. All events in the run, plus the current event in the manifest, are in the scope of this constraint; $t(e_i)$ refers to all event times, $\mathbf{w_j(o_i)}$ refers to all window durations, and so on. The only free variable in each such constraint is the new time for the current event.

We keep track of how many times $\texttt{window}_j(o_i)$ is reached (not exceeded) in a year, 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 are also 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. This cumulative is thus written $Cum(b_{ji,s}, b_{jn,s}, w_{ji,s}, w_{jn,s}, b_{ji,d}, b_{jn,d}, w_{ji,d}, w_{jn,d},$

$\mathbf{r_1(b_{ji}), r_1(b_{jn}), r_1(w_{ji}), r_1(w_{jn}), r_1^m)}$. The second constraint ensures the refurbishment tasks using the shop resource r_2 respect the shop limit r_2^m; this constraint is written $Cum(b_{ji,s}, b_{jn,s}, b_{ji,d}, b_{jn,d}, \mathbf{r_2(b_{ji}, r_2(b_{jn}), r_2^m})$ As above, all events in the run, plus the current event in the manifest, are in the scope of these constraints; $b_{ji,d}, w_{ji,d}, \mathbf{r_1(b_{ji})}$ and so on refer to all wait and refurbish jobs of all events in the run.

Figure 3 shows the declarative version of the rescheduling problem. 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.

6 STAR for KSC

STAR is configured to evaluate manifests using 20 distinct launch vehicles. There are 400 pairwise constraints, divided into 6 classes of constraint. 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 25 customer-imposed minimum spacing constraints.

While the specific details of launch vehicles STAR is used to analyze can't be divulged due to commercial sensitivity of launch providers and U.S. Government considerations, we can give a sense of the scalability needed to enable STAR analyses. There are 20 distinct types of launch vehicles. There are 6 classes of KSC-wide pairwise vehicle separation constraints; however, not every pair of vehicles is constrained by every class of constraint. There are 4 KSC-wide resources; one consumable, one rolling resources 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 (pad support equipment and the shop). There are multiple infrastructure-prescribed blackout windows; customer launch windows can be added. A typical manifest could contain 100 launches per year; a large manifest could contain $175 - 200$ launches. Analyses can run multiple years. Scrub and 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).

7 How to Build a STAR

Knowledge engineering for STAR consisted of several parts. First, what are the key constraints on KSC launch operations, and how do we formally represent them? Second, what form of analysis is STAR required to do, what algorithms did STAR require to perform that analysis, and how were they implemented? Third, how did we ensure STAR was implemented properly and performing its required functions? We discuss each of these elements of STAR in turn.

The domain knowledge required to build STAR was gathered by interviewing domain experts at KSC familiar with each of the main classes of constraints. These constraints were first documented using natural language. Each constraint, in turn, needed to be represented in a formal way. As we describe in Section 3, the bulk of STAR's constraints are straightforward temporal inequalities, but the specifics of the resource constraints required significant iteration with the KSC customer before settling on a formal definition. The specific analysis required (discussed in the next paragraph) also resulted in some simplification of the resource constraints. Ultimately, as we describe in Section 5, we mapped every constraint described by the KSC customer (both KSC infrastructure and customer constraints) to a constraint in minizinc's constraint language, and use a minizinc-compliant solver (CBC, in our case), to solve the resulting scheduling problems. The most complex constraint model elements were the inter-related reusable resource constraints modeling launch support equipment.

As we describe in Section 4, analysis performed by STAR required simulating each launch, random events that could delay the launch, and reporting constraint violations. Documenting the specific process prior to design and implementation revealed numerous problems to be solved, but also resulted in simplifications of what would otherwise be a very hard problem. STAR uses a combination of constraint modeling (minizinc (Nethercote et al. 2007) and CBC constraint solver (Forrest and Lougee-Heimer 2014)) and direct implementation of constraint checking during manifest simulation in Python. We need to record what constraints are violated for a fixed assignment. That means commodity constraints technology isn't suited to our needs, because most solvers don't report violated constraints, which is what STAR needs to do. Since constraint violations are computed from a run plus the time of the next event in the manifest, most of these constraint violations are determined using Python code directly. The most complex constraints needing to be solved when rescheduling are those involving refurbishment of the pad support equipment, which are written to a minizinc model to check for constraint violations.

When rescheduling, however, we must write a somewhat more complex minizinc model involving all the derived inequalities from the run and the resource constraints, then find the earliest time we can reschedule the event in the manifest that has violated some constraints. Python code is used to compute the inequalities, as described in Section 4, which augment the minizinc model for the reusable resources, along with the rest of the constraints, and a minimization objective instead of a feasibility objective.

The need to split constraints between KSC infrastructure constraints and customer pairwise constraints deep inside the manifest simulation process led to custom code to write and solve separate minizinc models, one for rescheduling, and a second model (not described in this paper) to propagate customer spacing constraints after rescheduling was completed. However, STAR's requirements didn't include building an up-front schedule in the presence of uncertainty, allowing us to use a 'classical' constraint representation for rescheduling. Furthermore, the requirement to only reschedule a single launch at a time led to great simplification of the minizinc representation; in particular, the consumable resources devolve to simple inequalities.

The testing of the STAR constraint model posed a significant challenge. We constructed multiple test cases for each class of constraint present in STAR; that is, each of the 6 classes of pairwise constraint, customer spacing constraints, the consumable resource, the reusable resource, the 5 lim-

$$
\begin{aligned}
&\min t(l_n)^* \\
&\text{s.t. } \mathbf{x_i} \leq t(l_n)^* && \forall c_i \in C && (1)\\
&\quad \vee_i \left(\mathbf{z_{i,l}} \leq t(l_n)^* \leq \mathbf{z_{i,u}}\right) && \forall z_i \in Z && (2)\\
&\quad \mathbf{k_{hcj}} \leq t(l_n)^* - \mathbf{t}(\mathbf{l_j}) && \forall k_h(l_j, l_n) \in K && (3)\\
&\quad Cum(\mathbf{t}(\mathbf{e_i}), t^*(l_n), \mathbf{w_j}(\mathbf{o_i}), \mathbf{w_j}(\mathbf{o_n}), \mathbf{o}(\mathbf{e_i}), \mathbf{o}(\mathbf{a_n})) && \forall o_{i,j} \in O && (4)\\
&\quad w_{ji,s} = \mathbf{t}(\mathbf{l_i}) && \forall W_i, \forall w_{ji} \in W_i && (5)\\
&\quad w_{jn,s} = t^*(l_n) && \forall w_{jn} \in W_n && (6)\\
&\quad w_{ji,s} + w_{ji,d} = w_{ji,e} && \forall W_i \cup W_n, \forall w_{ji} \in W_i && (7)\\
&\quad b_{ji,s} + b_{ji,d} = b_{ji,e} && \forall B_i \cup B_n, \forall b_{ji} \in B_i && (8)\\
&\quad b_{ji,s} = w_{ji,e} && \forall B_i \cup B_n, \forall b_{ji} \in B_i && (9)\\
&\quad b_{ji,d} = \mathbf{bd} && \forall B_i \cup B_n, \forall b_{ji} \in B_i && (10)\\
&\quad Cum(b_{ji,s}, b_{jn,s}, b_{ji,d}, b_{jn,d}, \mathbf{r_2}(\mathbf{b_{ji}}, \mathbf{r_2}(\mathbf{b_{jn}}), \mathbf{r_2^m}) && && (11)\\
&\quad Cum(b_{ji,s}, b_{jn,s}, w_{ji,s}, w_{jn,s}, \\
&\qquad b_{ji,d}, b_{jn,d}, w_{ji,d}, w_{jn,d}, \\
&\qquad \mathbf{r_1}(\mathbf{b_{ji}}), \mathbf{r_1}(\mathbf{b_{jn}}), \mathbf{r_1}(\mathbf{w_{ji}}), \mathbf{r_1}(\mathbf{w_{jn}}), \mathbf{r_1^m}) && && (12)
\end{aligned}
$$

Figure 3: The constraint program for the rescheduling problem. 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.

its on rolling resources, the 3 limits on rolling window resources, along with a variety of other test cases for launch windows, blackout periods, and other related behavior. Test cases included scenarios in which a constraint should, and should not, be violated. Dozens of tests were constructed to validate STAR's ability to detect and repair every constraint correctly. Many tests could be constructed with either one or two launches. Blackouts can be tested with a single launch; pairwise constraints with two launches. Testing resource constraints often required as many as six launches.

Several key features of STAR facilitated testing. First, the ability to specify launch date and launch hour let us create tests guaranteed to satisfy or violate constraints. Second, the ability to turn off probabilistic events ensured we could construct tests with deterministic outcomes. Third, ensuring that unexpected failed events that consume resources led to the correct behavior was accomplished by implementing flags that would automatically fail the event in question. It was often not possible to construct test cases to only violate a single constraint; when multiple constraints were violated, careful documentation of all expected outcomes was necessary. STAR problems identified with such tests ranged from bugs in code to mis-configured constraints.

8 Related Work

The problem faced by KSC is to identify *bottlenecks* that prevent scheduling, and thus customer satisfaction. A typical approach is to use Discrete Event Simulations (DES) (Lai, Che, and Kashef 2021). A challenge of DES 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.

(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.

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. 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. Whether such approaches are scalable unclear, and none of the methods above incorporate uncertainty.

9 Conclusions and Future Work

STAR presents an interesting use case for knowledge engineering for scheduling. While we have described these challenges for a specific application, our design and approach employ 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.

References

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*.

Brücker, P. 1998. *Scheduling Algorithms*. Springer.

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. Https://pubsonline.informs.org/doi/pdf/10.1287/educ.1053.0020.

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.

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.

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.

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.