Utilizing Schedule Constraints to Improve Automated Scheduling in NASA's Deep Space Network

Evan Davis, Nihal Dhamani, Martina Troesch, Mark D. Johnston
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Dr., Pasadena CA 91109
evan.w.davis, nihal.n.dhamani, mark.d.johnston @ jpl.nasa.gov

Abstract

NASA's Deep Space Network (DSN) is a mission critical facility that supports many different space missions, from near earth to deep space exploration. In recent years, as the network has suffered from increasing oversubscription, more restrictions have been added to reduce the amount of manual scheduling labor necessary to come to consensus. In this paper, we describe a new automated scheduling tool to be used by schedulers to further reduce the amount of manual scheduling labor. This new scheduler takes advantage of these new restrictions to bring the scheduling problem closer to feasibility. In addition, we describe an algorithm for constraint relaxation given a partially-solved problem, thereby bringing the problem even closer to feasibility.

Introduction

NASA's Deep Space Network (DSN) is a primary resource for communications and navigation for a wide range of space missions (Imbriale 2003). As the number of active missions rises (from around 35 users to over 60), and bandwidth requirements increase with larger data sets, oversubscription is a constant battle. In the current scheduling process, a human named the Builder of Proposal (BOP) must first edit the schedule in order to remove the majority of "conflicts" (places where the proposed tracks would violate resource availability or other hard scheduling rules). Then, the schedule is further edited by users over a span of several days of negotiation with proposals and counter-proposals until consensus is reached, at which point many requirements might have "violations" (where stated constraints are unsatisfied), or have been dropped from the schedule entirely.

In order to reduce the workload for the BOP, several changes are being introduced to limit the amount of oversubscription in the schedule. First, mission requests are being separated into priority tiers, such that higher priority requests are more likely to be satisfied, while lower priority requests might be dropped (Shouraboura, Johnston, and Tran 2016). Secondly, the DSN is adding limits on the amount of time that missions can submit for scheduling in the higher-priority requests, based on a forecast loading calculation of what can actually be supported (Werntz, Loyola, and Zendejas 1993) (Johnston and Lad 2018).

The intended effect of these changes is to make the scheduling problem given to the BOP easier. The focus of this paper is the new scheduler we are developing to assist with this. Since this scheduler is going to be used only by the BOP, the end goal is to generate a conflict-free schedule, even if this requires introducing violations. In addition, this new scheduler can take advantage of the new limits on schedule time per-mission. This allows separating the scheduling problem into two parts, taken one at a time. Together, this allows a different scheduling approach that gets much closer to solving the problem than prior work.

Formal Statement

The scheduling problem for the DSN is a specialization of the job shop scheduling problem. For each problem we are given the following:

- A Workspace $W = (t_s, t_e)$ with start time t_s and end time t_e.
- A set of Requests $\{R_i\}$. Each request $R_i = (t_s, t_e, M, \{r_j\})$ consists of a start time t_s and end time t_e, owning mission M, and a set of requirements $\{r_j\}$. Note that each request must fall within the workspace.
- Each requirement $r_j = (A, \{C_k\})$ consists of an "alias" A, which determines parameters about acceptable use of the antenna, as well as a set of constraints $\{C_k\}$.

The goal is to produce a schedule $S = \{G_i\} = \{(r, Y, \{T_j\}\}$ which contains a set of sets of tracks. Each track set specifies which requirement the tracks belong to r, as well as the type of the set Y. Each track $T_j = (M, a, \{E_k\}, t_s, t_e, \text{setup}, \text{teardown})$ has a mission M, an antenna a, a set of equipment to be used $\{E_k\}$, a start and end time, and some amount of setup and teardown that happen before and after the track, respectively. The choices of antenna and equipment have to be valid choices according to the requirement's alias A.

There are several simple constraints that apply to the whole schedule. Generally tracks are not allowed to overlap time ranges on a single antenna. Each track has to respect the viewperiods of its associated mission. Additionally, tracks which overlap at the same complex must respect limits on

equipment. Each complex only has a certain amount of each equipment type (known ahead of time), and using more than is available is a conflict.

In addition to these general constraints, there are the more complicated constraints attached to individual requirements ($\{C_i\}$). The complete list is too verbose to describe here in full, but further details are described in the 2014 AI Magazine article (Johnston et al. 2014). Some example additional constraints are:

- Total track duration (minimum, maximum, preference).
- Whether the tracks for this requirement can be split. If they can be split, minimum and maximum separation, as well as minimum and maximum duration of the individual tracks.
- More restrictive visibility constraints, e.g. due to signal strength requirements.
- Timing relationships to other requirements.

Approach

Given the complexity of the formal problem statement, we are relying on pre-existing schedulers for this new scheduler. In order to take advantage of the new high-priority/low-priority split, we first verify that all missions are abiding by their agreed-upon weekly limits, after which we remove all low-priority requirements from the schedule. Next, we try to find the best solution possible to the now-reduced problem, utilizing Squeaky Wheel Optimization (Joslin and Clements 1999) and a pre-existing scheduler, as well as some constraint relaxation. Finally, we opportunistically re-insert the low-priority requirements while avoiding conflicts. Overall, the majority of work on this effort has been on improving the solution to the reduced problem, rather than improving the filling of low-priority requirements, as creating schedules with fewer conflicts in the reduced problem pays dividends while inserting low-priority requirements.

Algorithm

The core of the scheduler is a standard Squeaky Wheel Optimization (SWO) setup, where we first schedule all requirements according to some ordering, and then iteratively permute the ordering based off of the results of the prior scheduling pass. The algorithm is described in Algorithm 1. To improve these results further, we leverage the fact that the most tolerated requirement violation is the total track duration. Whenever SWO stops providing meaningful improvements, we relax total track time constraints by reducing requirements located in the areas of highest contention (places in the schedule where many requirements are vying for tracking time). Finally, to keep track of our best schedule we utilize a scoring function, saving the order of requirements as well as the resulting schedule whenever we find a new best score.

Many aspects of the algorithm can be configured to explore different approaches:

- Metric
- Baseline scheduling algorithm

Algorithm 1 Pseudocode for SWO Scheduler

```
1:  bestScore ← SCHEDULE(ordering)
2:  save the schedule
3:  for r ← 0 to NUM_REDUCTIONS do
4:      REDUCE(requirements)
5:      newReductionScore ← SCHEDULE(ordering)
6:      bestReducedScore ← newReductionScore
7:      if newReductionScore > bestScore then
8:          bestScore ← newReductionScore
9:          save the schedule
10:     end if
11:     newOrdering ← GETINITIALORDERING()
12:     newOrderScore ← SCHEDULE(newOrdering)
13:     if newOrderScore > newReductionScore then
14:         bestReducedScore ← newOrderScore
15:         ordering ← newOrdering
16:     end if
17:     if newOrderScore > bestScore then
18:         bestScore ← newOrderScore
19:         save the schedule
20:     end if
21:     loop
22:         REORDER(ordering)
23:         newScore ← SCHEDULE(ordering)
24:         if newScore > bestScore then
25:             bestScore ← score
26:             save the schedule
27:         end if
28:         if newScore > bestReducedScore then
29:             staleIterations ← 0
30:         else
31:             staleIterations ← staleIterations + 1
32:         end if
33:         if staleIterations >= STALE_ITER then
34:             break
35:         end if
36:     end loop
37: end for
```

- Size and choice of reduction
- Number of reduction iterations
- Number of SWO iterations
- Conditions for breaking out of SWO (STALE_ITER)
- Algorithm for reordering based on previous schedule

Our goal with this work is to reduce the amount of work necessary to produce a schedule with minimal conflicts. Because we cannot know for sure what will help the BOP with their job, we instead used a simple quantitative metric as an approximation. We decided that a "good" schedule is one that minimizes the number of conflicts, while also maximizing the total scheduled tracking time. Finally we would like our algorithm to have some sense of fairness, such that we aim for roughly the same satisfaction percentage (ratio of scheduled time to minimum time constraint) across all missions. The selection of the metrics and baseline scheduling algorithms were chosen keeping this goal in mind.

Metrics

We tried out two different metrics to score schedules. Both metrics focused on track times, with some adjustments to ensure that schedules prioritized distributing track time evenly between missions.

The first metric that we tried was the average requirement satisfaction per mission. For each mission, find the set of requirements associated with that mission, total up the scheduled tracking time and minimum tracking time, and divide. Average these numbers over all missions to get the score. This metric prioritizes more tracking time, but it prefers adding tracking time to missions which request less tracking time. Although this helps with fairness, it does still have a bias towards small missions.

The second metric we tried was an application of Jain Fairness (Jain, Chiu, and Hawe 1998) to the mission satisfaction values from the previous metric. Jain Fairness is a fairness measure originally developed to help with networking algorithms. In our case, we are computing it using each missions satisfaction ratio r_i

$$jain = \frac{(\sum_{i=1}^{n} r_i)^2}{\sum_{i=1}^{n} r_i^2}$$

Using this instead of just averaging out the satisfaction ratios helps slightly with the bias towards small missions.

Baseline Schedulers

The two schedulers that we used to schedule individual requirements were both created in prior work (Johnston et al. 2014). We are using the basic Systematic Scheduler that underlies most DSN Scheduling Engine (DSE) schedulers, with no extra configuration, as a "strict" scheduler. This scheduler will only schedule tracks if it can find a conflict- and violation-free location, see Algorithm 2. This is nice because it prevents our resulting schedule from being swamped in conflicts and violations, but it does mean that we will have fewer tracks scheduled when all requirements have been attempted.

Algorithm 2 Pseudocode for Systematic Scheduler

```
for each valid start time do
    for each valid end time (starting at the end) do
        if this is a valid track then
            if there is leftover time then
                Recurse
            end if
            if the minimum time is satisfied then
                Return the current solution
            end if
        end if
    end for
end for
```

Algorithm 3 Pseudocode for the Shuffle Scheduler

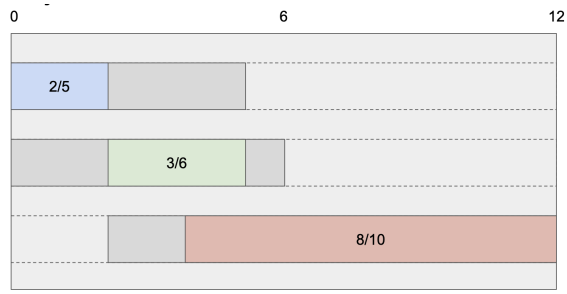
```
for each track in this requirement do
    for each valid asset choice (shuffled) do
        Get all legal intervals for this track/asset pairing
        Shuffle the list of legal intervals
        for each legal interval (shuffled) do
            if this is a valid track then
                Continue to the next track in the requirement
            end if
        end for
    end for
end for
```

The other scheduler we are using is the "Relayout" or "Shuffle" scheduler. This scheduler utilizes information about where and how the track was previously scheduled, and tries to find a new location for the same track, as shown in Algorithm 3. This scheduler also avoids placing tracks where they would introduce conflicts or violations, and thus has similar tradeoffs as the previous scheduler.

There was one other scheduler that we tried using, but found inadequate. The "Initial Layout" scheduler is a refinement of the "Systematic" scheduler. In addition to performing a strict scheduling pass, the initial layout scheduler also attempts to schedule requirements with a series of successive relaxations, in order to produce tracks that at least meet a subset of the most important constraints. These relaxations means tracks which cannot be scheduled easily will end up with either violations or conflicts. This has the downsides of both causing later tracks to have more issues scheduling, as well as causing the resulting schedule to have more conflicts. These issues meant that initial layout failed to perform at the same level as the other two schedulers.

Reduction

Initial testing was done with a flat 20% reduction of total track duration across all requirements. While this produced good results, it unnecessarily penalized requirements that were not be contributing to crowding just because parts of the schedule had high contention. To alleviate this, an algorithm was developed that attempts to find areas of the sched-



(a) Three requirements, each with their reservation ratio written out. The first two requirements scheduled successfully, and the third requirement failed to schedule.

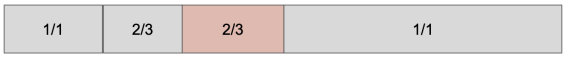
(b) $unusedRange$

(c) $totalBadRes$

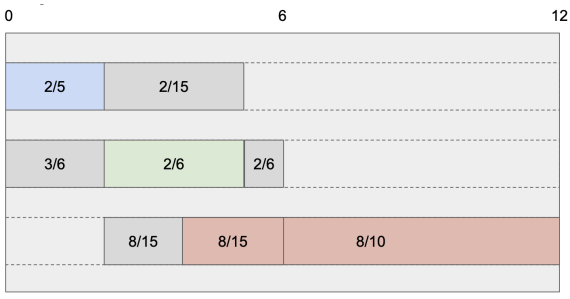
(d) $excessRes$

(e) $badRes/excessRes$

(f) Updated $badRange$



(g) $impact$



(h) Original reservations multiplied by $impact$

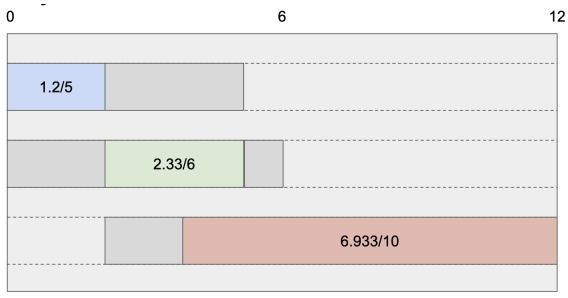


Figure 1: Visual Illustration of the steps of Reduce
This is with three requirements and a single antenna over 12 hours

ule with high contention, and reduce all requirements that contribute to it.

Given the high-level goal of the reduction algorithm, there were two specific aims. First is to avoid penalizing requirements that are in low-contention parts of the schedule for the existence of high-contention parts of the schedule. The second is that wherever possible the reduction should be spread equally. Together, this would hopefully lead to reductions that make the schedule feasible, without either reducing single requirements by extreme amounts, or reducing every requirement by the same amount.

The main tool that we used to achieve this is a notion of the space that a requirement could use when scheduling, what we call a "reservation". This is a more specific object than just the time bounds on the requirement, incorporating information like spacecraft visibilities as well as antenna downtime. Here we define a reservation $res(a, t)$ to be a function that maps antenna a and time t to a value. This leads to a trivial definition of addition, as well as the following definitions:

$$\text{Integral}(res) = \sum_{i=1}^{n} \int_{t_s}^{t_e} res(a_i, t)\, dt \tag{1}$$

$$\text{Duration}(res) = \sum_{i=1}^{n} \int_{t_s}^{t_e} \begin{cases} 1 & \text{if } res(a_i, t) \neq 0 \\ 0 & \text{otherwise} \end{cases} dt \tag{2}$$

$$\text{SetValue}(res, ratio)(a, t) = \begin{cases} ratio & \text{if } res(a, t) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$\text{FindUnused}(res) = \{(a, t) \mid res(a, t) = 0\} \tag{4}$$

$$\text{Intersect}(res, range)(a, t) = \begin{cases} res(a, t) & \text{if } (a, t) \in range \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Using these functions, we can write out the full description of our algorithm for reduction in Algorithm 4. The algorithm first looks at the unscheduled requirements, and stores info about their reservations. Using these reservations, we then try and calculate some way of fitting both the missing amount, as well as the currently existing amount, and reduce any requirements whose reservations touch the affected space. A brief worked example is shown in Fig. 1.

Empirical Evaluation

In order to evaluate our algorithms, we need to obtain workspaces (inputs for a single week's schedule) that reflect real-world usage. A pilot study was done in the past that gathered the relevant inputs, including limits and preferences, for a single week. This workspace is a good starting point for evaluation, but more workspaces are necessary to fully test our algorithms. Currently the DSN is not gathering the requisite inputs as part of the scheduling process, but changes are being made to gather this input in the future.

In future work, in addition to the schedules being input by current DSN users, we would like to investigate creating synthetic DSN schedules for testing purposes. One way

Algorithm 4 Pseudocode for Reduction step

```
goodResList ← empty list
badResList ← empty list

for requirement ∈ requirements do
    reservation ← RESERVATIONS(requirement)
    allowableTime ← DURATION(reservation)
    currentTime ← DURATION(requirement)
    minTime ← MINTIME(requirement)
    if currentTime > 0 then
        ratio ← currentTime/allowableTime
        goodRes ← SETVALUE(reservation, ratio)
        PUSH(goodResList, goodRes)
    end if
    if currentTime < minTime then
        ratio ← (minTime − currentTime)/
allowableTime
        badRes ← SETVALUE(reservation, ratio)
        PUSH(badResList, badRes)
    end if
end for

totalGoodRes ← SUM(goodResList)
unusedRange ← FINDUNUSED(totalGoodRes)
excessRes ← INTERSECT(totalBadRes, unusedRange)
totalBadRes ← SUM(badResList)
for badRes ∈ badResList do
    HANDLEEXCESS(badRes, excessRes)
end for

totalBadRes ← SUM(badResList)
totalBadRes ← totalBadRes + 1
impact ← 1/totalBadRes

for requirement ∈ requirements do
    reservation ← RESERVATIONS(requirement)
    newTotal ← INTEGRAL(reservation ∗ impact)
    currentTotal ← MINDURATION(requirement)
    reduction ← newTotal/currentTotal
    REDUCE(requirement, reduction)
    REDUCE(reservation, reduction)
end for
```

Algorithm 5 Pseudocode for HandleExcess

```
totalExcess ← INTEGRAL(badRes/excessRes)
currentBadTotal ← INTEGRAL(badRes)
newBadTotal ← currentBadTotal − totalExcess
unusedRange ← FINDUNUSED(excessRes)
usedRange ← unusedRange^C
badRes ← INTERSECT(badRes, usedRange)
newDuration ← DURATION(badRes)
newRatio ← newBadTotal/newDuration
badRange ← SETVALUE(badRes, newRatio)
```

to do this is to take historical schedules and add the relevant bits of information that will be in future schedules. This means setting weekly limits for each mission in the historical schedule, as well as assigning preference values to each requirement.

The other potential method for creating synthetic DSN schedules would be to create de novo schedules that still maintain key features of real-world schedules.

Results

Some preliminary results were computed with a single input schedule, showing significant promise for this method. A collection of statistics about the resulting schedules can be found in Figure 2. The statistics show a factor of 10 reduction in antenna and equipment conflicts, the two types of conflicts that are hardest to resolve manually. The best scheduling runs manage to remove these conflicts while still maintaining most of the tracking time originally present in the schedule. At present this single schedule is the extent of available input data, and more work is being done to acquire more varied workspaces. Even with this single schedule, much of the data in the chart suggests future work that could be done in targeting specific issues like MSPA conflicts or trackQuantize violations.

Conclusion

This work is still early. We have presented a scheduling problem as well as a relaxation problem. We have presented an approach to the relaxation problem designed to avoid relaxing wherever possible, and an approach to the scheduling problem utilising this relaxation method. In future work we hope to gather more workspaces to test with, and to perform user testing with the DSN schedulers who will be utilizing the tool (BOPs).

Acknowledgements: This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

Imbriale, W. A. 2003. *Large Antennas of the Deep Space Network*. Wiley.

Jain, R.; Chiu, D.; and Hawe, W. 1998. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR* cs.NI/9809099.

Johnston, M. D., and Lad, J. 2018. Integrated Planning and Scheduling for NASA's Deep Space Network – from Forecasting to Real-time. In *SpaceOps*.

Johnston, M.; Tran, D.; Arroyo, B.; Sorensen, S.; Tay, P.; Carruth, J.; Coffman, A.; and Wallace, M. 2014. Automated Scheduling for NASA's Deep Space Network. *AI Magazine* 35:7–25.

Joslin, D. E., and Clements, D. P. 1999. Squeaky Wheel Optimization. *Journal of AI Research* 10:353–373.

Shouraboura, C.; Johnston, M. D.; and Tran, D. 2016. Prioritization and Oversubscribed Scheduling for NASA's Deep Space Network. In *ICAPS SPARK Workshop*.

Werntz, D.; Loyola, S.; and Zendejas, S. 1993. FASTER - A tool for DSN forecasting and scheduling. In *AIAA Computing in Aerospace Conference, 9th*. San Diego, CA: AIAA.

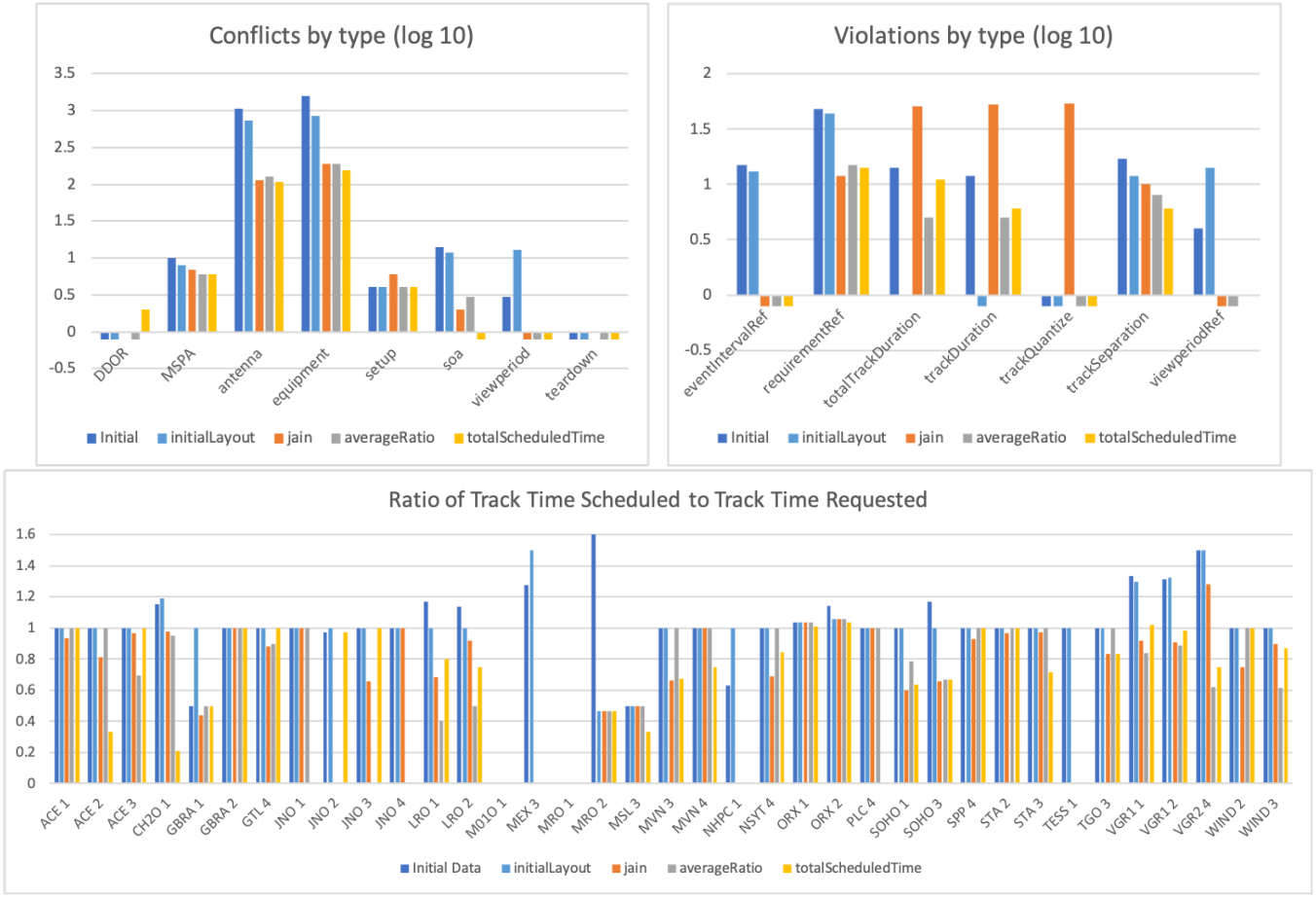


Figure 2: Summary of the different algorithms performance. ”Initial” is the initial state of the schedule before scheduling. ”initialLayout” is running the initialLayout scheduler on all requirements once. The remaining three bars are all variations of the new preference-based scheduler, using the strict subscheduler. ”jain” uses jain fairness as a metric, ”averageRatio” uses requirement satisfaction averaged over missions, and ”totalScheduledTime” uses the total scheduled time across all requirements.