Experience with a Constraint and Preference Language for DSN Communications Scheduling

Bradley J. Clement, Mark D. Johnston, Steven R. Schaffer, Daniel Q. Tran
Artificial Intelligence Group, Jet Propulsion Laboratory
firstname.lastname@jpl.nasa.gov

Abstract

As part of a new scheduling system for allocating NASA Deep Space Network (DSN) ground antenna and equipment resources to space missions, we have designed and implemented a requirements language oriented towards the clear and efficient specification of mission communication constraints and preferences. In this paper we discuss the requirement language itself, the model into which the language is translated for automated scheduling, and the impact it has had on scheduling algorithm design. The scheduling engine supporting the language is being integrated into a mixed-initiative scheduling system for a 2008 delivery.

1. Introduction

NASA's Deep Space Network (DSN) maintains and schedules 16 large antennas (26m, 34m, and 70m) to support interplanetary missions, radio and radar astronomy, and also some Earth orbiting missions. Services include command uplink, data downlink, ranging, and navigation among others. There are around 150 missions listed as DSN users, with 20 to 25 spacecraft serviced per month.

In an effort to ease the burden of human scheduling, negotiation, and conflict resolution in allocating resources of the DSN, we have developed a scheduling engine that implements a new requirements language. The language is XML based and designed for clear and efficient specification of mission communication requirements. The Aspen planning system [4] uses these requirements to create schedules and resolve conflicts using local and systematic search techniques.

The language and the scheduling engine are part of a mixed-initiative scheduling system, currently under development as the new scheduling system for the DSN. The initial delivery to users for evaluation is scheduled for early 2008, with an initial operational capability planned for late 2008. Users have estimated that this more powerful and flexible scheduling system can save several millions of dollars per year in mission operations costs.

The language captures all the key metric resource, durative action, and temporal constraints of a mission's communication requirements. DSN service activities can be requested individually or in bulk. Shared services are also represented, for example simultaneous communications with multiple spacecraft at Mars. The language expresses constraints and preferences on both local contact timing and non-local factors such as total contact time, gap time, and gap-to-track ratio over some time period. The users and the scheduling engine adjust the schedule within the flexibility of the constraints and to accommodate indicated preferences as much as possible.

This paper discusses our experience in developing the requirement language, the modeling the language in Aspn, and modifying search and repair strategies in Aspen based on language features. This paper does not describe scheduling constraints outside the language, details of the system architecture, details of the algorithms, or related scheduling algorithm work that are specified elsewhere [5],[6]. We discuss other work related to the requirement language of this paper in Section 6.

2. Background on Aspen Models

Here we briefly describe a subset of the modeling language features of Aspen [4] that are used to implement the requirement language. In Aspen, a task is defined to have decompositions, parameters, dependencies, and reservations.[1] A *decomposition* of a task is a choice of sets of child tasks to be instantiated with the task as their parent. A *parameter* of a task is a variable that is assigned from the outside as input or computed as a function of other parameters. *Dependencies* of the task specify the connection of parameters through functions and through assignment between a parent tasks and their children. When a value of a parameter changes, it propagates through

[1] We use the term "task" to refer to an Aspen activity because we wish to use the DSN meaning of "activity" as a service provided for a mission. DSN activities and requirements are modeled as Aspen activity schemas.

dependencies to *dependency functions* for which the parameter is an argument and to the parameters of child tasks to which it is assigned. A *reservation* of a task is a constraint or effect on a state or resource timeline. A *timeline* is represented as a function of time to value. For example, a decoder resource timeline keeps track of how many decoders are available for use. A viewperiod state timeline keeps track of when a spacecraft will be in view of an antenna.

3. Modeling the Basic Requirement Types

The most basic goal of the requirement language is to help a mission convey its service needs, flexibility, and preferences in terms of timing and resource use to other missions and users and to the scheduling engine for aid exploring scheduling options to avoid or resolve conflicts or to improve the schedule. The language also aims to make it easier for a user to specify many similar activities concisely as a group. A third purpose of the language is for missions to be able to manage their activities more easily by editing requirements instead of repeating edits for individual activities.

The different types of requirements (named in the sub-section headings 3.1 - 3.5) share many basic parameters and some layer on others. The structures of these types are general and may apply to other problem domains. The single-activity, continuous, and periodic types originated from a past attempt to get users to manage their activities through requirements.

3.1. Single-activity

A single-activity requirement specifies service for a single *pass*, i.e. the time period the spacecraft is in view of a *complex* of colocated antennas. It is a basic building block of other requirement types, and its parameters are used by all of the others. It is meant to capture flexibility in start time, duration, and usable antennas. As shown in Figure 1, the `min/max individualTrackDuration` and `absoluteTime` parameters specify timing flexibility. Antenna flexibility is specified in `assetSpec` as choices (using an `OR` operator) of antenna IDs or by group name, and multiple antennas can be requested by grouping them with an `AND` operator. The model in Aspen currently only supports a single `AND` at the top level with no restriction on `OR` nesting.

The model of a single-activity requirement enforces its timing constraints using parameter dependency functions (shown in Figure 1 as precedence constraints

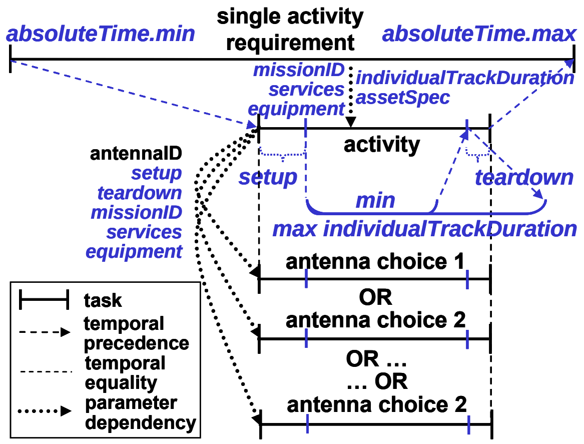


Figure 1. Model of the single-activity requirement with parameters in *italics*.

on a child task) to constrain the start time and durations to fall within the `absoluteTime` bounds. In addition, service `setup` and `teardown` time durations are looked up using a dependency function of the `services` being performed. A *track* is the service time between the `setup` and teardown periods of an activity. The duration of the track is bounded by a dependency function to the `individualTrackDuration` range.

The single-activity requirement is modeled as a task that encloses the activity task, passing down many parameters as shown in Figure 1. The `assetSpec` is modeled as a string parameter, capturing the `AND/OR` nesting of individual `antennaIDs`. The `assetSpec` is parsed, and each `OR antennaID` choice is assigned a task decomposition choice, each specific to a `missionID-antennaID` pair. An `AND` grouping is modeled as a separate decomposition choice (not shown in the figure) for a continuous requirement (see Section 3.2) where the `overlap` is for the entire track.

Many parameters pass down to the “antenna choice” task for making reservations on timelines such as the viewperiod timeline (e.g. the missionA-antenna3-viewperiod state timeline must be “in view”). The `equipment` string parameter is used to encode sharable equipment resources needed for the service activity (such as specific types of receivers).

3.2. Continuous

A *continuous requirement* represents uninterrupted service for longer than a single pass, so the service is passed off to another antenna, creating another single-activity instance. This is typical for providing a spacecraft navigation during launch, trajectory control, and planetary entry, descent, and landing. As shown in Figure 2, the track portion of the activities have a specified `overlap`, and the continued duration of the

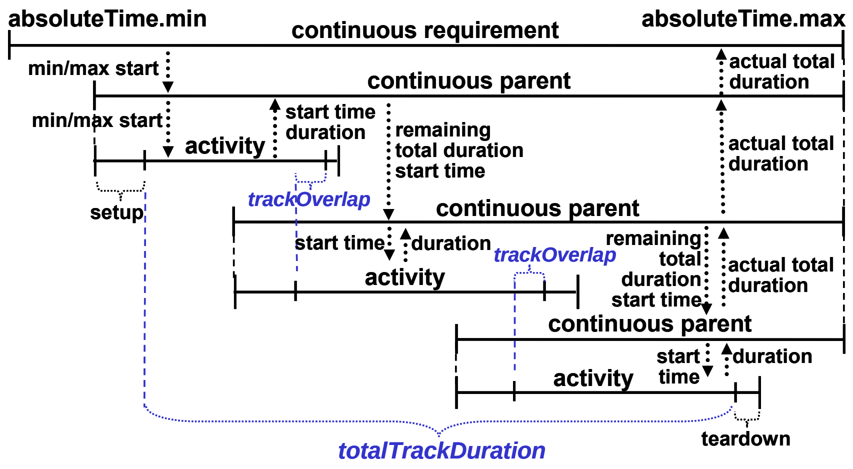


Figure 2. Model of a continuous requirement with its *unique parameters*.

combined service (without double-counting the overlap time) must add up to `totalTrackDuration`.

This requirement is modeled in Aspen as a task that recursively decomposes into successive single-activities until the total duration is achieved. Like the single-activity requirement, the start time of the first activity is restricted with dependency functions so that the overall continuous sequence is contained inside the `absoluteTime` window. The last activity may accumulate a total duration beyond the `totalTrackDuration` and extend beyond the end of the `absoluteTime` window with the minimum `individualTrackDuration`. In rolling out the activities, the start time of the next activity is calculated as a dependency function in the prior continuous parent from the start time and duration passed up from the previous activity. The next start time is then passed to the next continuous parent. Similarly, the remaining total duration is passed through the continuous parent tasks subtracting the track duration of activities along the way. To determine if the `totalTrackDuration` is achieved, the actual total duration is calculated in the last parent and passed up/back to the top, where the constraint is checked.

3.3. Segmented

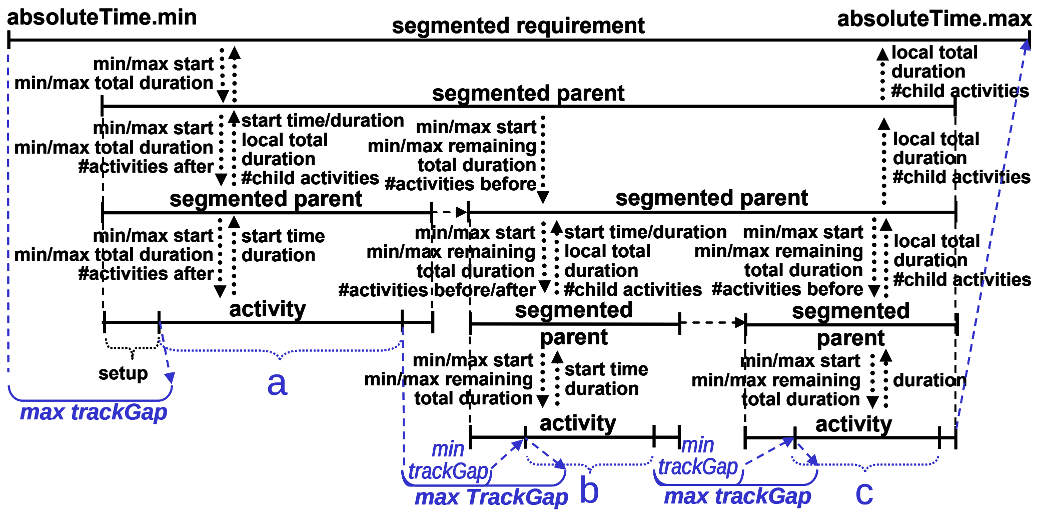
A *segmented* requirement is useful for specifying the fraction of time a spacecraft needs service and the constraints on gaps between services. The main motivation for this requirement is that spacecraft often accumulate data at a particular rate and need to downlink it often so that the onboard storage does not fill up and start losing data. At the same time, too much downlink time can clear out the data, and there will be none to send, resulting in wasted antenna time.

As Figure 3 depicts, the segmented requirement states that a collection of activities is needed within an `absoluteTime` window where the total track time of the service is within a specified `totalTrackDuration` range, the gaps between services are all within a specified `trackGap` range, and/or the ratio of time gap gaps between tracks and the track time is within some `gapToTrackRatio` range. Since `absoluteTime` is a fixed timeframe, the total duration range and gap-to-service ratio can be computed from each other:

total gap time =
`totalTrackDuration * gapToTrackRatio` =
`|absoluteTime| - totalTrackDuration`.

The segmented requirement is modeled as a task that decomposes recursively into “segmented parent” tasks, each decomposing into either two temporally ordered new segmented parent tasks or a single activity as a leaf. We chose this binary tree structure so that the scheduling engine could add or delete tasks in the middle of a sequence without disturbing other activities. When Aspen changes a task’s decomposition, it first *abstracts* the task by removing all child tasks underneath and then re-detailing (re-decomposing) and scheduling new tasks for the new decomposition choice.

For example, in order to divide the third activity of a continuous requirement into two, the engine would abstract the continuous parent, removing the third and all following activities of the requirement. Then the engine would recreate and reschedule all of those activities. The rescheduling is necessary for the continuous requirement anyway since the fixed overlap constrains the start time to one value for all but the first activity, so changing one end time will change the start of all following activities. However, the gap range for the segmented gives its activities slack so that it is



$$\text{total} = a + b + c$$

$$totalTrackDuration.min \leq \text{total} \leq totalTrackDuration.max$$

$$gapToTrackRatio.min \leq (|\text{absoluteTime}| - \text{total}) / \text{total} \leq gapToTrackRatio.max$$

Figure 3. Model of a segmented requirement with its *unique parameters*.

easier to keep the changes local. So, when a lowest-level segmented parent task is re-detailed, its child activity may often be the only one affected.

Decomposition of the segmented parent task is performed similarly to an in-order tree traversal so that activities are instantiated and scheduled in time order. We chose to restrict the first activity to start within the maximum `trackGap`. The range of the next activity's start time is passed down from the activity's segmented parent ancestors. The previous activity's start time and duration propagate up the tree to the parent it has in common with the next activity, and that parent calculates the start time range for the next activity. The remaining total track duration range is also calculated for all segmented parents like for continuous requirements. The scheduling engine uses this range to spread tracks evenly and to decide whether to split segmented parents during decomposition. The numbers of activities before, after, and below are also propagated and computed to help make the same scheduling decisions. The use of this information in decomposition heuristics is described in Section 5.3.

3.4. Periodic

A *periodic requirement* is meant to make it easy to request many of the same kinds of activities at once, especially capturing the need for activities that are regularly spaced apart, such as "a track every day" or "3 tracks each week." Specifically, a periodic requirement asks for a `tracksPerPeriod` number of instances within a time interval of size `trackWindow` every specified `period` of time starting at the minimum `absoluteTime`, as depicted in Figure 4.

Like the continuous type, a periodic is modeled to recursively decompose and roll out parent tasks. However, each of these also have a "window parent" child task that recursively decomposes, generating `tracksPerPeriod` activities which are constrained to fit within a duration of `trackWindow` into the period in the same way an activity is confined to the `absoluteTime` of a single-activity. Notice that the activity tasks have no ordering constraints within the same `trackWindow`.

Although not shown in Figure 4, the `period`, `trackWindow`, and `tracksPerPeriod` parameters pass down through the decomposition. The engine currently assumes that the periodic requirement is for single-activities but there are plans to be able to also decompose into segmented and continuous tasks.

3.5. Event Interval

An *event* is a named time point, interval, or set of intervals (e.g. apogee, eclipse, day shift). Every mission plans operations based on their own set of events, so the scheduling system is designed for users to schedule with respect to their own event definitions. *Static events* are those that the scheduling engine cannot change. This is the normal sense of events—the engine does not decide when eclipses or vacations occur. *Dynamic events* are those that are events that are not static. The requirement language allows dynamic events that reference the time intervals of all of a requirement's activities or of the *n*th activity from the start or end.

There are two original motivations for dynamic events for satisfying temporal constraints between requirements. (1) Different spacecraft sometimes need

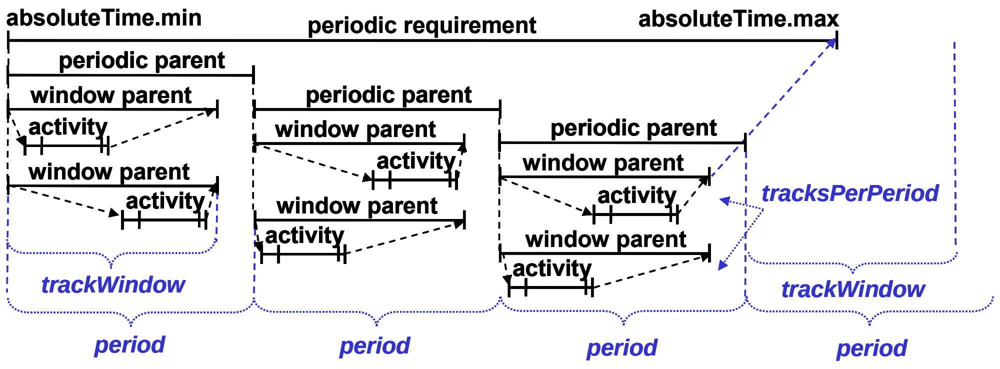


Figure 4. Model of a 2-`tracksPerPeriod` periodic requirement with its *unique parameters*. Only three `trackWindows` will fit within the `absoluteTime` bounds.

to have contacts that are simultaneous (the case of Cluster 1 and 2) or back-to-back (STEREO A and B). (2) Missions may have different requirements for different services, but the same activities must fulfill both. For example, Mars missions must keep track of when they receive uplink services during a pass because two spacecraft/rovers can share an antenna as long as only one is uplinking. Thus, since downlinks always accompany uplinks, the engine schedules each uplink within a downlink.

An *event interval requirement* is similar to a periodic requirement, except that the time windows into which instances of activities are placed are a specified set of intervals instead periodic windows of the same size. For example, a mission may request two activities between every occlusion.

The model for the event interval requirement uses the periodic model but adds a string parameter of the set of intervals and dependency functions to set the timing of window parents to intervals of the set.

So, like the periodic, the event requirement is also useful for requesting many activities at once, and it has an additional convenience in that if there are many requirements specified relative to an event (e.g. launch), if the event time changes, the requirements do not. The user interface of the scheduling system allows any date/time of a requirement to be an event.

4. Other Requirement Attributes

The last section explained the structure of the basic scheduling requirement types. We now describe some other requirement attributes independent of these types.

4.1. Event constraints

Additional temporal constraints may be specified on activities in reference to external events (as defined in Section 0). The requirement language expresses both "within" and "avoid" logic for specified event sets. For example, a mission's tracking activities could be constrained to avoid planetary occultations, to fall within daylight hours, and to avoid operator vacation time. While a event interval requirement specifies how activities are created, the event constraint specifies where (in time) they are scheduled.

The original choice for modeling these constraints was to create a timeline for each, but this required an unknown amount of effort to ensure that Aspen would support dynamically defining timelines and tasks. This would be necessary when the scheduling engine accepted requests to add and remove different kinds data in the middle of a session.

Instead, we encode the events and their constraints into a string parameter, and a dependency function of the start and end times evaluates the string to see if the constraint is met. If the constraint is violated, the activity creates a conflict through a reservation on a timeline devoted to event checking.

The string encodes the constraint as list of allowable interval relations, Allen relations [1], and an indicator of whether the constraint must be satisfied for all or at least one of the event's intervals. For example, the string could encode that the activity's time interval must be during or containing (but not having the same start or end as) one of a following list of intervals. The XML interface currently only exposes the "within" and "avoid" constraints, but we expect users will later exploit the much greater expressiveness of this underlying representation.

4.2. Override

Missions will occasionally interrupt a sequence of services with another activity. For example, regular, periodic tracking passes every other day may be suspended on one day for a trajectory control maneuver (TCM). Breaking the periodic requirement into two pieces does not work since the day of the TCM may change. Instead, we directly represent this kind of relationship. Any requirement can be specified to `override` any other. This means that anytime a track A of a requirement overlaps a track B of another requirement it overrides, track B is removed, but B still satisfies its requirement.

This is modeled by decomposing the overridden track to a noop (phantom activity), so that no reservations are made on the timelines. The engine also does not report the activity in schedules and conflicts it returns. A dependency function of the activity's start time and duration determines when it is overridden by seeing if there are any overlapping activities of overriding requirements. This function is also triggered whenever the start time or duration of an activity of an overriding requirement changes through one of its dependency functions. These dependency functions are written in C++ and contain static data structures keeping track of which requirements override which, so that checking overrides is simple.

5. Impact on algorithm design

The scheduling engine will generate an initial set of activities by decomposing requirements based on the corresponding models described in Section 2. As the engine creates each activity, it schedules it by choosing a start time, duration, and allocation of antennas. We will refer to a choice of these three values as a *state* of the activity. A *state space* is the allowable combination, of choices for one or more activities. The scheduling engine's state spaces only includes states that meet the constraints of the requirements (excluding dynamic events constraints as we explain later) and the spacecraft's viewperiods.

5.1. State scoring

The states are scored by common and type-specific criteria. The common criteria are

- ♦ whether all of the other constraint rules (those of the DSN, not defined by the requirement) are met,
- ♦ how well they can satisfy dynamic event constraints (which we will explain later), and
- ♦ the closeness of values of the state to their corresponding preferred values.

For the initial layout, the activity is assigned the highest scoring state. If there are no legal states, the activity is rescheduled to a state close to its current state and maybe satisfying some of the requirement's constraints. This is how the single-activity requirement is scheduled. There are other criteria used for continuous and segmented types discussed later.

When resolving conflicts, the engine iteratively chooses a conflict, chooses an activity (or a group of activities) involved in the conflict, and reschedules them in one of three ways, depending on the kind of conflict and type of requirement(s) involved:

1. stochastically assigns a state from the legal space of choices for the activity, each weighted by score,
2. abstracts (removes) the activities (for a single continuous or segmented requirement) and regenerates them through decomposition, applying method 1 to reschedule each, or
3. *systematically searches through the combined state space of the chosen activities to find and assign a state for each that together resolves the conflict if such a combination exists.*

The continuous requirement has additional criteria:

- ♦ the number of subsequent activities that are moved by assigning the state and
- ♦ whether the subsequent activities would have legal state choices.

The segmented requirement uses the same score criteria as the continuous plus others

- ♦ for spreading activities evenly across the `absoluteTime` range and
- ♦ for improving the total duration when not within `totalTrackDuration` bounds.

If the total accumulated duration falls short of the evenly-spread ideal by more than its longest legal duration, earlier start times and longer durations are preferred. If the total is larger than the ideal by the min duration, then later start times and shorter durations are preferred. Otherwise the total duration is acceptable, and the score is then based on how close the start time is to an evenly-spread ideal computed as follows:

```
ideal start time =
   absoluteStartTime.min +
   (|absoluteStartTime| *
    accumulated total track duration)  /
   totalTrackDuration.middle –
   state.duration – trackGap.middle
```

In situations when the segmented actual total track duration is outside the `totalTrackDuration` bounds, each state is also scored by how much its duration increases or decreases in the direction to get the total within the `totalTrackDuration` bounds, getting a zero score if not improving.

5.2. Dynamic event constraints

The scheduling engine does not restrict the state space based on dynamic event constraints as it does

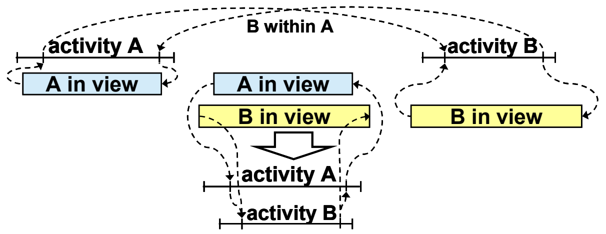


Figure 5. Dynamic event conflict resolution

with static because doing so could prevent the possible resolution of an event conflict. Suppose that a state of an activity that satisfies its requirement A can violate a dynamic constraint that another requirement B has with A. We cannot simply remove the states that cause this violation from the state space because it can keep the engine from exploring some legal combinations of states, those that require rescheduling activities of both A and B. For example, as shown in Figure 5, if B's activity must be within A's, but A's is not in a legal viewperiod of B, B will not have any legal states. A's activity could also have no states if it were required to cover B, preventing them both from being able to move to some other time/antenna where they could satisfy all constraints, shown at the bottom of Figure 5.

Since the state space of activities cannot be reduced for dynamic event constraints, scoring is used to favor states that satisfy these constraints, as mentioned in the second bullet of Section 5.1. Note that when scoring whether an activity state meets constraint rules, only the constraints defined for the activity's requirement are checked—not those that other requirements have on it. So, this scoring is based on event constraints defined in other requirements. The only design choice here is how to find for an activity the others that have an event constraint on it. The engine does this by looking at nearby constraints and effects on common timelines to find the constraining activities and chooses states that help satisfy those events constraints.

5.3. Segmented task decomposition heuristics

In Section 3.3, we discussed the model of the segmented requirement, how the engine decomposes the task, and how the task decomposition propagates information in parameters to inform decomposition heuristics. Here we describe how the decomposition heuristic uses this and other related information.

The decomposition of the segmented requirement task is delicate because the number tasks and their durations are not fixed, allowing for a great number of possible activity sequences, but viewperiod, event, and resource constraints can complicate the choices. In our experience, the performance of the engine was most influenced by the choice of these heuristics. For one test, performance ranged from generating a schedule with ~100 conflicts and growing the conflicts during rescheduling to a schedule with 6 conflicts on average and resolving). The effort put into the heuristics was a significant portion (~10%) of the overall development.

When a requirement is only partially decomposed, it is difficult to determine whether to split a segmented parent to add another activity in decomposing the entire task tree or only a part. In this case, the engine benefits from knowing

- ♦ the number of activities there are before, after, and underneath this parent task in question,
- ♦ the prior, remaining, and local total track duration,
- ♦ how many segmented parent tasks are there after this one that are not yet decomposed,
- ♦ when the next existing activity starts,
- ♦ whether it is the next activity after this task,
- ♦ the min/max number of activities in the tree,
- ♦ the number and duration of activities outside this requirement constrain this one, and
- ♦ the number and duration constrained by this one.

In some cases the decision to split is obvious; e.g. there will not be enough maximum remaining total duration to create a minimum duration activity. Otherwise, the engine compiles the information above into a few metrics (each ranging in value from 0.0 to 1.0) for computing the probability to split. We do not give detailed definitions or formulas for these but list them to give an intuition of our experience in balancing them to keep the decomposition from making obvious errors without restricting the combined state space.

- ♦ `howFarIn` = fraction of time into the requirement,
- ♦ `deepEnough` = task is deep enough into the tree,
- ♦ `enoughTrackRate` = enough prior total track duration per time,
- ♦ `enoughDuration` = enough overall track duration,
- ♦ `enoughSpread` = enough activities for an even spread of duration over time, and
- ♦ `enoughActivities` = enough activities overall.

The overall formula for the probability of *not* splitting (i.e. no need for another activity here) is

```
((1.0 – howFarIn) * deepEnough +
 0.5 * enoughTrackRate +
 howFarIn * (enoughDuration + enoughSpread
             + enoughActivities)) /
(2 * howFarIn + 1.5)
```

The engine uses the `howFarIn` metric to emphasize the importance of others with respect to the beginning or end of the activity sequence. For example, expanding deep enough is important to balance the tree for more flexibility in local problem solving, but the `deepEnough` metric is not important (and potentially damaging) for choosing whether to add activities to the end. Likewise, there will never be enough activities or total duration at the beginning of the decomposition, so placing importance on the last three metrics up front would result in a very unbalanced tree. The dividend of the formula normalizes the value to a probability ranging between 0.0 and 1.0.

6. Related Work

A design of the overall scheduling system [6] gives more context for the role of the DSN scheduling engine and the challenges faced in other parts of the project. An earlier report of this work [5] discusses other details of the overall scheduling problem, resource modeling, search strategies, and how other scheduling problems and approaches relate to that of DSN scheduling. The report also includes some history of prior attempts to automate DSN resource allocation.

Some other network scheduling systems use similar higher-level goal specifications. Mexar2 [3] addresses the problem of scheduling downlinks for the Mars Express Orbiter in order prevent the overwriting of data stores onboard. The segmented requirement is also meant to capture this problem but does not treat it as directly as Mexar2 as it is meant to provide a simple abstraction to avoid the complexity of representing operations at this more detailed level for each mission.

The European Space Agency tracking network Management and Scheduling System (EMS) [9] represents and schedules periodic requests with min/max gap and other similar constraints. EMS also uses iterative repair as the engine discussed in this paper does along with systematic local search. Continuous tracking and other single-activity services are allocated by the NASA Space Network Demand Access System using a first-come, first-serve policy with no rescheduling [8]. Scheduling for the Air Force Satellite Control Network [2] is treated as oversubscribed variant of job-shop scheduling with additional constraints, similar to single-activity scheduling for the DSN.

7. Conclusion

In designing a scheduling engine for a requirement language that more directly and concisely captures higher-level understanding of scheduling goals, we found complexity in modeling and search heuristics inside an iterative repair framework. Our solutions to these problems point out that hierarchical task models of basic goal structures are non-trivial, and local search heuristics can be intricate and fragile. In future work, we expect to expand the scope and role of systematic search in the scheduling engine to help offset the burden of developing and maintaining heuristics.

8. Acknowledgments

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with NASA.

9. References

[1] Allen, J. F., "Maintaining knowledge about temporal intervals." *Comm. of the ACM* 26(11) pp.832—843, 1983.

[2] Barbulescu, L., Watson, J., Whitley, D. and Howe, A. "Scheduling Space-Ground Communications for the Air Force Satellite Control Network." *Journal of Scheduling*, Vol. 7, Issue 1, pp. 7-34, January 2004.

[3] Cesta, A., Cortellessa, G., Fratini, S. and Oddi, A. "An innovative product form space mission planning an *a posteriori* evaluation." In *Proc. of International Conference on Automated Planning and Scheduling*. pp 57—64, 2007.

[4] Chien, S., Rabideau, G., Knight, R.., Sherwood, R., Engelhardt, B., Mutz, D., Estlin, T., Smith, B., Fisher, F., Barrett, A., Stebbins, G. and Tran, D. "ASPEN - Automating Space Mission Operations using Automated Planning and Scheduling." in *Proc. of SpaceOps*,. Toulouse, France, 2000.

[5] Clement, B. J., and Johnston, M. D. "The Deep Space Network Scheduling Problem." In *Proc. of the Innovative Applications of Artificial Intelligence Conference*, 2005.

[6] Clement, B. J., and Johnston, M. D. "Design of a Deep Space Scheduling System." In *Proc. of the 5th International Workshop on Planning and Scheduling for Space*, 2006.

[7] Fayyad, Kristina E., Hill, R. W., Jr., and Wyatt, E. J. "Knowledge engineering for temporal dependency networks as operations procedures." In *Proc. of the AIAA Computing in Aerospace Conference*, pp. 1040—1047, Oct, 1993.

[8] Gitlin T. A., Kearns, W., Horne, W. D. "The NASA space network demand access systems (DAS)." In *Proceedings of SpaceOps*, 2002.

[9] Nizette, M., Götzelmann, and M., Calzolari, G. P. "Automated planning of ESA's tracking network services." In *Proceedings of the 7th International Symposium on Reducing Costs of Spacecraft Ground Systems and Operations* (RCSGSO), Moscow, pp. 11—15, June 2007.