|  |
| --- |
| Roadmap to Transits in Optimizer |
| SAROPS Version 2.2 |
|  |
| This document describes the initial approach to getting transit considerations into the optimizer |
| This document contains data that is delivered to the Government with Unlimited Rights in accordance with contract HSCG44-16-F-001007 and FAR 52.227-17. |
| Thomas M Kratzke, Strategic Data Systems, 25 July 2017 |

Table of Contents

[I. Introduction 0](#_Toc488743585)

[II. Reading and Writing 0](#_Toc488743586)

[III. Critical Details of new SORTIE tag 1](#_Toc488743587)

[IV. Critical Details of new SRU tag 1](#_Toc488743588)

[V. Basic Rules-Based Algorithm 2](#_Toc488743589)

[VI. Sprint 4-5 2](#_Toc488743590)

[VII. Sprint 6 3](#_Toc488743591)

[VIII. Sprint 7 3](#_Toc488743592)

[IX. Future Work 3](#_Toc488743593)

List of Tables

**No table of figures entries found.**

List of Figures

**No table of figures entries found.**

Roadmap to Transits in Optimizer

SAROPS Version 2.2

# Introduction

This document lays out an initial rough “target-path” of work. This is a very informal document that probably will not survive many sprints. We will use it primarily to get going. It represents my thoughts on how to get something done.

Surprisingly, we are not too concerned (yet) with VS or SS patterns. This note addresses how to read and write sequences of SRUs within Sorties *and* an approach to crafting a solution.

Right now, the code and xml use SRU to indicate an individual pattern to be optimized. That may cause confusion, but for backward compatibility, that is the term we use in this document. The entire *Sortie* is not a concept in Planner 2.1, so we feel free to introduce that term here; we will use all capitals for that since it will be a critical tag in the xml and, likewise, we will use all capitals for SRUs. It may be confusing that a SORTIE is a sequence of SRUs, but without major code re-factorization, we’re stuck with that.

The big goal is to have a collection of SORTIEs, and each has a sequence of SRUs. Only the SORTIE is guaranteed to have an “On-Scene-Endurance” (or *OSE*). The SRUs *might* have *OSE*s in the xml, but the SORTIE will always have one. An SRU’s endurance can either be prescribed in the SRU tag, or Planner will compute it. If it is prescribed, then the SRU’s endurance must be used for the transit from the launch point or the previous SRU in the sequence. If it is the last SRU, its endurance must also be used for the transit to the recovery point.

We will try to write the code so that old cases will continue to work.

# Reading and Writing

Judy might hate me for this, but I propose for the first task, a re-write of the xml. In the old Java structure, a PlannerModel had a collection of SRUs. I propose that in the new PlannerModel, there will be a collection of SORTIEs. Each SORTIE will have its own sequence of SRUs. This corresponds to the xml changing from:

1. <SRU…>
   1. Pattern information
2. </SRU…>
3. …
4. <SRU…>
   1. Pattern information
5. </SRU…>

to:

1. <SORTIE…>
   1. Sequence of SRUs as above
2. </SORTIE>
3. …
4. <SORTIE…>
   1. Sequence of SRUs as above
5. </ SORTIE >

If an SRU tag appears where a SORTIE tag is expected, Planner will automatically generate a SORTIE tag and put the singleton SRU in it.

# Critical Details of new SORTIE tag

The most critical difference will be that it will have an On-Scene-Endurance attribute. It might have a single launch latLng and a single recovery latLng as well. But it must have On-Scene-Endurance.

I will assume that we are not tracking fuel, only time. Hence, On-Scene-Endurance’s units is minutes. If this is correct, then it can have a transit speed, but each child SRU must provide its own search speed and, optionally, its own On-Scene-Endurance.

We will interpret a SORTIE’s On-Scene-Endurance to be an upper bound for its child SRUs’ On-Scene-Endurances plus the transit times.

# Critical Details of new SRU tag

The new SRU tag will look much like the old SRU tag. It will have an additional attribute patternType. This will default to PSCS, but (the only) other possibilities are VS and SS. For the first few sprints, only PSCS will be allowed.

These SRU tags will be ordered; the order that Judy writes them out within the SORTIE is the sequence.

Each SRU tag can have a pattern specification as is the case in Planner 2.1. However, if some SRU tag does *not* have one, Planner will ignore subsequent SRUs’ pattern specifications; I thought about applying the same thinking to SRUs’ On-Scene-Endurance, but I don’t think that’s necessary. We may even loosen this requirement for pattern specifications.

# Basic Rules-Based Algorithm

As in all algorithms in Planner 2.1, we will place SRUs one at a time. This is far more difficult since we are missing the pattern’s length as an input. We will simply make one up and try a few nearby ones. Alternatively, we could use a random input for this.

Assume that we have placed 5 SRUs and have 8 to go. Each of the remaining SRUs has an earliest possible commence search time. The next one we place (i.e, the 1st of the remaining 8) must have its predecessor SRU (if any) placed and, among those, can start earliest. Ties are broken randomly.

The above is a key concept to a workable algorithm so I will leave it in its own paragraph and spend another paragraph (this one) pointing out its importance.

If we knew how long we would take on the pattern, most of this could take place with Planner 2.1 code, we would have almost reduced this problem to Planner 2.1. Alas, we don’t know the On-Scene-Endurance. Instead, we will estimate it.

We will take the remaining On-Scene-Endurance for the SORTIE, subtract off some multiple of the straight-line distance to the recovery point, and allocate evenly the remaining On-Scene-Endurance among the remaining SRUs in this Sortie to place. This rough estimate will be adjusted in subsequent group discussions. The multiple could decay from say, 2.0 to 1.0, as we have fewer and fewer SRUs to place within this SORTIE.

In the Planner 2.1 of an SRU, we try several different configurations (this is *not* the optimization part; the placement part tries different angles). In the same way, we could try different values for allocated time, and impose a small per-hour penalty on the resulting POS for time-above suggested allocated time. In this way, we would use larger times when warranted, and save time for later SRUs.

The allocated time must include the transit from either the launch point or the previous end point. If it is the last SRU in the SORTIE, it must include the transit to the recovery point as well. This will be used as an upper bound so that first a proposed pattern is constructed and then it will be checked for feasibility.

# Sprint 4-5

This may be too ambitious, but I would like to get everything discussed so far done in the next two Sprints. In the first sprint, we would adjust the xml. Judy and I would have to coordinate this, but the “patch jar-file mechanism” would make this possible.

Sprint 5 might involve others agreeing on the Basic Rules Based Algorithm above. In Sprint 4, I will start developing a straw-man procedure as a catalyst for others’ ideas, but we will need (e.g.) Jack and Robert agreeing on a basic On-Scene-Endurance allocation.

This would be limited to PSCS patterns.

# Sprint 6

Introduce VS patterns. We need the rules-based individual SRU code as well as the iteration part for VS. Start working on SS patterns

# Sprint 7

Finish SS patterns

# Future Work

Get away from perfect patterns except for VS patterns.

If we do this, we might be able to use NlOpt (<http://ab-initio.mit.edu/wiki/index.php/NLopt>). An interesting note is that in Planner 2.1, we re-cast the overlap constraints to be more amenable to this. The overlap constraints not only measure how badly overlapped an illegal set of SRUs is, but it also computes how close to being illegal *legal* sets are.

The methods of NlOpt date from 1953 or earlier.

Consider Amazon Web Services (AWS) to generate a lot of possible starting points. Planner 2.1 made this possible by moving to basing SRU initial solutions on a small random number of randomly selected particles. Using AWS could allow thousands of initial placements with refinements. Take the best one, refine it, and we’re done.

This is not nearly as sophisticated as NlOpt; it is simply using expanded computer power to generate a huge brute-force collection of possible solutions