Discrete Event Simulation for Space Mission Logistics

Benjamin A. Merrel
Georgia Institute of Technology, Atlanta, GA, 30332, USA

David B. Gomez
Georgia Institute of Technology, Atlanta, GA, 30332, USA

Edmund H. Chen
Georgia Institute of Technology, Atlanta, GA, 30332, USA

Team Number: 35

GitHub Repository
`https://github.gatech.edu/SpaceMissionDES/SpaceMissionDES`

Abstract

Space mission logistics can be a complex system of vehicles, activities, and schedules. Discrete Event Simulation (DES) is one tool that space mission designers sometimes use to predict key metrics for various space missions, such as duration, cost, and probability of success. In this work, we leverage DES to compare different strategies for a crewed mission to Mars. The general mission architecture is the Split Mars Mission Concept from the Evolvable Mars Campaign. This mission involves a series of (pre-crew) cargo only launches to Mars, followed by a series of crew launches to first lunar orbit, then to Mars. Our DES model will be used to simulate this mission architecture and allow for flexibility in several key processes, such as the number of cargo / crew launches and conditional re-attempt procedures. The goal of our project is to demonstrate the utility of DES for space mission design, specifically in comparing alternative in-space architectures for the same space mission.

I. Project Description

Our project's ultimate goal is to utilize Discrete Event Simulation (DES) to model a mission concept for getting humans to Mars. In progressing towards this goal, we found it useful to develop proof-of-concept models for simpler space missions. This report thus describes the development and application of DES for thgree missions: the Mars Transfer Vehicle Mission, the Two Vehicle Merge Mission, and the Mars Cargo/Crew Split Mission. In the next subsections, we describe the mission and identify the key elements that are most relevant to capture in our models.

Please note the culminating work of this project is the Mars Split Mission, in which the concepts and paradigms designed and tested with the prior two case studies are put to test in a much larger and complex scale.

A. Case Study 1: On-Orbit Propellant Aggregation

This mission involves sending a Mars Transfer Vehicle (MTV) into a parking orbit around Earth, then sending several "Tanker" vehicles to rendezvous with the MTV and transfer propellant. When the MTV is full, it departs for Mars. Each class of entities has a different concept of operations (ConOps). The MTV launches to parking orbit and waits for the Tankers to fill its tank; the Tankers on the other hand, launch, rendezvous with the MTV, and transfer their propellant. Thus, this mission involves multiple entities with different ConOps.

B. Case Study 2: Two Vehicle Merge Mission

This mission involves a Moonship and a Tanker which have independent launch sequences and merge to form an aggregate vehicle. The purpose of this case study is to primarily demonstrate the abilities of the aggregation and nesting abilities. With a generalized model, this is especially helpful to extrapolate to more complex missions later on.

C. Case Study 3: Mars Split Mission

The mission is based of the Evolvable Mars Campaign's proposed Split Mission Concept. This mission involves a series of (pre-crew) cargo-only launches to Mars, followed later by the crew. The key elements in the mission that are the most relevant to our model include (1) the launch vehicles, (2) the transit habitat, (3) the propulsion systems (both chemical and electric), and (4) the Mars lander systems.

The mission involves a series of cargo-only launches on board the Space Launch System (SLS) to Low Earth Orbit (LEO), followed by a Solar Electric Propulsion (SEP) transit to Mars. The cargo mission contains systems for Mars operations as well as propulsion system to return crew to Earth. The cargo missions are followed by series of launches transporting the crew and their habitat to lunar orbit. The crewed habitat then departs from lunar orbit to Mars using chemical propulsion. After the Mars surface operations, the crew depart from Mars and are returned to Earth. This mission architecture is the final subject of our DES model.

II. Literature Review

Space mission analysis deals with assembling various vehicles and concepts into a coherent system that works together to accomplish a set of mission objectives. During the early phases of conceptual design, mission planners often have freedom to choose between vehicles and concepts of operations which may vary greatly in scale and timeline. To select the best option from a suite of alternatives, designers often consider system-level metrics such as total number of launch space launches or total system mass. A great deal of work, tracing back before the dawn of the space age, has been devoted to developing sophisticated methods for estimating the size, weight, and performance of conceptual spacecraft. Such methods ensure that the early phase of design can set a program up for success. In contrast, the analysis of mission schedule and risk at the conceptual design phase has received little treatment in the literature, prior to the early 2000s.

As the shuttle program matured in the 80s and 90s, the challenges with operating complex space mission motivated researchers to develop methods for estimating reliability and maintainability during conceptual design. In particular, Ebeling and Donohue developed a series of modules used for predicting the number of missions which could be accomplished by future shuttle variants [1]. Their work relied on applying multiple linear regression to historical data on relevant processes, and then summing the time estimates for all of the required processes. Later, when NASA began exploring an increased flight rate for the schedule, discrete event simulation (DES) was proposed to enable more granular control over the specific parameters and sequences of events. Cates and Mollaghesemi began a years-long development of increasingly sophisticated DES to approximate space shuttle processing [2].

Discrete event simulation (DES) can be described as a process towards simulating a discrete sequence of events, a situation where changes happen exclusively at certain discrete points in time. This is in contrast to continuous simulation, where the process in question is modeled over a continuous set of times. By only marking discrete state changes in the model at hand, DES allows simulations to jump directly from one event to the next, allowing for efficient simulation of complex time-dependent processes [3]. As such, DES simulation is often used in many applications of operations research, scheduling problems, and risk analysis. One application of DES concerns decision making, where the crux of the model is to stipulate when and which certain events should happen. Within healthcare, this is widely used to model disease progression and various health behaviors, as outlined in Zhang's 2018 paper [4]. On the other hand, DES can be also used for risk analysis by coupling together the probabilities associated with different steps in a sequential process. Such applications are especially helpful in manufacturing, where complex supply chains with multiple dependencies can be modeled to capture where critical risks lie [5].

For DES processes concerning some determination aspect, much work has been done to investigate the different queuing and optimization methods to determine best policies for the application in question. For instance, in network modeling, DES models can simulate different congestion protocols when sending packets over some network link [6]. Many tools have been developed for discrete event modeling of computer networks. One such framework is ns-3, which provides a flexible platform to pen-test different protocols used in wireless networks as described by Stojmenovic [7]. Ancillary tools to model different mobility behaviors of real-life applications can also be utilized to better gauge the performance of such protocols [8]. An application of implementing such network queuing models can be seen in Yousefpour et al., where a fog-offloading policy is developed to determine which jobs should be allocated between the fog and cloud layers [9]. Different queuing policies can be designed to optimize for different metrics; for instance in certain situations an earliest deadline may be observed, whereas in others, a first come first serve policy may be more paramount.

DES models continued to be utilized throughout the early 2000s for space shuttle launch logistics and the assembly of the International Space Station (ISS). In 2002, Cates and Rabadi created a DES model to analyze the transportation and integration logistics of the highly complex space shuttle system and used it to predict process scheduling, storage, and overall project strategy [2]. In 2005, Cates and Mollaghasemi created a DES model to analyze the proposed assembly schedule of the ISS and used it to predict the estimated completion date of the ISS as a function of space shuttle launch frequency [10]. This model continued to evolve to incorporate changing timelines imposed by new administrations [11]. In 2006, Cates et al. created a DES model to analyze NASA's proposed Low Earth Orbit (LEO) rendezvous strategy for returning humans the the moon and to explore the ramifications of such a strategy [12]. Their model includes ground logistics and various uncertainties surrounding the pre-launch operations.

After successfully demonstrating the utility of DES to existing space systems and operations, DES practitioners began to explore bringing their tooling to the world of conceptual space mission design. This new area of application is demonstrated by Cirillo, Stromgren, and Cates in their 2010 paper on applications of DES for space mission risk assessment [13]. In the paper, Cirillo et al. showed that a DES model can be used to capture the schedule and risk variability of both ground and in-space events. They showed that by bringing DES into the conceptual phase, designers can perform trade studies to show how the selection of alternative vehicle types affects the overall mission duration and success probability. Over the next several years, Cates, Cirillo, and their collaborators continued to build on that work by adding additional types of events and architectures. A 2012 paper introduced events for on-orbit assembly and crewed vehicle operations [14]. In 2013, the same authors continued to advance their modeling capability with Mars mission-specific events and data dashboard for interrogating results [15]. Then in 2014, Cates et al. demonstrated that the space mission DES approach could be leveraged to compare architectures which rely on alternative sets of launch vehicles [16]. Comparisons were made on the basis of campaign duration and reliability for the launch and assembly segments of the campaign. Finally, papers from 2016 and 2022 demonstrate further advancements in the model to include individual entity routing and analysis of evolving space exploration architectures [17], [18].

From another perspective, DES can also be a valuable approach in modeling resource allocation when considered from a commodity use first standpoint. For instance, in Leonard, Parsons, and Cates demonstrate in their 2014 paper a model which investigated the commodity use of hydrogen, oxygen, nitrogen, and helium in the SLS, stipulating how fast a turnaround could happen in replacing such commodities [19]. A similar approach to the manufacturing risk analysis seen in [5] has also been used in the risk modeling of ground operations as seen in [20], simulating the operational processes leading up to space launch and extrapolating relevant contingency analyses.

The previous applications of DES to space mission design share three common themes. First, previous work has largely focused on ground logistics and pre-launch processing. Second, previous work has usually been applied to an already established, proposed mission architecture. And third, previous DES models were usually used for predicting various mission outputs, such as absolute cost, duration, and risk. In this work, we use DES to conduct a comparative analysis of different space mission architectures for the same space mission. We will focus more on the in-space concept of operations (ConOps) and less so on ground operations. And we will avoid the ambiguity of absolute predictions in favor of a comparative analysis approach. We intend to demonstrate the utility of this approach for comparing alternative in-space architectures for the same mission.

III. Conceptual Model

Both missions described in Section I can be conceptualized as collection of vehicles which proceed through a sequence of events and activities. The subjects of our DES models are the vehicles and other transportation systems. These entities will proceed through a variety of activities which begin and end with an event. The activities includes items such as ascent, transit, descent, and rendezvous, while the events include items such as liftoff and various orbital maneuver burns. These will be discussed in more detail in the following section.

One of the main considerations we took into account was how to define the worldview of our discrete event simulation. As many space processes are complex, an event such as liftoff could be broken down into many constitutive events, and get progressively granular. Since a part of our project may constitute a risk analysis, we sought to strike a balance between using a granularity that could stipulate a meaningful model while not running into complexity issues. Furthermore, we considered the distinction between event-centric and activity-centric modeling, for instance, the difference between taking into account the events of liftoff start and liftoff end as opposed to the activity of liftoff. In our current scope, we utilize both to signify both noteworthy events and activities. We also sought to introduce dependencies, so that we may establish failures of an event that were not fatal to the overall mission, perhaps just inducing a greater resource usage on a successive event or invoking another retry event. Hence, resource allocation then also becomes a valuable part to model, tracking how much of each resource is used in each stage and stipulating limits on how much a certain activity may be retried. Finally, we considered queues, where a certain number of events need to happen, but the script is not provided a strict ordering with which they need to be executed. Given either a dependency order or a queuing strategy, it would then process the queue in the order it saw fit.

IV. Simulator Model

A. Overview

The main model operates on the implemented conceptualizations of three key elements: vehicles, events, and activities. These are then further implemented upon by an accessory of advanced functionalities, such as predicates and resource tracking. In the end, we consider a generalized framework to better describe the complex simulations at hand. Here, we first describe these main paradigms, then outline the simulation flow and routine, and lastly go over the advanced functionalities, Critically, subsection **??** refers to the generalized orientation of the simulation framework.

1. Vehicles

The vehicles can be instantiated as any entity that moves in space. For example, the vehicles for the Mars Transfer Mission are the MTV and the Tankers. For the Mars Split Mission, a slightly more nuance approached was required. Because the Split Mission requires several rendezvous and vehicle separations, we needed to develop the infrastructure to aggregate / separate multiple vehicles that each proceed through their respective ConOps. To develop this capability, in our simulation, a vehicle can be a single entity or an aggregation of multiple entities. For example, the launch vehicle for the Split Mission begins as one assembly: the launch vehicle and the payload. Since the SLS is a two-stage launch vehicle, their will be a staging event in which the SLS separates from the the payload. After this separation the payload can proceed with the remainder of its mission. Developing this aggregate vehicle capability was a major challenge in this project. Thus, in lieu of performing a suite of experiments, as was done for the simpler Mars Transfer Mission, our objective for the Split Mission was to develop the aggregate vehicle capability and to demonstrate its successful application.

2. Events

The events are pseudo-instantaneous events that change the state variables and generally delineate time within the simulation. In our model, events represent items such as liftoff, separation, rendezvous, transfer / capture burns, landing, etc. A vehicle proceeds through a sequence of events (its ConOps) until it reaches some terminal state.

3. Activities

We define the activities to be the processes that a vehicle undergoes between events. For example, the activity that occurs between the events, liftoff and separation, is ascent. Likewise, the activity that occurs between transfer burn and capture burn is transit. Sometimes, an activity should not be processed until some condition is met. We developed this capability in the form of Predicated Activities. These activities are essentially in a waiting phase until its predicate (condition) is satisfied. For example, in the Split Mission, several vehicles need to rendezvous together in lunar orbit before the transfer burn which takes them to Mars. The transfer burn in this case, would be a predicated activity that does not execute until after the other components have rendezvoused together. If a given activity fails, there are contingency activities that may be defined to execute accordingly.

B. Simulator Architecture

We begin by introducing a set of classes which enable the user to construct a concept of operations (ConOps) that defines how entities in the simulation are expected to behave. A ConOps is defined by listing a sequence of activities which each begin and end with an event. First, we define a class called `Event` which contains a single field to store a recognizable name. Instances of this simple `Event` class are used to define a ConOps, but they are not assigned a time or any other scheduling information. Instead, they serve as a template for a `ScheduledEvent`. Next, we define a class called `Activity` to represent the processes defined by a given ConOps. Each `Activity` is given a recognizable name, a starting `Event`, an ending `Event`, and a duration. Note that this duration should be considered a baseline value, as some versions of the simulation will allow for duration to vary according to a statistical distribution.

Given a set of activities and events, we can define a ConOps. The figure shows the sequence of activities and events required for a rocket vehicle to launch to orbit. Notice that there is a specific order of the activities, and each event serves as the end of one activity, and the beginning of the next. We collect all of this into a data class called `ConOps` which contains a sequence of activities and a mechanism for selecting the next activity based on an occurring event. Python code to define and query a `ConOps` is shown below.

After defining a `ConOps` and instantiating it with a specific sequence of `Event` and `Activity` objects, we define a vehicle which will "fly" it. We define a `Vehicle` class as an object with has a name, some collection of states, a current activity, and a mechanism for tracking its history of events, activities, and states. A key feature is that an instantiated vehicle always has an associate activity. The `Vehicle` instances interact with `Activity` instances and when the ending `Event` for an activity occurs, the `Vehicle` queries the ConOps for the next activity, and schedules the corresponding events. In addition, a vehicle can have states such as current propellant mass or location, and the code for handing the end of an activity includes logic for updating the states according to the activity. For example, when an activity for a rocket firing completes, the vehicle updates its propellant load based on activity.

As previously mentioned, the basic `Event` class does not include information about the conditions required to trigger it. For that, we implement a related class called `ScheduledEvent`. This more detailed class has a name, a reference to the template `Event` a time at which it will occur, and a reference to the vehicle that scheduled it. Note that `ScheduledEvent` instances enable time-based conditions, but other types of conditions such as the success of specific other events much be address by future work.

With a given simulation, there may be many `ScheduledEvent` instances with different times and associated vehicles. A fundamental feature or the DES machinery is the ability to select the next event to occur based on the expected event times. We implement this capability by created a class called `FutureEventList` which stores all scheduled events and has the capability of selected the event with the lowest time value. Specifically, we leverage the python class heapq which implements a priority-based queue in which priority is given to the the lowest value of occurrence time. The specific implementation is based on an example provided in class.

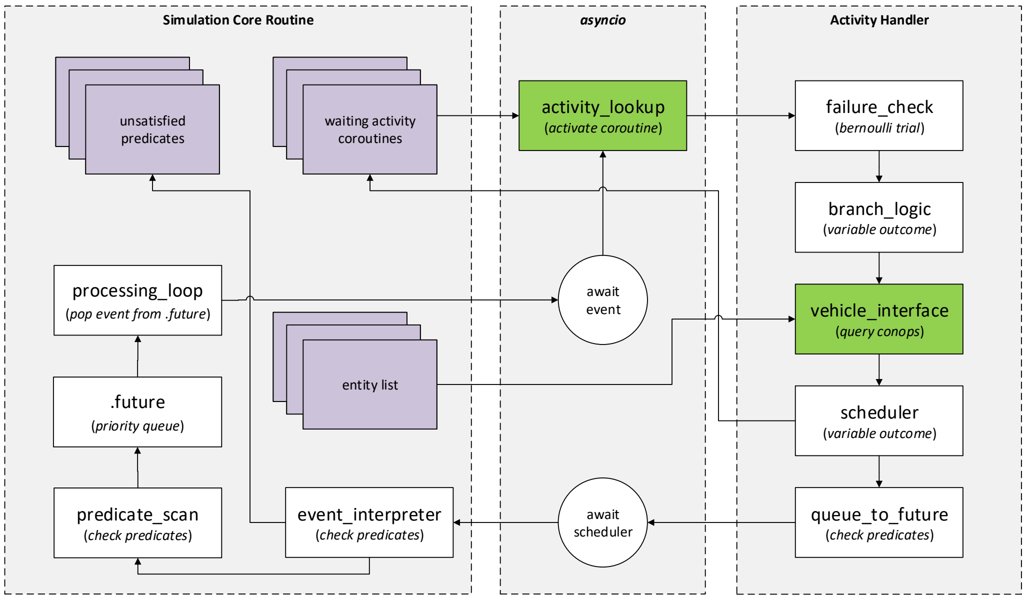


Fig. 1 DES Architecture

The final simulation architecture can be seen above. First, an input file with the relevant vehicles and ConOps is fed to the simulator. The simulator populates a priority queue with the timing of the immediate activities in the various initial vehicle ConOps, and calls an activity handler which runs through the various checks. The activity handler first checks if the activity has failed, whereupon it runs a contingency on the fail case if one is provided. If not, the activity is passed to the vehicle where the vehicle instantiation can tell the simulator the next activity in the queue. This activity is then populated into the general simulator queue, disassociated from the vehicle instantiation. Within this activity handler is also an aggregation handler that can instantiate new vehicles within simulation if so desired. This process repeats until there are no more activities in the main simulation queue. The special type of activity, a predicate activity, will provide a function that checks for the condition of interest, which is then passed to the main simulation core routine. Every successive timestep, the simulation will then run the predicates to check if any have been satisfied. If one has, it is then scheduled for the timestamp the simulator is currently on, and discarded from the unsatisfied predicates list. This repeats until the simulation lapses in entirety. The following subsections go into more detail on each of the advanced functions of our simulation framework.

C. Predicate Checks

As expected, some activities in this simulation will not be strictly sequential. That is to say that some activities depend on another vehicle completing some event rather than the activity that immediately precedes it. In these scenarios, we introduce the notion of a predicate event. These predicate events do not maintain a duration or a probability of failure, rather they are treated as "if" conditions for the simulation to proceed. We introduce utility functions that can be systematically input into our simulation so such conditions can be checked. This flexibility gives us the freedom to define any number of predicate checks that we may want, for instance with the custom functions we are able to access the full scope of every entity in the simulation. Therefore, we could hypothetically check that some vehicle has completed a certain activity, while another vehicle has resource above a certain value. However, for the more common predicate checks, namely that a certain vehicle has completed a certain event, we use a functools *partial* functionality to create a template for creating multiple predicates. As predicate events function essentially as an "if" condition without duration or probability of failure, the *Completer* event to any given vehicle cannot be directly attached to the next event in a Predicate activity. We can see this illustrated in 2, where Vehicle A's Activity A2 cannot execute until Vehicle B has also executed Activity B2. In this scenario, there would a predicated activity implicated before the starting event of Activity A2 whose condition is the end event of Activity B2.

These predicates are then tied to specific schedule events, which are in turn tied to vehicles. Critically, the predicate event is not existing in only the scope of the vehicle it is concerned with. Since we need to check this predicate in the main simulation loop on every successive timestep, we have to disassociate it from the vehicle it originates from as the predicate will be satisfied by a vehicle other than the originating vehicle. In this sense, we keep a running data structure of the predicates in the core routine of the simulation and iteratively check all of them in each successive event. If a predicate is found to be satisfied, then it is marked to start at the current simulation time, and the next iteration of the simulation core routine will handle it as expected. Just like any other activity, this will then lead to a successive activity. Critically, as predicate events do not have duration or probability of failure, they cannot end in a terminal *Completer* event, there must be at least one successive activity that follows it.

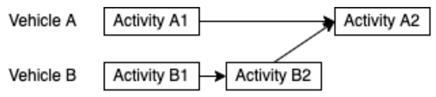


Fig. 2 Predicate Logic Diagram

Implementation-wise this would be realized within the *ConOps* of the respective vehicle.

```python
moonship_conops = ConOps({
    ...
    meco.name: Activity("Circularize", meco, begin_loiter, duration = 10),
    begin_loiter.name: PredicatedActivity("WaitForProp", begin_loiter, prop_full, predicate =
    vehicle_in_activity("tanker", "docked")),
    prop_full.name: Activity("Checkout", prop_full, tli_burn, duration = 110),
    ...
})
```

Listing 1 Predicate Declaration Example

The template predicate declaration referred to by vehicle_in_activity is shown by

```python
def check_func(p, sim, vehicle, activity):
    veh = sim.entities[vehicle]
    if veh.activity.name == activity:
        logging.info(f"Predictate <{p.predicate.name}> Satisfied")
        return True
    else:
        return False

def vehicle_in_activity(vehicle: str, activity: str):
    return partial(check_func, vehicle = vehicle, activity = activity)
```

Listing 2 Predicate Template

which in the main simulation loop shows up as a predicate check prescribing the following.

Algorithm 1 Predicate Check Algorithm

```
Require: Predicates Exist
  for Every Predicate do
      if Predicate is satisfied then
          Schedule Predicate at Current Time
          Remove Predicate
      else
          continue
      end if
  end for
```

D. Generalized Object Model

1. Overview

Naturally, space missions are very complex processes that stipulate a wide array of possibility in mission structure. In this sense, having a rigid structure to simulator object is untenable, namely, such a simulation framework will be unable to realize complex missions that may have multiple levels of hierarchy. Towards this end, we wanted to assure our simulation framework was generalized enough so that it could realize a wide array of missions flows. In concrete terms, this meant that we needed to establish a framework that could be adaptable enough to suit any number of mission parameters, including those that may need to track certain resources in specific manners, and nest multiple objects within each other. To accomplish this goal, we made the core component of the simulation, a Vehicle, to be a generalizable object. That is to say, the Vehicle object is able to have parents and children objects, whereupon operations to add, drop, join, or disperse children Vehicles can be called upon.

To start off, we must consider that such an object needs to be self-contained and nestable. That is to say, a Vehicle instatiation must be able to be nested within another Vehicle instantiation without having other modifications to its class structure. We accomplish this first by introducing child and parent class variables to the Vehicle class. As these are stored as strings, this stipulates that all entities within the simulation must have unique names. With these class variables, declared vehicles can be directly defined in a hierarchy within the simulation input. Furthermore, this stipulates no maximum bound of the nested structure of the simulation, namely, we can have as many successively nested Vehicles as we desire. To visualize this, figure 5 shows a doubly-nested hierarchy of Vehicles, where aggregate Vehicle A1 has Vehicle A and Vehicle B under it, and aggregate Vehicle A2 has Vehicle C and aggregate Vehicle A1 under it.

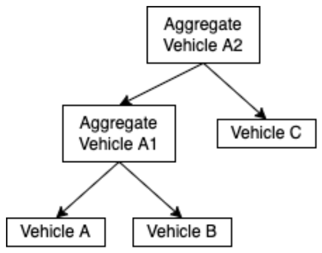


Fig. 3 Aggregation Conceptual Diagram

2. Aggregation Events

To then modify this structure, we introduce four different aggregate functions and their appropriate parameters towards this goal. Firstly, and the most commonly used, is the *join* aggregation type. The join aggregation type is used when we have multiple vehicles which we wish to join into a newly instantiated Vehicle, which was previously not an active entity in the simulation. In other words, this provides a way to initialize an entity during the simulation that does not exist upon the initial inputs towards the simulation. To successfully create such a join operation, we then require three different parameters. Namely, in consistency with the other simulation objects, we firstly require a ConOps that has be pre-initialized to describe the events the aggregated entity undertakes; next, we require a name for the aggregated entity, and lastly, we require a list of vehicles that are to be joined into this aggregate entity as children. Note that since in complex missions we may not have the full ConOps realized during the aggregation event call, a temporary reference is first inserted and then substituted later on in the input file. As seen below, these parameters are directly input into the ConOps, and a Predicated Activity is shown to pick up the operations after the aggregate vehicle has disbanded.

```
moonship_conops = ConOps({
    # Nominal
    ...
    dock.name: Activity("Docking", dock, almost_arrive, duration = 10, p_fail = 1/10, agg_type="
    join", agg_params={"conops": aggregate_conops, "vehicles": ["Tanker", "MoonShip"], "name": "
    aggregate"}),
    almost_arrive.name: PredicatedActivity("spin", almost_arrive, finale, p_fail = 1/10,
    predicate = Predicate("aggregate activities finished", vehicle_in_activity("aggregate", "
    FINISH"))),
    finale.name: Activity("Finale", finale, ARRIVE, duration = 2, p_fail = 1/100)
})
```

Listing 3   Aggregation Join Activity Example

In a logical continuation, we then have the *dejoin* aggregation event. A *dejoin* aggregation event drops all children vehicles of the given parent vehicles, and updates that information accordingly on both the parent vehicle and the child vehicles(s) class variables. This is helpful in eliminating the need to have multiple *dropchild* aggregation events, and also provides a way to drop a dynamic amount of children. Where this scenario may arise is that we have some entity which is docking with a dynamic amount of tankers depending on how many successfully make it to the end goal. This *dejoin* aggregation event likewise does not accept any parameters.

Now in the other aggregation scenario, say we seek to add a child to existing vehicle, rather than create a new aggreagtion vehicle for two vehicles which exist in parallel, which is the case for *join*. In this scenario, we then introduce an aggregation event called *addchild*. This quite straightforwardly, adds a child to the vehicle it was called upon and updates the class variable accordingly. In general, it is useful to have the distinction between a parallel join and a nested child object if there is a clear primary vehicle in the aggregation of the multiple vehicles. For instance, if it is a resupply ship along with a main spacecraft, then it makes more sense to treat the resupply ship as child vehicle of the main spacecraft, rather than put both the resupply ship and main spacecraft into a new aggregated vehicle. Likewise, we have the *dropchild* aggregation event that does the opposite, in that it drops the child vehicle and updates the information accordingly on both the parent and child vehicle class variables. In the aforemetioned scenario this would be invoked when we want to remove the resupply ship from the aggregate vehicle's child list. All in all, the class structure and parameters behind such activities would look like the following.

```
@dataclass
class Activity:
    name: str
    start: Event
    end: Event
    duration: float
    p_fail: float = 0    # probability that the activity will fail to be completed (i.e. failed
    launch)
    failure: Event = Failure()
    resource_change: dict = field(default_factory = dict)
    agg_type: str = ""   # either join, dejoin, dropchild, addchild
    agg_params: dict = field(default_factory = dict)
    # dict of agg params
    #   FOR join {conops: conops, vehicles: [va, vb], name: "va-vb"}
    #   FOR dejoin N/A
    #   FOR addchild {vehicles: [va, vb]}
    #   FOR dropchild {vehicles: [va, vb]}
```

Listing 4   Aggregation Activity Parameters

3. Aggregation Simulation Flow

Lastly, we must discuss the simulation flow that dictates the execution of such aggregates. Namely, we assume the input file is aware of all events and no duplicate aggregation activities are given, though this is also made to raise an exception on the execution side of things. In this sense, the aggregation activity needs only to be called by one constituent vehicle. Namely, if we want to join *Vehicle A* and *Vehicle B* into *Aggregate 1*, only one of the two vehicles must call the aggregation event. After this is called, the ConOps of the independent vehicles can undertake two different paths.

Firstly, if the child vehicle will not be used independent of the aggregate vehicle in any point during the simulation, then it can be marked as completed. It is critical that this means the child vehicle will not have any other outstanding events as completed vehicles in our simulation framework will never resurface in the simulation. If this is the case, a *Completor* event can be given which will effectively mark the child vehicle as an object no longer of interest to the simulation, and it will soley exist as a child entity of the aggregate vehicle. On the other hand, the child vehicle can continue a ConOps independent of it's parent vehicle while inside the aggregate. For instance, if we have a crew module that is docked to a transport vehicle, then we perhaps need the crew to do an event inside the crew module independent from the transport vehicle or the aggregate of the transport and crew vehicle. In this case, this event would solely exist under the crew module's ConOps, as it would deal with neither the transport vehicle or the aggregation.

Likewise, these events would then most likely take into account some dependency. For instance, if we wanted the crew vehicle in the aforementioned example to only undertake such an event after the docking has been completed, we can introduce a *PredicatedActivity* on the condition that the aggregated vehicle has completed it's docking activity. This logically holds similarly for child vehicles that may have processes to do after decoupling from the aggregate vehicle. For instance, consider a situation where a spacecraft docks with a refueling tanker, refuels, and then splits off from the refueling tanker to continue its mission. In this case, after the docking of the two vehicles and the fuel transfer has been successful, we need to decouple the refueling tanker and the spacecraft from their aggregate, and send both on their separate ways - the fuel tanker will likely go towards a deorbit activity while the spacecraft will continue upon it's scheduled trajectory. In this scenario, we use the predicated activity to check that the aggregated vehicle has lapsed it's *Completor* event, namely that the aggregate entity has been decoupled and no longer exists as far as we are concerned. In this sense, each of the constituent vehicles is handed back "control" of their activities from the aggregated entity, and can go about their ConOps as defined. Critically, the difference between these two end scenarios is whether the child vehicle will still have activities to do that are independent of the aggregate vehicle. If there are, it must maintain a ConOps going past the existence of the aggregation, while if it is not it can safely Complete. To visualize this, we see the *join* aggregation event execution in the *activity_handler*

```python
async def activity_handler(name: str, start: ScheduledEvent, sim: Simulator, vehicle: Vehicle):
    ...
    #event execution code omitted for sake of brevity
    if not isinstance(current_end, Failure):
        if current_activity.agg_type == "join":
            logging.info(f"\t  VEHICLES {current_activity.agg_params['vehicles']} > Begin ACTIVITY:  {current_activity.name}")
            logging.info(f"\t  VEHICLES {current_activity.agg_params['vehicles']} > JOINED TO:  {current_activity.agg_params['name']}")
            if len(current_activity.agg_params['vehicles']) < 2:
                raise Exception("Not enough arguments provided for object collation")
            resources_agg = Counter({})
            for vc in current_activity.agg_params['vehicles']:
                resources_agg += Counter(sim.entities[vc].resource)

            parent_vc = Vehicle(current_activity.agg_params['name'], current_activity.agg_params['conops'], dict(resources_agg), children = current_activity.agg_params['vehicles'][:])

            for vc in current_activity.agg_params['vehicles']:
                sim.entities[vc].parent = current_activity.agg_params['name']

            sim.add_vehicle(parent_vc, sim.clock + current_activity.duration)
```

Listing 5 Join Aggregation Event Logic

As can be seen, the simulation aggregates the resources of the children, checks that everything is sufficient, creates the aggregated vehicle, and then appends it to the simulation entities so the aggregated vehicle's ConOps will be executed. Likewise, the overall simulation routine that deals with aggregation events can be seen as follows.

Algorithm 2 Aggregation Handling Algorithm

```
Inside Activity Handler
for Every Aggregation Type do
    if Aggregation Type is Selected then
        if Does Not Violate Hierarchy and Sufficient Parameters are Provided then
            Execute Aggregation Operation and Modify Simulation Entities
            Break out of Loop
        end if
    end if
end for
```

4. Usage and Multiply-Nested Vehicles

In this sense, as long as we are correctly affirming that no child vehicle is simultaneously an ancestor to two different aggregate vehicles, we are able to design as many hierarchy packs as we want. We can have later stages which are bound to separate initialized as child vehicles on their parents, we can have as many flatpacked children vehicles to one parent, and all the other scenarios that come with such a functionality. Note that this also implies that we can have as many levels of hierarchy as desired. For instance, if there is a one-level hierarchy parent vehicle that then needs to be attached to another vehicle, the aggregate of those will be a two-level hierarchy parent vehicle. This is especially useful in the full Mars case study that is the culminating goal of this project. As with such complex simulations, there will be many different vehicles that are joining and dejoining over the course of the mission. The implementation of this generalizable functionality aids that simulation in that we can then have these multiple hierarchy structures.

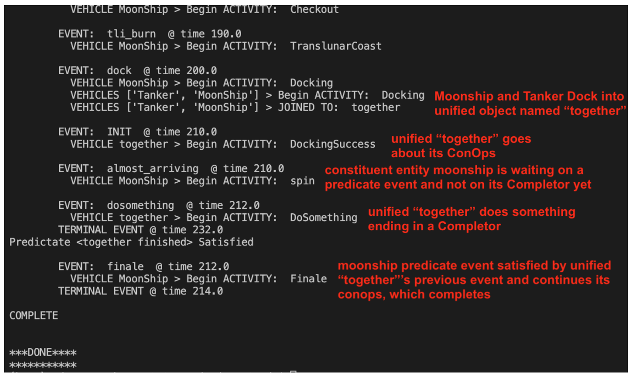
5. Proof of Concept Example

To demonstrate the ability of the aggregation functionality, we consider a scenario where we have a MoonShip and Tanker vehicle launching separately, and then joining as an aggregate vehicle once both launched.

```
moonship_conops = ConOps({
    # Nominal
    INIT.name: Activity("Countdown", INIT, liftoff,  duration = 10),
    liftoff.name: Activity("Ascent", liftoff, meco, duration = 10),
    meco.name: Activity("Circularize", meco, begin_loiter, duration = 10),
    begin_loiter.name: PredicatedActivity("WaitForProp", begin_loiter, prop_full, predicate =
    tanker_prop_transfered),
    prop_full.name: Activity("Checkout", prop_full, tli_burn, duration = 110),
    tli_burn.name: Activity("TranslunarCoast", tli_burn, dock, duration = 10, p_fail=1/10),
    dock.name: Activity("Docking", dock, almost_arrive, duration = 10, p_fail = 1/10, agg_type="
    join", agg_params={"conops": together_conops, "vehicles": ["Tanker", "MoonShip"], "name": "
    together"}),
    almost_arrive.name: PredicatedActivity("spin", almost_arrive, finale, p_fail = 1/10,
    predicate = Predicate("together finished", vehicle_in_activity("together", "DoSomething"))),
    finale.name: Activity("Finale", finale, ARRIVE, duration = 2, p_fail = 1/100)
})

tanker_conops = ConOps({
    # Nominal
    INIT.name: PredicatedActivity("WaitForMoonship", INIT, begin_countdown, predicate=
    moonship_predeployed),

    begin_countdown.name: Activity("Countdown", begin_countdown, liftoff,  duration = 10),
    liftoff.name:         Activity("Ascent", liftoff, meco, duration = 10, p_fail = 1/10, failure
     = get_spare),
    meco.name:            Activity("Rendezvous", meco, final_approach, duration = 10, p_fail =
    1/10),
    final_approach.name:  Activity("Docking", final_approach, dock, duration = 10),
    dock.name:            Activity("PropTransfer", dock, undock, duration=10, p_fail = 2/10),
    undock.name:          Activity("Disposal", undock, DONE, duration=10, p_fail = 1/10),

    # Contingency
    get_spare.name: Activity("PrepSpare", get_spare, begin_countdown, duration = 100)
})

together_conops = ConOps({
    INIT.name: Activity("DockingSuccess", INIT, dosomething, duration = 2),
    dosomething.name: Activity("DoSomething", dosomething, dejoin, duration = 20, p_fail = 1/20),
    dejoin.name: Activity("Decouple", dejoin, FINISH, duration = 5, p_fail = 1/10, agg_type="
    dejoin")
})

# populate conops that were used
moonship_conops.sequence['dock'].agg_params['conops'] = together_conops

initial_vehicles = [
    (0.0, Vehicle("MoonShip", moonship_conops, {"propellant": 0})),
    (0.0, Vehicle("Tanker", tanker_conops, {"propellant": 100}))
]
```

Listing 6 Two Merge ConOps Snippet

The ConOps of this mission is roughly described by the above, with specific event declarations omitted for brevity. Critically, notice that moonship calls the aggregation event and then waits until the aggregated vehicle is finished to continue on its ConOps. Also, as the aggregate vehicle does not exist at simulation start, it is not a vehicle within *initial_vehicles*.



```
          VEHICLE MoonShip > Begin ACTIVITY:  Checkout

        EVENT:  tli_burn  @ time 190.0
          VEHICLE MoonShip > Begin ACTIVITY:  TranslunarCoast

        EVENT:  dock  @ time 200.0
          VEHICLE MoonShip > Begin ACTIVITY:  Docking
          VEHICLES ['Tanker', 'MoonShip'] > Begin ACTIVITY:  Docking
          VEHICLES ['Tanker', 'MoonShip'] > JOINED TO:  together

        EVENT:  INIT  @ time 210.0
          VEHICLE together > Begin ACTIVITY:  DockingSuccess

        EVENT:  almost_arriving  @ time 210.0
          VEHICLE MoonShip > Begin ACTIVITY:  spin

        EVENT:  dosomething  @ time 212.0
          VEHICLE together > Begin ACTIVITY:  DoSomething
        TERMINAL EVENT @ time 232.0
Predictate <together finished> Satisfied

        EVENT:  finale  @ time 212.0
          VEHICLE MoonShip > Begin ACTIVITY:  Finale
        TERMINAL EVENT @ time 214.0

COMPLETE


***DONE****
***********
```

Moonship and Tanker Dock into unified object named “together”
unified “together” goes about its ConOps
constituent entity moonship is waiting on a predicate event and not on its Completor yet
unified “together” does something ending in a Completor
moonship predicate event satisfied by unified “together”’s previous event and continues its conops, which completes

Fig. 4 Two Merge Simulation Output Sample

In the resulting simulation output, we can see that in the first block MoonShip and Tanker undergo the aggregation event into an unified object named "together". This aggregate vehicle "together" then undergoes it’s ConOps. Constituent vehicle "MoonShip" has not completed its ConOps, so it initializes a PredicatedActivity which is left to spin as "together" has not completed yet. Once "together" finishes its ConOps in a Completer, the PredicatedActivity is satisfied and the MoonShip continues its ConOps and subsequently finishes. The conceptual diagram corresponding to the part of the simulation shown which deals with the merged vehicles can be seen as the following.

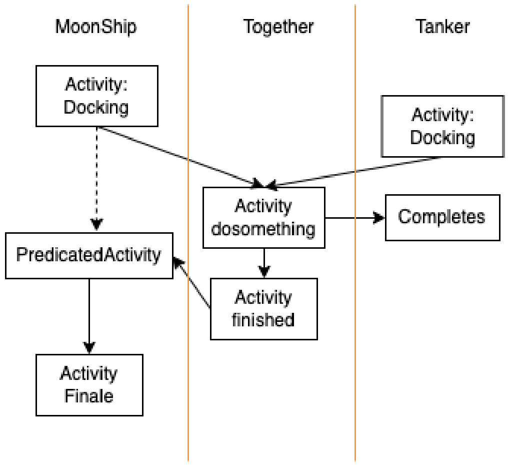


Fig. 5 Two Merge Simulation Diagram

E. Resource Tracking

Another key functionality that will be helpful in realizing our simulation goals is then the function of resource tracking. Namely, we want to be able to capture the state of any current vehicle and the resources it holds. For instance, if we have only a certain amount of fuel, we can then stipulate how much fuel each activity requires, and then implicate a restriction on how many times a certain activity may be able to be retried before the mission fails. For instance, if some docking does not work for whatever reason, we may be able to retry a few times, but ultimately the mission would still be bounded by the amount of fuel that it has available to do however many dockings. However, not every resource is of the same nature.

This for instance, can also be used to track failure quotas, say for some activity we can at most only allot 3 failure. Then we can assign a resource level of 3 to failure_allowance and then stipulate that each failure subtracts 1 from this resource. On the other hand, it can also be used in a more rough state variable interpretation. For instance, it can be used for tracking where the crew are, events can move crew to and from vehicles when they are aggregated together as described in the previous section.

We found the best way to implement this was then a dictionary collection that maps a certain set of resource keys into numerical values. Namely, crew, failure allowance, and fuel could all be tracked in this manner. The simulation itself is blind to the nature of these resources, it knows only that each vehicle has a resource list, and each activity may modify this resource list by a certain delta. That is to say, we can both deduct and add resources, for instance in the case of expending fuel and refueling. Every resource change is the uniquely matched up to the according resource, for instance if the vehicle as 5 of Resource A and the activity invokes a -3 change in Resource A and a +2 change in Resource B, the vehicle after the successful activity will have 2 of Resource A and 2 of Resource B. In a similar fashion, we then check that none of the resources are below 0, if they are, then a resource has been expended and the simulation fails. If we had more time, we would have considered making different severities of resource failures, namely, perhaps some resource failure is not mission critical and can still make the mission proceed without a failure altogether. In this interpretation, it is then implied that all resources are mission critical. This is implemented by the *resource* parameter on the vehicle and the *resource_change* parameter on the Activity.

```
# Example of Vehicle Declaration with resource variable
Vehicle("Tanker", tanker_conops, {"propellant": 100})

# Example of Depletion Event with resource_change variable
tanker_conops = ConOps({
    ...
    prop_full.name: Activity("Checkout", prop_full, tli_burn, duration = 110, resource_change = {
    "propellant": -50}),
    ...
})
```

Listing 7 Resource Tracking Parameter Examples

V. Experimental Results and Validation

A. Case Study 1: On-Orbit Propellant Aggregation

To demonstrate the utility of the SpaceMissionDES, we first consider a mission involving *on-orbit propellant aggregation*. As described in Section I, this mission involves deploying a Mars Transfer Vehicle (MTV). Historically, large exploration capabilities have been launched as large, monolithic vehicle stack. Consider the Apollo Program ConOps, illustrated in Figure 6, in which the Crew Module, Lunar Exploration, and Ascent module were all launched on a single vehicle. At the time, this represented one of the most complicated and risky ventures ever undertake by man.

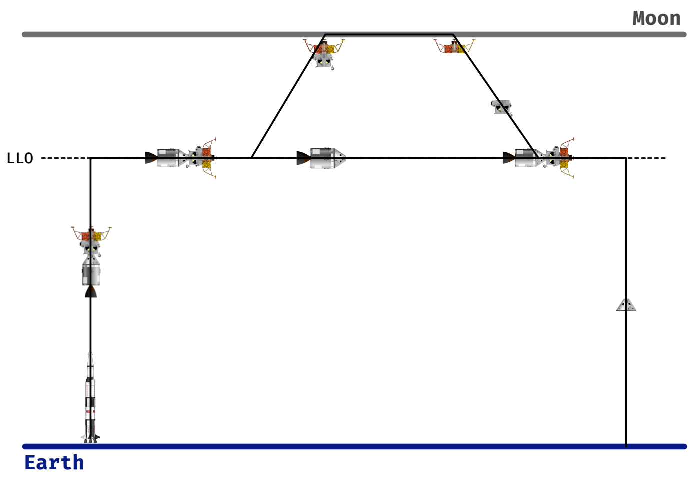


Fig. 6 ConOps for the Apollo Program Missions

Compared to the Apollo vehicle set, the total mass of an MTV is expected to be substantially larger. This makes it impractical to develop a launch vehicle that is sufficiently large enough to deploy an entire MTV in a single launch. Instead, space mission designers have developed numerous concepts to distribute the MTV across multiple launch vehicles. In some cases, the MTV is divided into multiple sub-vehicles that must aggregate on orbit. In the case we consider for this case study, we propose that the MTV should be launch *without* a full load of propellant to reduce the total launch mass. The propellant is then aggregated by launching one or more Tanker vehicles to rendezvous with and transfer propellant to the MTV. Figure 7 provides a graphical depiction of this ConOps.

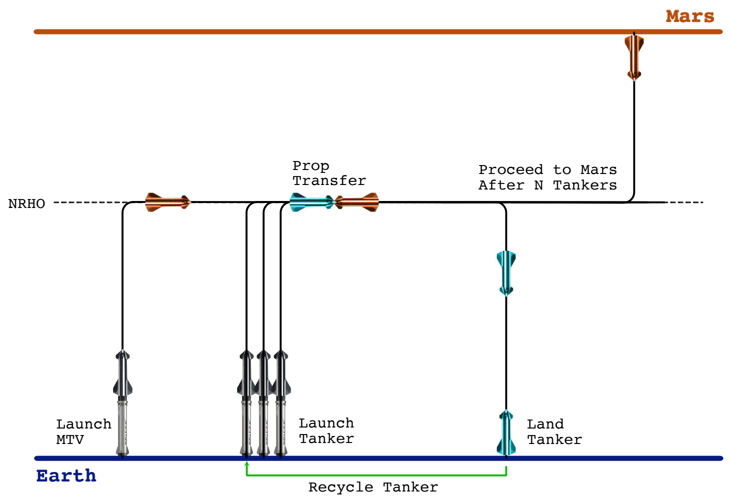


Fig. 7 A candidate Mars mission ConOps depends on aggregating propellant across numerous launches.

1. Translation to Discrete Event Simulation

In the language of discrete event simulation, this mission relies on two distinct types of entities; an MTV and a Tanker. Each of these entities are space vehicles with independent ConOps. These ConOps are shown as activity sequence diagrams in Figure 8.

The mission begins with the MTV entering a countdown period which lasts a fixed number of days. The countdown activity is assigned a 1/4 probability of failure, in which case, the countdown is reset. If this countdown activity completed successfully, the vehicle proceeds through launch, ascent, and orbit insertion before entering a *predicated activity* called propellant aggregation. Entering this activity instantiates a predicated event which can only be satisfied by a pre-set number of successful tanker missions. If the predicate is satisfied, we say that the MTV has received sufficient propellant to continue the mission and it proceeds through final checkout and enters the Mars Transit. Note that except for the countdown, failure in any of these activities leads to loss of mission. The ConOps for this mission is provided immediately below. Note that we can specify appropriate probabilities of failure and durations for each activity. Additionally, activities *Final Checkout* and *Recycle* showcase the ability to provide statistical distributions to represent delays of an unknown duration.

```
# ConOps
conops_MTV = ConOps({

    # Nominal
    INIT.name:     Activity(
        "Countdown", INIT, launch, duration=3, p_fail=pra["scrub"], failure=scrub
        ),
    launch.name:   Activity(
        "Ascent", launch, burnout, duration=1, p_fail=1/500
        ),
    burnout.name:  Activity(
        "Orbit Insertion", burnout, capture, duration=2, p_fail=pra["mps_burn"]
        ),
    capture.name:  PredicatedActivity(
        "Propellant Aggregation", capture, filled, predicate=until_N_transfers
```

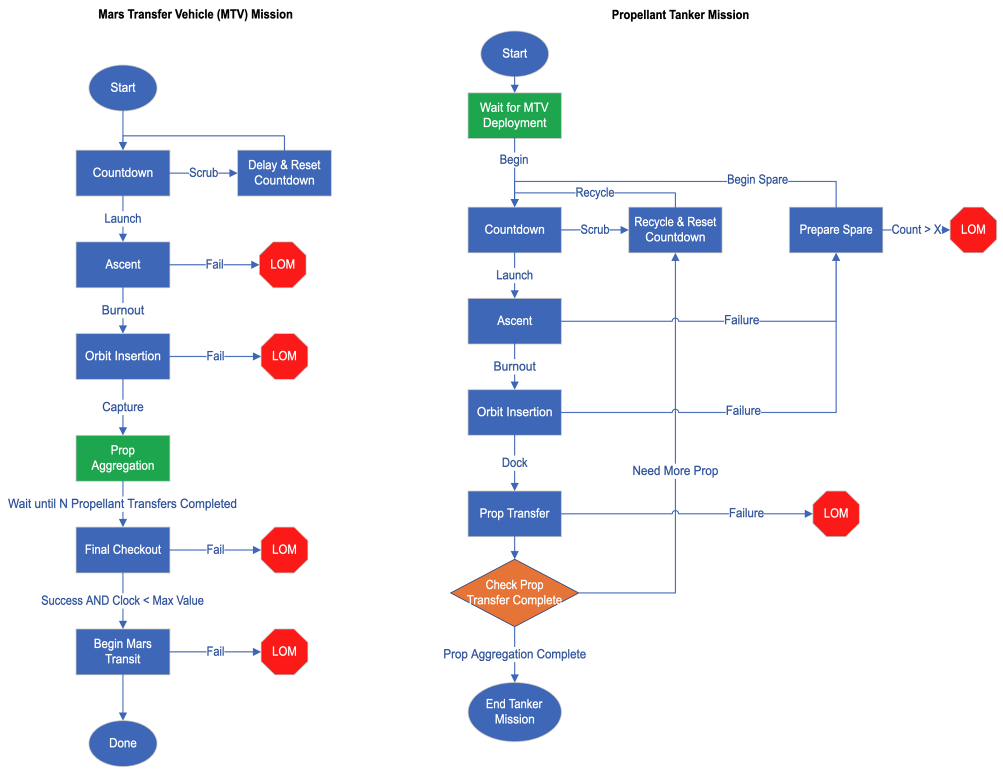


Fig. 8 The logical flows for each vehicle are express as an activity sequence diagram.

```
        ),
    filled.name:   Activity(
        "Final Checkout", filled, tmi_burn, duration=2, delay=weibull_min(c=0.5, loc=0, scale
    =0.1), p_fail=pra["checkout"]
        ),
    tmi_burn.name: Activity(
        "Begin Mars Transit",tmi_burn, DONE,duration=0, p_fail=pra["mps_burn"]
        ),

    # Contigencies
    scrub.name: Activity("Recycle", scrub, INIT, duration=0, delay=weibull_min(c=1, loc=0, scale
    =4)),
})
```

Listing 8 Defining a ConOps for the MTV Vehicle with SpaceMissionDES Interface

We also initialize the Tanker vehicle at time zero, but we begin its ConOps with predicated activity that cannot begin until the MTV enters it's predicated activity titled *Propellant Aggregation*. When that predicate is satisfied, the Tanker proceeds through countdown, ascent, and orbital insertion. Note that failures during ascent and orbital insertion do not lead to loss of mission, but instead to a contingency scenario in which a spare Tanker is processed, and the countdown is reset. Following orbital insertion, the tanker proceeds to dock with the MTV and enters the Propellant Transfer activity. After successful propellant transfer, the Tanker checks whether the MTV propellant quota has been met and either resets to fly another mission or completes the ConOps. See Appendix B for the full input file for this case.

2. Analysis Question 1.1 - How does the number of required transfers affect the mission?

There are key two parameters to trade for on-orbit propellant aggregation missions. The first is the number of tanker vehicles which are required to dock with and transfer propellant to the MTV. Practically speaking, this quantity can vary depending on the mission requirements and on the capacity of each individual tanker. To evaluate the effect of this parameter on the mission reliability and duration, we execute the model described above using the Monte Carlo driver.

For this case, we performed a convergence check (Figure 9) and determined that running the Monte Carlo for 5,000 replications offered a sufficient degree of consistency.

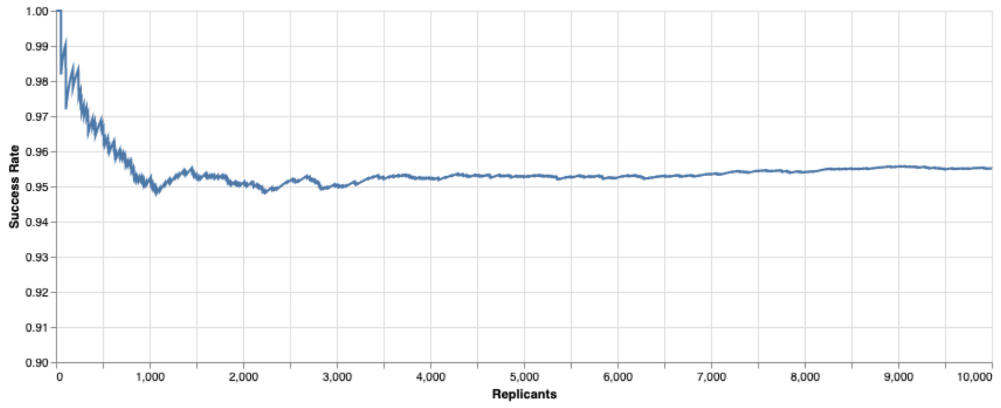


Fig. 9 A convergence test indicates that 5,000 replications should be sufficient.

We perform a parameter sweep on the number of transfers required, gather the reported durations and outcomes, and plot an empirical cumulative distribution function for each case. Then, assuming a target deployment duration at 120 days, we can examine the probability of completion. As expected, increasing the number of propellant transfers required reduces the probability that the deployment will be completed by the target.

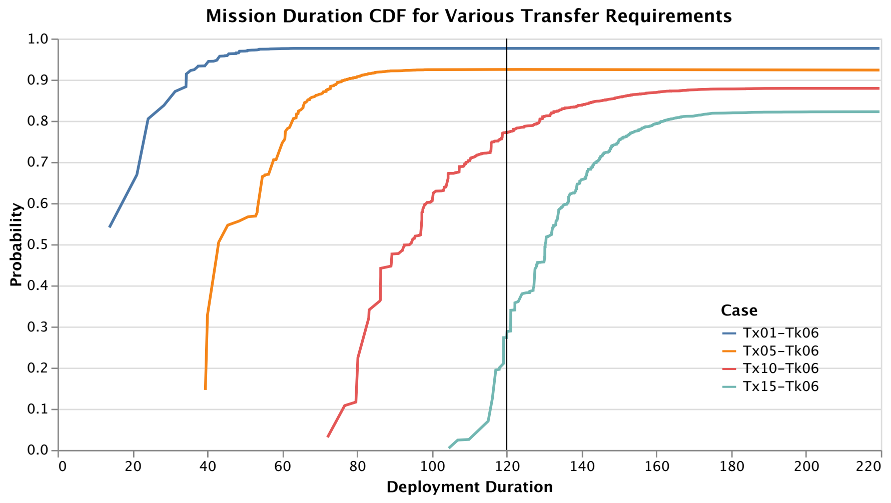


Fig. 10 CDFs for each case

3. Analysis Question 1.2 - How many tankers should be in the fleet?

The other salient question for this particular case is how many tankers are required in the fleet to maintain a high probability of success. To address this question, we ran Monte Carlo studies for a range of transfer requirements (Tx) and fleet sizes (Tk). The results of this parameter study are presented in Figure 11. The curves in that figure showcase an unexpected finding – that there is a point of diminishing returns at which adding additional tankers does not further reduce the number of cases that fail to complete the overall ConOps. Together, these findings allow space missions designers to select an appropriate vehicle concept and fleet size.

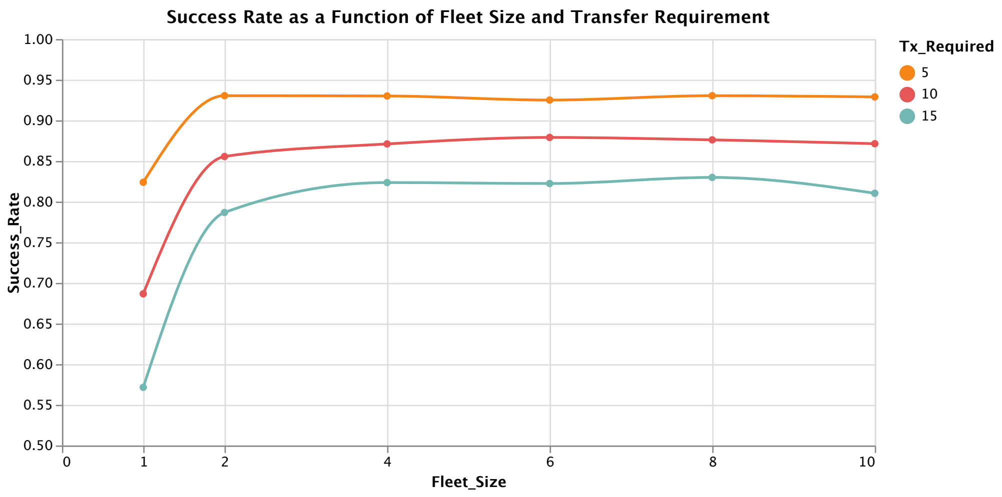


Fig. 11 Increasing the fleet size shows diminishing returns.

B. Case Study 2: Two Merge Mission

The discussion of this case study is integrated into the Generalized Object Model under Simulator Model, Section IV.

C. Case Study 3: Mars Split Mission

As described in Section I.C, the Mars Split Mission involves a more complex sequence of activities. The entire mission as proposed by [21], is depicted in Fig 12. Note, that we based our case study off only the 2039 portion of the mission, which is described below.

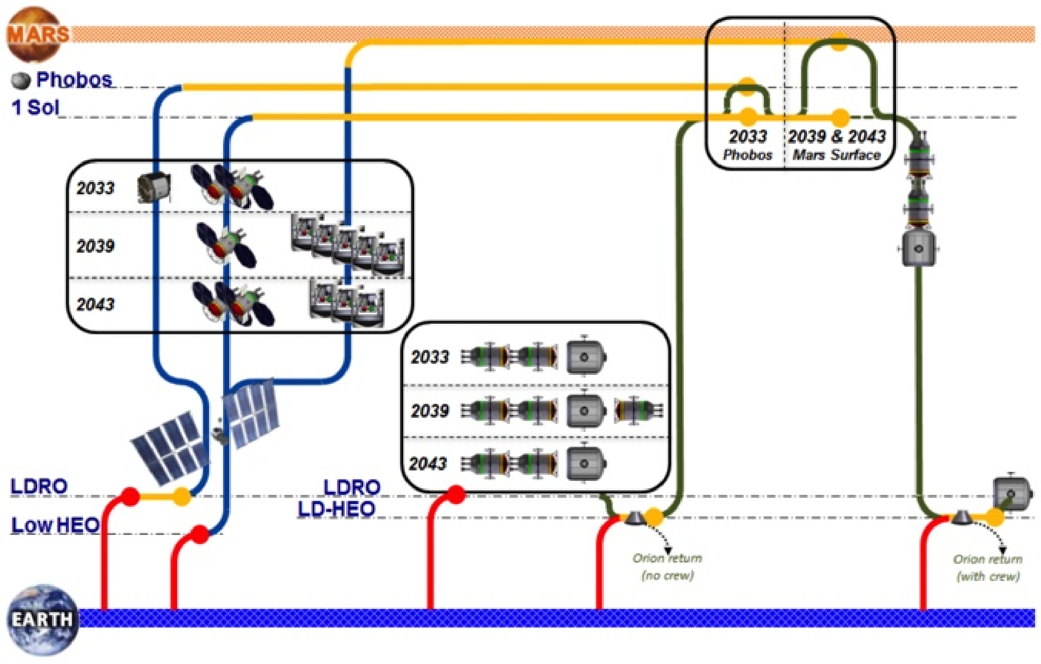


Fig. 12 Mars Split Mission Architecture (from [21]).

The first phase of this mission involves a pre-deployment of two Earth return propulsion systems to Mars parking orbit: the Trans Earth Injection - Methane Cryogenic Propulsion System (TEI-MCPS) and the Earth Orbit Insertion - Methane Cryogenic Propulsion System (EOI-MCPS). This is followed by the pre-deployment of the Mars Lander System (MLS) to Mars parking orbit.

Once these cargo are in place, the Crew-phase of the mission begins. A series of launches positions the Mars Transfer Injection (TMI) MCPS and the Mars Orbit Insertion (MOI) MCPS, followed by the crew Habitat (HAB) into a Lunar parking orbit. These elements are aggregated in lunar orbit, before the crew is delivered to them onboard a separate launch of the Orion spacecraft. Once the crew successfully rendezvous with the HAB and Mars transfer elements, they are sent Mars.

At Mars, the HAB rendezvous with the pre-deployed Mars Lander System (MLS), which takes them down to the surface of Mars for their 300 days of surface operations. The MLS contains a module called the Mars Crew Taxi (MCT), which launches the crew from the surface of Mars back to the HAB in parking orbit, which has rendezvoused with the TEI and EOI propulsion system during the surface operations. The TEI and EOI propulsion systems return the HAB and Crew to Lunar orbit, where they are met with another Orion spacecraft which returns them to the surface of Earth.

1. Translation to Discrete Event Simulation

Because the Mars Split Mission involves several entities aggregating and separating in space, a separate ConOps had to be created for each vehicle in the mission, as well as for each aggregation of vehicles in the mission. For example, consider the 2039 portion of the black rectangle in the center of Figure 12. Here we see a sequence of launches, followed by a sequence of rendezvous. Once the stack is assembled (HAB + MOI + TMI), stack waits for the crew from the Orion spacecraft. After the Orion docks and the crew transfers from Orion to the HAB, the Orion is jettisoned, and the stack fires one the TMI propulsion system then jettisons. The ConOps for the aggregate HAB + MOI + TMI vehicle is shown in the Listing below.

```python
# ConOps
conops_HAB_MOI_TMI = ConOps({
    INIT.name: PredicatedActivity(
        name='Waiting for Orion...',
        start=INIT,
        end=rendezvous,
        predicate=Predicate(name='Is Orion in Lunar Orbit?', check=vehicle_in_activity('Orion', '
    Lunar Insertion'))),
    rendezvous.name: Activity(
        name='Rendezvous-ing',
        start=rendezvous,
        end=wait,
        duration=1,
        p_fail=pra['dock'],
        agg_type='join',
        agg_params={'conops': conops_HAB_MOI_TMI_Orion, "vehicles": ['HAB + MOI-MCPS + TMI-MCPS',
     'Orion'], 'name': 'Orion + HAB + MOI-MCPS + TMI-MCPS'}),
    wait.name: PredicatedActivity(
        name='Waiting for Orion Jettison...',
        start=wait,
        end=transfer_burn,
        predicate=Predicate(name='Orion Jettisoned?', check=vehicle_in_activity('Orion + HAB +
    MOI-MCPS + TMI-MCPS', 'Jettisoning Orion'))),
    transfer_burn.name: Activity(
        name='Mars Transfer Burn, TMI Jettisoned',
        start=transfer_burn,
        end=DONE,
        duration=350,
        p_fail=pra['mps_burn'],
        agg_type='dejoin'),
})
})
```

Listing 9 Defining a ConOps for the Aggregate HAB + MOI + TMI Vehicle with SpaceMissionDES Interface

The corresponding output from the simulation for this particular aggregate vehicle ConOps is shown below. The hierarchy of vehicle aggregation for the HAB + MOI + TMI vehicle is shown in Figure 13.

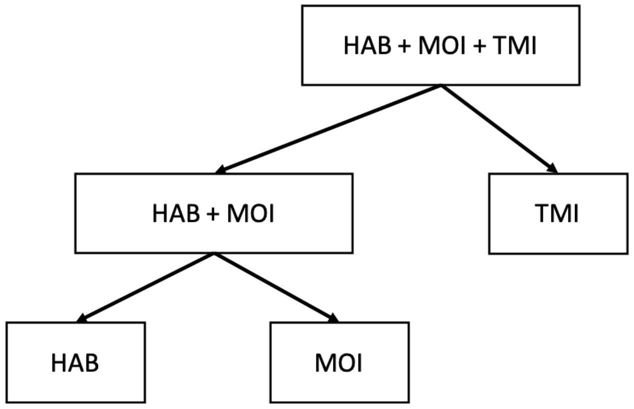


Fig. 13 Hierarchy of Vehicle Aggregation for the HAB + MOI + TMI Vehicle.

```
    EVENT:  INIT  @ time 3113.0
    VEHICLE HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Waiting for Orion...
Predictate <Is Orion in Lunar Orbit?> Satisfied
  EVENT:  wait for mars arrival  @ time 3113.0
    VEHICLE HAB + MOI-MCPS > Begin ACTIVITY:  Waiting for Mars Arrival...
  EVENT:  rendezvous  @ time 3113.0
    VEHICLE HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Rendezvous-ing
    VEHICLES ['HAB + MOI-MCPS + TMI-MCPS', 'Orion'] > Begin ACTIVITY:  Rendezvous-ing
    VEHICLES ['HAB + MOI-MCPS + TMI-MCPS', 'Orion'] > JOINED TO:  Orion + HAB + MOI-MCPS + TMI-
    MCPS
  EVENT:  crew transfer burn  @ time 3114.0
    VEHICLE Orion > Begin ACTIVITY:  Crew Transferring from Orion to HAB.
  TERMINAL EVENT named DONE @ time 3114.0
  EVENT:  INIT  @ time 3114.0
    VEHICLE Orion + HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Crew Transferring From Orion to
    HAB.
  EVENT:  wait  @ time 3114.0
    VEHICLE HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Waiting for Orion Jettison...
  EVENT:  separation orion  @ time 3114.0
    VEHICLE Orion + HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Jettisoning Orion
    VEHICLES Orion + HAB + MOI-MCPS + TMI-MCPS > decoupled CHILDREN:  ['HAB + MOI-MCPS + TMI-MCPS
    ', 'Orion']
  TERMINAL EVENT named DONE @ time 3114.0
Predictate <Orion Jettisoned?> Satisfied
  EVENT:  transfer burn  @ time 3114.0
    VEHICLE HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Mars Transfer Burn, TMI Jettisoned
    VEHICLES HAB + MOI-MCPS + TMI-MCPS > decoupled CHILDREN:  ['HAB + MOI-MCPS', 'TMI-MCPS']
  TERMINAL EVENT named DONE @ time 3464.0
Predictate <Arrived at Mars?> Satisfied
```

Listing 10 Mars Split Mission Simulation Output for HAB + MOI + TMI ConOps

2. Monte-Carlo Simulation for Mars Split Mission

By studying the results of the Monte Carlo, we can determine the relative contributions to failure on a per-vehicle and a per-activity level. This aides space mission designers by allowing them to determine which activities are drivers for overall probability of failure. For example, by reviewing Figure 14, we can determine that ascent and countdown failures are drivers, with Lunar insertion as the next contributor. Likewise, by reviewing the Figure 15, we can determine the which vehicles or aggregation of vehicles were most prone to failure. We see that the pre-deployment of SEP + TEI / EOI propulsion system are key drivers.

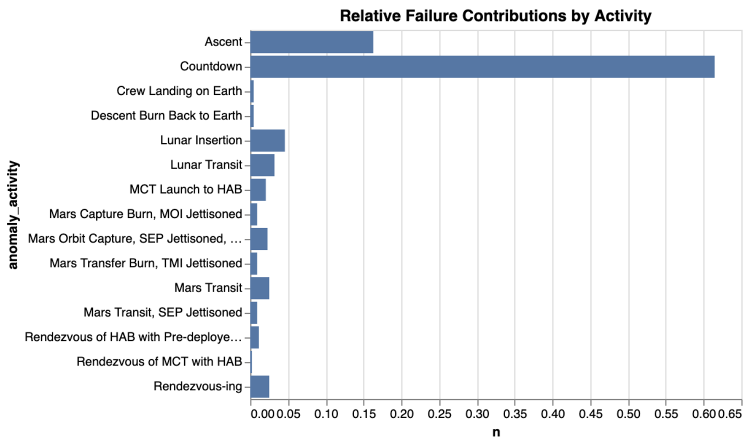


Fig. 14 Relative Contributions to Failure by Activity

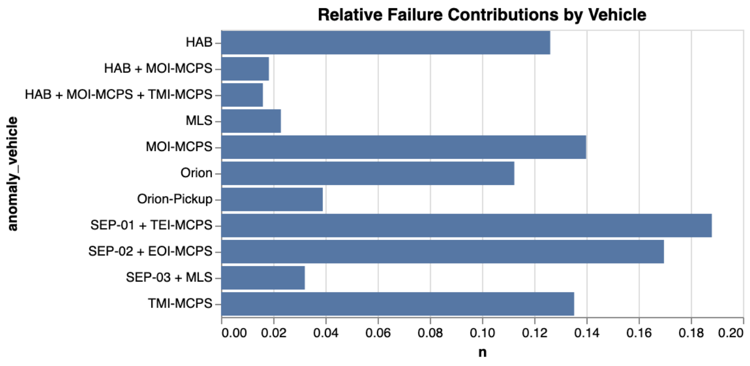


Fig. 15 Relative Contributions to Failure by Vehicle

VI. Conclusion

As we have shown, space mission logistics can be a complex system of vehicles, activities, and schedules. The three case studies presented here demonstrate the utility of Discrete Event Simulation as a powerful tool for space mission designers to predict key metrics and identify primary sources of failure. In developing our DES models, we came across many challenges that helped inform our development process. Due to the complex nature of space missions, we made sure to make the activity-centric perspectives and paradigms clear from the start. In this manner, we were able to more clearly wireplan which parts of the simulation needed what levels of granularity. For instance, a launch activity could be split up into more granular constituent activities. Striking a careful balance between standardized paradigms and meaningful detail was a constant discussion throughout our project. On the technical side, the simulation evolved from a central routine to add additional advanced features that allowed for more complicated operations, for instance, the generalized vehicles, the predicate activities, and the resource tracking. Some of these functionalities could be further iterated upon in future work by introducing more possibilities. For example, the failure events could be split by severity rather than a binary success or fail result. However, for our purposes we tried to optimize to prioritize granularity for more mission-critical sections. Future work would involve (1) increasing the fidelity of the simulation by including more nuanced activities, (2) adding more refined failure and retry processes, and (3) including the complex system of ground logistics into the model. In summary, our project centered around building an adaptable discrete event simulation framework for space mission simulation and implementing a complex case study of a space mission to Mars.

References

[1] EBELING, C., “Integrating O/S models during conceptual design, part 2(Annual Report, 1 Jul.- 31 Dec. 1994),” 1994.

[2] Cates, G. R., Steele, M. J., Mollaghasemi, M., and Rabadi, G., “Modeling the space shuttle,” *Proceedings of the Winter Simulation Conference*, Vol. 1, IEEE, 2002, pp. 754–762.

[3] Robinson, S., *Simulation: The Practice of Model Development and Use*, John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[4] Zhang, X., “Application of discrete event simulation in health care: A systematic review,” *BMC Health Services Research*, Vol. 18, 2018. https://doi.org/10.1186/s12913-018-3456-4.

[5] De Landtsheer, R., Ospina, G., Massonet, P., Ponsard, C., Printz, S., Jeschke, S., Härtel, L., Cube, J., and Schmitt, R., *Assessment of Risks in Manufacturing Using Discrete-Event Simulation*, 2016, pp. 869–891. https://doi.org/10.1007/978-3-319-42620-4_66.

[6] Larocque, G., and Lipoff, S., “Application of discrete event simulation to network protocol modeling,” *Proceedings of ICUPC - 5th International Conference on Universal Personal Communications*, Vol. 2, 1996, pp. 508–512 vol.2. https://doi.org/10.1109/ICUPC.1996.562625.

[7] Stojmenovic, I., “Simulations in wireless sensor and ad hoc networks: matching and advancing models, metrics, and solutions,” *IEEE Communications Magazine*, Vol. 46, No. 12, 2008, pp. 102–107.

[8] Mousavi, S. M., Rabiee, H. R., Moshref, M., and Dabirmoghaddam, A., “MobiSim: A Framework for Simulation of Mobility Models in Mobile Ad-Hoc Networks,” *Third IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2007)*, 2007, pp. 82–82. https://doi.org/10.1109/WIMOB.2007.4390876.

[9] Yousefpour, A., Ishigaki, G., Gour, R., and Jue, J. P., “On Reducing IoT Service Delay via Fog Offloading,” *IEEE Internet of Things Journal*, Vol. 5, No. 2, 2018, pp. 998–1010. https://doi.org/10.1109/JIOT.2017.2788802.

[10] Cates, G., and Mollaghasemi, M., “A Discrete Event Simulation Model for Assembling the International Space Station,” *Proceedings of the Winter Simulation Conference, 2005.*, IEEE, Orlando, FL. USA, 2005, pp. 1260–1264. https://doi.org/10.1109/WSC.2005.1574385, URL http://ieeexplore.ieee.org/document/1574385/.

[11] Cates, G., and Mollaghasemi, M., “Supporting the Vision for Space with Discrete Event Simulation,” *Proceedings of the Winter Simulation Conference, 2005.*, IEEE, Orlando, FL. USA, 2005, pp. 1306–1310. https://doi.org/10.1109/WSC.2005.1574391, URL http://ieeexplore.ieee.org/document/1574391/.

[12] Cates, G., Cirillo, W., and Stromgren, C., “Low Earth Orbit Rendezvous Strategy for Lunar Missions,” *Proceedings of the 2006 Winter Simulation Conference*, IEEE, Monterey, CA, USA, 2006, pp. 1248–1252. https://doi.org/10.1109/WSC.2006.323220, URL http://ieeexplore.ieee.org/document/4117744/.

[13] Cirillo, W., Stromgren, C., and Cates, G., “Risk analysis of on-orbit spacecraft refueling concepts,” *AIAA Space 2010 Conference & Exposition*, 2010, p. 8832.

[14] Cates, G., Gelito, J., Stromgren, C., Cirillo, W., and Goodliff, K., “Launch and assembly reliability analysis for human space exploration missions,” *2012 IEEE Aerospace Conference*, IEEE, 2012, pp. 1–20.

[15] Cates, G., Stromgren, C., Cirillo, W., and Goodliff, K., “Launch and assembly reliability analysis for Mars human space exploration missions,” *2013 IEEE Aerospace Conference*, IEEE, 2013, pp. 1–20.

[16] Cates, G., Stromgren, C., Arney, D., Cirillo, W., and Goodliff, K., “International human mission to Mars: Analyzing a conceptual launch and assembly campaign,” *2014 IEEE Aerospace Conference*, IEEE, 2014, pp. 1–18.

[17] Cates, G., Stromgren, C., Mattfeld, B., Cirillo, W., and Goodliff, K., “The exploration of Mars launch & assembly simulation,” *2016 IEEE Aerospace Conference*, IEEE, 2016, pp. 1–12.

[18] Cates, G., Coley, D., Goodliff, K., Cirillo, W., and Stromgren, C., “Launch Availability Analysis for the Artemis Program,” *2020 IEEE Aerospace Conference*, ????

[19] Leonard, D., Parsons, J., and Cates, G., “Using discrete event simulation to model fluid commodity use by the space launch system,” *Proceedings of the Winter Simulation Conference 2014*, 2014, pp. 2954–2965. https://doi.org/10.1109/WSC.2014.7020135.

[20] Trocine, L., Cummings, N., Bazzana, A., Rychlik, N., LeCroy, K., Cates, G., and Mollaghasemi, M., “Statistical and Probabilistic Extensions to Ground Operations’ Discrete Event Simulation Modeling,” *SpaceOps 2010 Conference*, American Institute of Aeronautics and Astronautics, Huntsville, Alabama, 2010. https://doi.org/10.2514/6.2010-2027, URL https://arc.aiaa.org/doi/10.2514/6.2010-2027.

[21] Percy, T., McGuire, M., and Polsgrove, T., “In-space transportation for NASA’s Evolvable Mars Campaign,” *AIAA Space 2015 Conference and Exposition*, 2015, p. 4519.

A. Division of Labor

To complete this project, we had to research the concepts and inputs for relevant space missions, develop a tailored discrete event simulation, run a series of experiments, and post-process them to gain insight. Benjamin Merrel led the development of the simulation core, Monte Carlo execution capabilities, and post-processing scripts. He also developed and executed Case Study 1: On-Orbit Propellant Aggregation. Ed Chen lead efforts to generalize the various simulation components and implemented the capability to stacked vehicles into aggregate entities. He also worked with David to better refine the model framework logic to suit Case Study 2. David Gomez led the research and development of the conceptual models and the collection of inputs to build the space missions. He also led the experimentation and implementation for Case Study 3: the Mars Split Mission.

B. SpaceMissionDES Input File Listings for Case Studies

```python
# ------------------------------------------------------------------------------------------------
# Case04_PropAggregation
# Standard Library
import logging
# Dependencies
from scipy.stats import *
# SpaceMissionDES
from objects.events import *
from objects.activities import *
from objects.vehicles import Vehicle
from objects.predicates import Predicate, vehicle_in_activity

initial_vehicles = []

# ------------------------------------------------------------------------------------------------
# Problem Settings
# ------------------------------------------------------------------------------------------------

# Probabilistic Inputs
pra = {
    "scrub": 1/4,
    "ascent": 2/100,
    "RPO": 1/500,
    "mps_burn": 1/250,
    "dock": 1/200,
    "checkout": 1/1000
}

# Fleet Management and Requirements
N_transfers_required = 1
N_tankers_available = 6

# ------------------------------------------------------------------------------------------------
# Vehicle and ConOps Settings
# ------------------------------------------------------------------------------------------------
# MTV - Mars Transfer Vehicle

# Events
INIT     = Event("INIT")
launch   = Event("launch")
burnout  = Event("burnout")
capture  = Event("capture")
filled   = Event("filled")
tmi_burn = Event("tmi_burn")
moi_burn = Event("moi_burn")
ARRIVE   = Completor("ARRIVE")
DONE     = Completor("DON")
scrub    = Event("scrub")

# Predicates
def check_transers(p, sim):
    # Check on the tanker
    tanker = sim.entities["Tanker"]
    if tanker.state["transfers"] < N_transfers_required:
        return False
    else:
        logging.info(f"Predictate <{p.predicate.name}> Satisfied")
        return True

until_N_transfers = Predicate(f"Wait until {N_transfers_required} propellant transfers are
    completed", check_transers)

# ConOps
conops_MTV = ConOps({

    # Nominal
    INIT.name:     Activity("Countdown",              INIT,     launch,   duration=3, p_fail=pra["scrub"], failure=scrub),
    launch.name:   Activity("Ascent",                 launch,   burnout,  duration=1, p_fail=1/500),
    burnout.name:  Activity("Orbit Insertion",        burnout,  capture,  duration=2, p_fail=pra["mps_burn"]),
    capture.name:  PredicatedActivity("Aggregation", capture,  filled,   predicate=until_N_transfers),
    filled.name:   Activity("Final Checkout",         filled,   tmi_burn, duration=2, delay=weibull_min(c=0.5, loc=0, scale=0.1), p_fail=pra["checkout"]),  #TODO: add check to see if clock is < critical value
    tmi_burn.name: Activity("Begin Mars Transit",     tmi_burn, DONE,     duration=0, p_fail=pra["mps_burn"]),

    # Contigencies
    scrub.name: Activity("Recycle", scrub, INIT, duration=0, delay=weibull_min(c=1, loc=0, scale=4)),
})

initial_vehicles += [
    (0.0, Vehicle("MTV", conops_MTV))
]

# ------------------------------------------------------------------------------------------
# Tanker

# Events
begin    = Event("begin")
approach = Event("approach")
dock     = Event("dock")
undock   = Event("undock")
land     = Event("land")

spare    = Event("spare")
no_more_spares = Event("no_more_spares")

# Predicates
after_MTV_deploy = Predicate("Wait for MTV", vehicle_in_activity(vehicle="MTV", activity="Propellant Aggregation"))

# Branching Events
def limit_spares(sim, vehicle):          # branching logic functions always take (sim, vehicle)
    if vehicle.state["fleet"] > 0:
        return begin
    else:
        return no_more_spares

limit_tanker_spares = Branch(f"Limit to {N_tankers_available-1} Spares", logic=limit_spares)


def count_transfers(sim, vehicle):          # branching logic functions always take (sim, vehicle)
    if vehicle.state["transfers"] < N_transfers_required:
        return begin
    else:
        return land

until_n_tankers = Branch(f"Until {N_transfers_required} Transfers", logic=count_transfers)

# ConOps
conops_Tanker = ConOps({

    # Nominal
    INIT.name:     PredicatedActivity("Wait for MTV Deploy", INIT, limit_tanker_spares, predicate=after_MTV_deploy),

    begin.name:    Activity("Countdown", begin, launch, duration=0, p_fail=pra["scrub"], failure=scrub),
    launch.name:   Activity("Ascent", launch, burnout, duration=1, p_fail=pra["ascent"], failure=
    spare, update={'flights': 1}),
    burnout.name:  Activity("Orbit Insertion", burnout, capture, duration=2, p_fail=pra["mps_burn
    "], failure=spare),
    capture.name:  Activity("Redezvous", capture, approach, duration = 1, p_fail=pra["RPO"]),
    approach.name: Activity("RPOD", approach, dock, duration = 1, p_fail=pra["dock"]),
    dock.name:     Activity("Prop Transfer", dock, undock, duration=0.5, update={'transfers': 1})
    ,
    undock.name:   Activity("Return to Base", undock, until_n_tankers, duration = 1, p_fail=pra["
    dock"]),
    land.name:     Activity("End Tanker Mission", land, DONE, duration=0),

    # Contigencies
    scrub.name:    Activity("Recycle", scrub, INIT, duration=0, delay=weibull_min(c=1, loc=0,
    scale=4)),
    spare.name:    Activity("Prepare Spare", spare, INIT, duration=10, delay=weibull_min(c=0.8,
    loc=0, scale=5), update={'fleet': -1}),
    no_more_spares.name: Activity("Out of Spare Tankers", no_more_spares, Failure(), duration=0)
})

initial_vehicles += [
    (0.0, Vehicle("Tanker", conops_Tanker, state={'flights':0, 'transfers':0, 'failures':0, '
    fleet': N_tankers_available}))
]
```

Listing 11 Full Input File for Case Study 1; On-Orbit Propellant Aggregation

```
***********
***BEGIN***


  EVENT:  INIT  @ time 0.00
    VEHICLE SEP-01 > Begin ACTIVITY:  Vehicle Creation
  TERMINAL EVENT named DONE @ time 0.0

  EVENT:  INIT  @ time 0.00
    VEHICLE TEI-MCPS > Begin ACTIVITY:  Vehicle Creation
  TERMINAL EVENT named DONE @ time 0.0

  EVENT:  INIT  @ time 0.00
    VEHICLE SEP-02 > Begin ACTIVITY:  Vehicle Creation
  TERMINAL EVENT named DONE @ time 0.0

  EVENT:  INIT  @ time 0.00
    VEHICLE EOI-MCPS > Begin ACTIVITY:  Vehicle Creation
  TERMINAL EVENT named DONE @ time 0.0

  EVENT:  INIT  @ time 1.00
    VEHICLE SEP-02 + EOI-MCPS > Begin ACTIVITY:  Countdown

  EVENT:  INIT  @ time 1.00
    VEHICLE SEP-01 + TEI-MCPS > Begin ACTIVITY:  Countdown

  EVENT:  launch  @ time 4.00
    VEHICLE SEP-02 + EOI-MCPS > Begin ACTIVITY:  Ascent

  EVENT:  launch  @ time 4.00
    VEHICLE SEP-01 + TEI-MCPS > Begin ACTIVITY:  Ascent

  EVENT:  transfer burn  @ time 5.00
    VEHICLE SEP-02 + EOI-MCPS > Begin ACTIVITY:  Mars Transit

  EVENT:  transfer burn  @ time 5.00
    VEHICLE SEP-01 + TEI-MCPS > Begin ACTIVITY:  Mars Transit

  EVENT:  capture burn  @ time 1462.00
    VEHICLE SEP-02 + EOI-MCPS > Begin ACTIVITY:  Mars Orbit Capture, SEP Jettisoned, Awaiting
    Crew...
```

```
TERMINAL EVENT named DONE @ time 1472.0

EVENT:  capture burn  @ time 1462.00
  VEHICLE SEP-01 + TEI-MCPS > Begin ACTIVITY:  Mars Orbit Capture, SEP Jettisoned, Awaiting
  Crew...
TERMINAL EVENT named DONE @ time 1472.0

EVENT:  INIT  @ time 1500.00
  VEHICLE SEP-03 > Begin ACTIVITY:  Vehicle Creation
TERMINAL EVENT named DONE @ time 1500.0

EVENT:  INIT  @ time 1501.00
  VEHICLE MLS > Begin ACTIVITY:  Vehicle Creation

EVENT:  wait  @ time 1501.00
  VEHICLE MLS > Begin ACTIVITY:  Waiting for HAB Separation

EVENT:  INIT  @ time 1502.00
  VEHICLE SEP-03 + MLS > Begin ACTIVITY:  Countdown

EVENT:  launch  @ time 1505.00
  VEHICLE SEP-03 + MLS > Begin ACTIVITY:  Ascent

EVENT:  transfer burn  @ time 1506.00
  VEHICLE SEP-03 + MLS > Begin ACTIVITY:  Mars Transit, SEP Jettisoned
  VEHICLES SEP-03 + MLS > decoupled CHILDREN:  ['SEP-03', 'MLS']

EVENT:  capture burn  @ time 2963.00
  VEHICLE SEP-03 + MLS > Begin ACTIVITY:  Mars Aerocapture, awaiting crew...
TERMINAL EVENT named DONE @ time 2968.0

EVENT:  INIT  @ time 3100.00
  VEHICLE HAB > Begin ACTIVITY:  Countdown

EVENT:  INIT  @ time 3101.00
  VEHICLE MOI-MCPS > Begin ACTIVITY:  Countdown

EVENT:  INIT  @ time 3102.00
  VEHICLE TMI-MCPS > Begin ACTIVITY:  Countdown

EVENT:  INIT  @ time 3103.00
  VEHICLE Orion > Begin ACTIVITY:  Countdown

EVENT:  launch  @ time 3103.00
  VEHICLE HAB > Begin ACTIVITY:  Ascent

EVENT:  INIT  @ time 3104.00
  VEHICLE Orion-Pickup > Begin ACTIVITY:  Waiting for Crew Arrival to LDRO

EVENT:  launch  @ time 3104.00
  VEHICLE MOI-MCPS > Begin ACTIVITY:  Ascent

EVENT:  transfer burn  @ time 3104.00
  VEHICLE HAB > Begin ACTIVITY:  Lunar Transit

EVENT:  launch  @ time 3105.00
  VEHICLE TMI-MCPS > Begin ACTIVITY:  Ascent

EVENT:  transfer burn  @ time 3105.00
  VEHICLE MOI-MCPS > Begin ACTIVITY:  Lunar Transit

EVENT:  launch  @ time 3106.00
  VEHICLE Orion > Begin ACTIVITY:  Ascent

EVENT:  transfer burn  @ time 3106.00
  VEHICLE TMI-MCPS > Begin ACTIVITY:  Lunar Transit

EVENT:  transfer burn  @ time 3107.00
```

```
    VEHICLE Orion > Begin ACTIVITY:  Lunar Transit

  EVENT:  capture burn  @ time 3110.00
    VEHICLE HAB > Begin ACTIVITY:  Lunar Insertion

  EVENT:  capture burn  @ time 3111.00
    VEHICLE MOI-MCPS > Begin ACTIVITY:  Lunar Insertion
  TERMINAL EVENT named DONE @ time 3112.0

  EVENT:  wait  @ time 3111.00
    VEHICLE HAB > Begin ACTIVITY:  Waiting for MOI...
Predictate <Is MOI in Lunar Orbit?> Satisfied

  EVENT:  rendezvous  @ time 3111.00
    VEHICLE HAB > Begin ACTIVITY:  Rendezvous-ing
    VEHICLES ['HAB', 'MOI-MCPS'] > Begin ACTIVITY:  Rendezvous-ing
    VEHICLES ['HAB', 'MOI-MCPS'] > JOINED TO:  HAB + MOI-MCPS

  EVENT:  capture burn  @ time 3112.00
    VEHICLE TMI-MCPS > Begin ACTIVITY:  Lunar Insertion
  TERMINAL EVENT named DONE @ time 3113.0

  EVENT:  INIT  @ time 3112.00
    VEHICLE HAB + MOI-MCPS > Begin ACTIVITY:  Successfully Rendezvous!

  EVENT:  wait for mars arrival  @ time 3112.00
    VEHICLE HAB > Begin ACTIVITY:  Waiting for Mars Arrival...

  EVENT:  wait  @ time 3112.00
    VEHICLE HAB + MOI-MCPS > Begin ACTIVITY:  Waiting for TMI...
Predictate <Is TMI in Lunar Orbit?> Satisfied

  EVENT:  rendezvous  @ time 3112.00
    VEHICLE HAB + MOI-MCPS > Begin ACTIVITY:  Rendezvous-ing
    VEHICLES ['HAB + MOI-MCPS', 'TMI-MCPS'] > Begin ACTIVITY:  Rendezvous-ing
    VEHICLES ['HAB + MOI-MCPS', 'TMI-MCPS'] > JOINED TO:  HAB + MOI-MCPS + TMI-MCPS

  EVENT:  capture burn  @ time 3113.00
    VEHICLE Orion > Begin ACTIVITY:  Lunar Insertion

  EVENT:  INIT  @ time 3113.00
    VEHICLE HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Waiting for Orion...
Predictate <Is Orion in Lunar Orbit?> Satisfied

  EVENT:  wait for mars arrival  @ time 3113.00
    VEHICLE HAB + MOI-MCPS > Begin ACTIVITY:  Waiting for Mars Arrival...

  EVENT:  rendezvous  @ time 3113.00
    VEHICLE HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Rendezvous-ing
    VEHICLES ['HAB + MOI-MCPS + TMI-MCPS', 'Orion'] > Begin ACTIVITY:  Rendezvous-ing
    VEHICLES ['HAB + MOI-MCPS + TMI-MCPS', 'Orion'] > JOINED TO:  Orion + HAB + MOI-MCPS + TMI-
    MCPS

  EVENT:  crew transfer burn  @ time 3114.00
    VEHICLE Orion > Begin ACTIVITY:  Crew Transferring from Orion to HAB.
  TERMINAL EVENT named DONE @ time 3114.0

  EVENT:  INIT  @ time 3114.00
    VEHICLE Orion + HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Crew Transferring From Orion to
    HAB.

  EVENT:  wait  @ time 3114.00
    VEHICLE HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Waiting for Orion Jettison...

  EVENT:  separation orion  @ time 3114.00
    VEHICLE Orion + HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Jettisoning Orion
    VEHICLES Orion + HAB + MOI-MCPS + TMI-MCPS > decoupled CHILDREN:  ['HAB + MOI-MCPS + TMI-MCPS
    ', 'Orion']
  TERMINAL EVENT named DONE @ time 3114.0
Predictate <Orion Jettisoned?> Satisfied

  EVENT:  transfer burn  @ time 3114.00
    VEHICLE HAB + MOI-MCPS + TMI-MCPS > Begin ACTIVITY:  Mars Transfer Burn, TMI Jettisoned
    VEHICLES HAB + MOI-MCPS + TMI-MCPS > decoupled CHILDREN:  ['HAB + MOI-MCPS', 'TMI-MCPS']
  TERMINAL EVENT named DONE @ time 3464.0
Predictate <Arrived at Mars?> Satisfied

  EVENT:  capture burn  @ time 3114.00
    VEHICLE HAB + MOI-MCPS > Begin ACTIVITY:  Mars Capture Burn, MOI Jettisoned
    VEHICLES HAB + MOI-MCPS > decoupled CHILDREN:  ['HAB', 'MOI-MCPS']
  TERMINAL EVENT named DONE @ time 3114.0
Predictate <Arrived at Mars?> Satisfied

  EVENT:  rendezvous of HAB with MLS  @ time 3114.00
    VEHICLE HAB > Begin ACTIVITY:  Rendezvous of HAB with MLS, Crew Transferring to MLS
    VEHICLES ['HAB', 'MLS'] > Begin ACTIVITY:  Rendezvous of HAB with MLS, Crew Transferring to
    MLS
    VEHICLES ['HAB', 'MLS'] > JOINED TO:  HAB + MLS

  EVENT:  INIT  @ time 3115.00
    VEHICLE HAB + MLS > Begin ACTIVITY:  Crew Transferred to MLS, HAB Separating
    VEHICLES HAB + MLS > decoupled CHILDREN:  ['HAB', 'MLS']
  TERMINAL EVENT named DONE @ time 3116.0
Predictate <HAB Separated?> Satisfied

  EVENT:  wait for surface operations  @ time 3115.00
    VEHICLE HAB > Begin ACTIVITY:  Waiting for Surface Operations

  EVENT:  transfer burn  @ time 3115.00
    VEHICLE MLS > Begin ACTIVITY:  Transfer Burn to Mars Surface

  EVENT:  surface operations  @ time 3120.00
    VEHICLE MLS > Begin ACTIVITY:  Mars Surface Operations
Predictate <Surface Operations Started?> Satisfied

  EVENT:  rendezvous of HAB with EOI  @ time 3120.00
    VEHICLE HAB > Begin ACTIVITY:  Rendezvous of HAB with Pre-deployed EOI
    VEHICLES ['HAB', 'EOI-MCPS'] > Begin ACTIVITY:  Rendezvous of HAB with Pre-deployed EOI
    VEHICLES ['HAB', 'EOI-MCPS'] > JOINED TO:  HAB + EOI-MCPS
  TERMINAL EVENT named DONE @ time 3121.0

  EVENT:  INIT  @ time 3121.00
    VEHICLE HAB + EOI-MCPS > Begin ACTIVITY:  Rendezvous of HAB + EOI-MCPS with TEI-MCPS
    VEHICLES ['HAB + EOI-MCPS', 'TEI-MCPS'] > Begin ACTIVITY:  Rendezvous of HAB + EOI-MCPS with
    TEI-MCPS
    VEHICLES ['HAB + EOI-MCPS', 'TEI-MCPS'] > JOINED TO:  HAB + EOI-MCPS + TEI-MCPS

  EVENT:  INIT  @ time 3122.00
    VEHICLE HAB + EOI-MCPS + TEI-MCPS > Begin ACTIVITY:  Waiting for Crew...

  EVENT:  wait  @ time 3122.00
    VEHICLE HAB + EOI-MCPS > Begin ACTIVITY:  Waiting for TEI Burn...

  EVENT:  crew transfer burn  @ time 3420.00
    VEHICLE MLS > Begin ACTIVITY:  Boarding MCT

  EVENT:  launch  @ time 3420.00
    VEHICLE MLS > Begin ACTIVITY:  MCT Launch to HAB

  EVENT:  rendezvous  @ time 3421.00
    VEHICLE MLS > Begin ACTIVITY:  Rendezvous of MCT with HAB
  TERMINAL EVENT named DONE @ time 3422.0
Predictate <Crew Onboard?> Satisfied

  EVENT:  transfer burn  @ time 3421.00
    VEHICLE HAB + EOI-MCPS + TEI-MCPS > Begin ACTIVITY:  Trans Earth Injection Burn, TEI
    Jettisoned
    VEHICLES HAB + EOI-MCPS + TEI-MCPS > decoupled CHILDREN:  ['HAB + EOI-MCPS', 'TEI-MCPS']
  TERMINAL EVENT named DONE @ time 3619.0
Predictate <TEI Burn Complete?> Satisfied

  EVENT:  capture burn  @ time 3421.00
    VEHICLE HAB + EOI-MCPS > Begin ACTIVITY:  Earth Orbit Insertion Burn into LDRO, EOI
    Jettisoned
    VEHICLES HAB + EOI-MCPS > decoupled CHILDREN:  ['HAB', 'EOI-MCPS']
  TERMINAL EVENT named DONE @ time 3422.0
Predictate <Crew at LDRO?> Satisfied

  EVENT:  launch  @ time 3421.00
    VEHICLE Orion-Pickup > Begin ACTIVITY:  Ascent

  EVENT:  transfer burn  @ time 3422.00
    VEHICLE Orion-Pickup > Begin ACTIVITY:  Lunar Transit

  EVENT:  capture burn  @ time 3428.00
    VEHICLE Orion-Pickup > Begin ACTIVITY:  Lunar Insertion

  EVENT:  rendezvous  @ time 3429.00
    VEHICLE Orion-Pickup > Begin ACTIVITY:  Rendezvous of Orion with HAB, Crew Transferring to
    Orion
    VEHICLES ['HAB', 'Orion-Pickup'] > Begin ACTIVITY:  Rendezvous of Orion with HAB, Crew
    Transferring to Orion
    VEHICLES ['HAB', 'Orion-Pickup'] > JOINED TO:  HAB + Orion-Pickup

  EVENT:  INIT  @ time 3430.00
    VEHICLE HAB + Orion-Pickup > Begin ACTIVITY:  Crew Successfully Transferred from HAB to Orion
    , HAB Jettisoned
    VEHICLES HAB + Orion-Pickup > decoupled CHILDREN:  ['HAB', 'Orion-Pickup']
  TERMINAL EVENT named DONE @ time 3430.0

  EVENT:  wait  @ time 3430.00
    VEHICLE Orion-Pickup > Begin ACTIVITY:  Waiting for Crew Transfer...
Predictate <Crew Transferred?> Satisfied

  EVENT:  descent burn  @ time 3430.00
    VEHICLE Orion-Pickup > Begin ACTIVITY:  Descent Burn Back to Earth

  EVENT:  landing  @ time 3450.00
    VEHICLE Orion-Pickup > Begin ACTIVITY:  Crew Landing on Earth
  TERMINAL EVENT named MISSION COMPLETE @ time 3451.0

COMPLETE


***DONE****
***********
```

Listing 12 Mars Simulation Output