An Integrated Software Architecture for Solar Cruiser Mission Design and Navigation

Jason Everett
NASA Marshall Space Flight Center
4600 Rideout Road
Huntsville, AL 35806
jason.m.everett@nasa.gov

Andy Heaton
NASA Marshall Space Flight Center
4600 Rideout Road
Huntsville, AL 35806
andrew.f.heaton@nasa.gov

Aaron Houin
NASA Marshall Space Flight Center
4600 Rideout Road
Huntsville, AL 35806
aaron.j.houin@nasa.gov

Kyle Miller
NASA Marshall Space Flight Center
4600 Rideout Road
Huntsville, AL 35806
kyle.miller@nasa.gov

Abstract – Solar Cruiser is a solar sailing mission, riding as a secondary payload to the Interstellar Mapping and Acceleration Probe (IMAP) mission, expected to launch in February of 2025. The Solar Cruiser vehicle will generate thrust via a complex, low-thrust solar sail. The extreme low-thrust nature of the solar sail will leave Solar Cruiser highly sensitive to external environmental effects (such as solar radiation pressure and high-order gravitational perturbations) throughout the entirety of flight. Because of this, preliminary & operational optimization routines must be intricately tied to high-order predictive propagation models to ensure the greatest possible confidence in mission success. The Solar Cruiser Mission Design and Navigation (MDNav) team has designed a software tool suite, employing the latest in software containerization technology, to accomplish this task, allowing for seamless development across several users and operating systems. Combining JPL's Monte toolkit with high-performance optimizers written by researchers at the University of Alabama, the proposed architecture allows for instant verification of optimized trajectories within the same development environment that the optimization takes place, removing the need for mission designers and navigators to switch between tools. The MDNav software suite itself is separated from the development and operational scripts to be used in flight, which allows for maintaining a low-footprint version control profile – thus avoiding unnecessary file bloating. This paper discusses the historical differences between previous iterations of the Solar Cruiser MDNav tool suite and the current iteration, planned operational interfaces of the tool with other software and subsystems, and the planned path forward in maintaining containerization services for the software throughout the lifetime of Solar Cruiser.

TABLE OF CONTENTS

1. INTRODUCTION

Over the last few decades, the roles of an aerospace engineer analyst/designer and a software developer have started to merge. Many methodologies and tools that have become standard in the software development industry have just begun to permeate into the engineering world to improve performance, efficiency, and user friendliness of software tools and models. This cross-pollination of knowledge across these two industries has allowed for aerospace engineering software developers to identify inefficiencies, problems, and non-standard software methodologies that have been previously maintained within the aerospace community without knowledge of alternatives.

As an example, extensive work has been underway on NASA's Program to Optimize Simulated Trajectories II (POST II) software suite to incorporate a more modularized software schema and to implement state-of-the-art software development practices, allowing the tool to be more user-friendly than its original interfaces [1]. Another example of software overhaul in the aerospace community can be seen in NASA's version 5 release of it's deep-space optimization system, Copernicus, which recently was converted from a standalone executable into a Miniconda/Anaconda Python environment – allowing users to customize plugins/modules much more efficiently [2].

Two significant areas of inefficiencies that this paper addresses pertain not directly to the internal architecture of individual software suites, but rather how aerospace software is implemented into different environments. These two inefficiencies are named the *environment* problem and the *dependency* problem, where the former addresses software compatibility with multiple operating systems (OSes), and the latter addresses operating system compatibility with multiple software suites. A concept called *software containerization*, which has been prevalent in the software community for over a decade, is introduced to mitigate these issues.

The Environment Problem

One of these areas of inefficiency in the aerospace software community pertains to the need for maintaining an up-to-date development environment (and operating system), while still

ensuring that heritage design and software tools remain compatible with these updated environments. It is often found in the aerospace engineering industry that a tool is developed so extensively in one operating environment that a conversion of that tool suite into another environment is an entirely dedicated (and usually costly) affair that requires significant attention. It often arises, either because of resource availability or software developer comfort, that a software tool is developed with specific OS-level dependencies, and that an update to a new OS breaks the software tool because of this dependency no longer being compatible with this new OS.

The eventual migration from a development environment to an operational environment of an aerospace software product is also affected by this compatibility issue. The operational phases of a spaceflight mission are, by nature, higher risk than the design phases of the mission. Operational software tools are typically required to be locked down to a specific version, where that version has been thoroughly verified across all functionality that is expected to be used in flight. Due to sometimes unfortunate resource availability circumstances, a user may not have direct access to the OS or environments that they will use within flight, which forces development in a different environment, and forces an extra level of planning to be purposefully outlined for conversion to operational use when preparing for launch. Once again, a bottleneck of implementation could cause schedule slips in software deliverables and strife among both the engineers designing the software, and the development & operations (DevOps) engineers in charge of maintaining and implementing this software in these environments.

The Dependency Problem

A similar problem can be categorized as the "opposite" of the problem above, which arises when several engineers plan on developing several different tool suites (all with their own use-case, build instructions, and dependencies) intended for *one specific operating machine*. This implies that a single operating machine must maintain all required software dependencies of each of the respective software tools, and that each of these software tools must be designed in such a way that they can maintain operational synchronicity when installed alongside one another. So, not only must a smart software tool suite be compatible for several different environments, but *also* must an operating environment be capable of maintaining and operating several different software tool suites simultaneously.

The left-hand diagram of Figure 1 displays several potential areas of "clashing" between software, dependencies, and OSes that arise from the two problems described above. The result is the need to maintain a very complex mapping of software organization, which leads to a large amount of overhead whenever a change is requested (a software update, dependency installation, or OS upgrade) on one of these operating systems.

The Dawn of Containerization

Historically, a significant portion of these issues have been mitigated by smart (yet time-consuming and costly) planning regimen, which involves:

- identifying potential operational environments and development environments to be used through the lifecycle of the software product,
- adhering to a standard dependency versioning through all software tools used in a specific OS/environment, and
- agreeing on a standard methodology for handling updates to operating systems, machines, software tools, and dependencies.

However, this "manual planning" leaves the human in the loop, which will almost certainly lead to (potentially costly) errors. There is currently no 100% efficient or accurate way

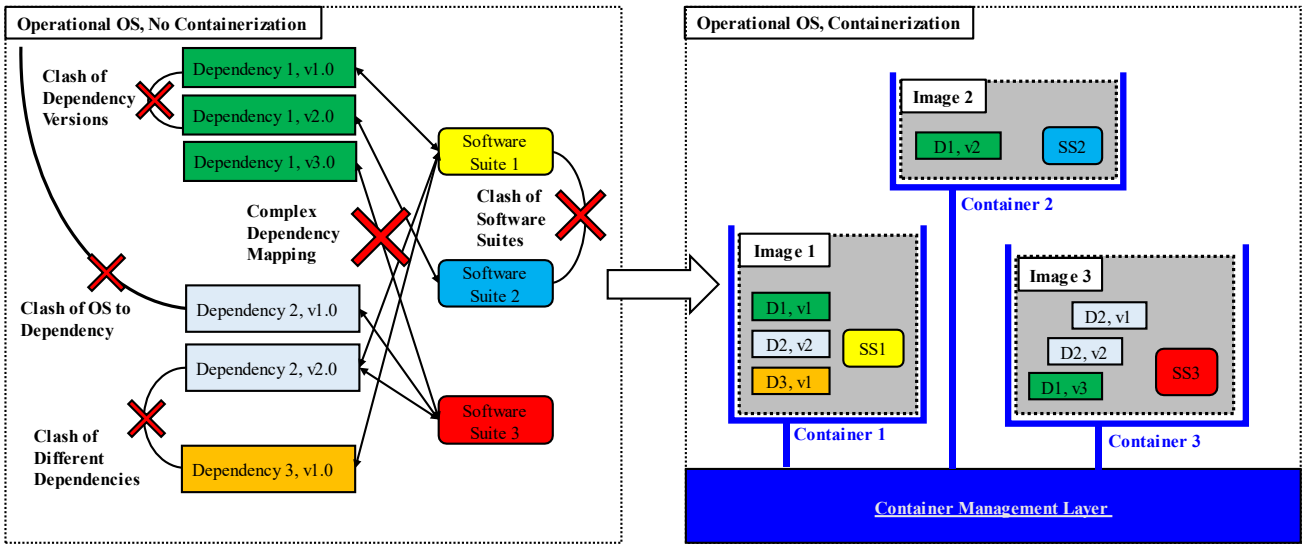

Figure 1. Operational Machines With vs. Without Containerization Technology

for a user, or a group of users, to track and maintain a list of all operational dependencies of a top-tier operational aerospace environment or a simulation tool suite without inevitably missing a dependency or a compatibility issue.

A solution to these problems, which has been popular in the software industry for a long time and has just recently permeated into the aerospace community, is the concept of software containerization. Containerization has been studied as a solution to the above problems on several different scales of computing in the science and engineering fields, from personal machines all the way to scalable high-performance machines [3].

The concept, in summary, pertains to ensuring that a software suite is completely "packaged" with everything the suite needs to run (dependencies/libraries, a low-level OS kernel, etc.), with the promise that the suite can be executed completely independent of the main operational environment, so long that said operational environment has the proper container management software installed. In this format, the machine that is running the containers is unaware of the dependencies and software suites running within the containers and needs only to focus on the management of the containers themselves. The only internal-to-external or external-to-internal communications that will occur through the container "boundary" are controlled and customized by the software developer that maintains the container.

The right-hand side of Figure 1 shows how containerization can address both the environment problem and the dependency problem described above. With a containerized environment, a developer can select a minimalistic set of dependencies that are required for the use or development of a simulation or tool suite, and as long as these dependencies are installed within the container, then any other operating machine that has installed the simple "container management" layer has the ability to run this container, even if the OS that the container was developed on differs from the OS of the target machine (the environment problem). Also, with this container management software layer installed on an operational computer, the computer then gains the ability to run several different software suites and simulation tools, all simultaneously, and all with their own sets of dependencies and requirements, as long as each of these software suites are running within their container boundaries (the dependency problem). Thus, both the environment problem and the dependency problem have been solved using containerization.

There are several other attractive components of current state-of-the-art containerization technology that are worthy of discussing in further detail, and two are mentioned frequently in this paper. The first is the level of simplicity of building up a container that contains all dependencies required for a specific simulation. Building up the instruction set of an *image* (an *image* can be thought of as a "template" of a container before it's spawned into life on an operational machine) is just as simple as installing dependencies in a normal fashion on a machine, with the only difference being that dependencies installed within an image, then "travel with" the image as it is distributed to team members. The second is the wide range of applicability of containerization to several different aspects of the aerospace industry. In the same fashion that simulation suites and software tool suites can be installed into an image for distribution, so can web servers, data bases, and any other object that has software representation on a computing machine. This means that a base "web server" image can be leveraged to rapidly prototype web pages that host simulation documentation, and these web pages can automatically be updated via Continuous Integration/Continuous Deployment (CI/CD) pipelines just as simply as a container can be spawned to run a simulation.

The concept of containerization changes the responsibility of operational machines in a productive manner. Previously, the role of an operational machine was to ensure that all dependencies for each software tool are completely compatible with one another and that each software installed on the machine does not interfere with each other software (a tedious affair!). If the above methodologies are followed in a program that employs several individually developed software tools, then the role of an operational machine becomes the requirement to manage the *containers*, rather than the software itself, which is in most cases simpler and more efficient.

2. THE SOLAR CRUISER MISSION

To orient the reader with the reasoning behind the different components of software selected for the Solar Cruiser mission design and navigation team, an overview of the mission is presented in this section. The Solar Cruiser mission is a 12-month technology demonstration mission that employs a spacecraft enabled with a solar sailing system, which is a system that nominally uses no propellant and is capable of generating thrust entirely through the application of solar radiation pressure (SRP) against a large "sail" structure. The maneuvering of the Solar Cruiser spacecraft through inertial space is accomplished by changing the orientation of the sail's "plane" with respect to the sun-relative vector. This orientation is generally described as a solar sail's Sun Incidence Angle (SIA), and is a key performance parameter when designing and executing solar sailing mission profiles.

The Solar Cruiser target operational orbit is a halo orbit around the Sun-Earth Lagrange-1 (L1) point, which has been selected to balance orbital stability and science opportunity [4]. Due to the unique nature of constant thrust generation made possible by the solar sail system, Solar Cruiser will be able to fly to, and operationally maintain, an "artificial" halo orbit that is shifted closer sunwards than a standard operational halo orbit flown by other missions (noted in this paper as a "Sub-L1" halo orbit). This puts Solar Cruiser at a unique vantage point to remain closer to the sun than possible by other conventional spacecraft, which tests a unique vantage point for never-before-possible scientific possibilities for future missions. Extensive analysis that covers several non-Keplerian orbits made possible only by solar sails can be found in [5].

Design Reference Mission

Figure 2 represents a visualization of Solar Cruiser's current Design Reference Mission (DRM) as the team approaches Preliminary Design Review (PDR). Solar Cruiser will launch as a secondary payload to the Interstellar Mapping and Acceleration Probe (IMAP), which is expected to launch no earlier than October 2024 on a SpaceX Falcon 9 rocket. After successful insertion into a ballistic trajectory by the launch vehicle, the Solar Cruiser vehicle will enter an approximate 4-month coasting period where the vehicle will maintain a cruising trajectory towards the nominal Sub-L1 insertion point and perform all adjustments and tracking necessary to prepare for solar sail deployment.

When the vehicle enters a designated "deployment window" defined by pre-flight analysis, Solar Cruiser will deploy its approximately 1700-square-meter solar sail and begin its transfer from the ballistic insertion trajectory onto a path to the target Sub-L1 halo orbit. Solar Cruiser is equipped with a unique set of reflectivity control devices (RCDs) that have the ability to change transmissivity of certain regions of the sail, which allow the sail to generate torque and slew to different orientations purely by solar radiation pressure alone.

Once Solar Cruiser merges onto a stable manifold leading to the target L1 halo orbit, the vehicle will begin an operational station-keeping phase that will prove this platform viable for long-term operational station-keeping on future missions. Afterwards, Solar Cruiser will begin performing a heliocentric plane change maneuver, wherein the vehicle will attempt to induce the greatest orbital plane change over the course of one month, proving plane change capabilities that will be crucial for future solar sailing mission profiles such as Solar Polar Imager (SPI) [6]. After successful demonstration of the inclination change maneuver, Solar Cruiser will begin decommissioning operations and inject into an end-of-mission heliocentric orbit.

Mission Specification

The Solar Cruiser vehicle itself is currently designed to be no more than approximately 100 kilograms of mass. If Solar Cruiser's sail deployment is properly executed, the sail is designed to demonstrate an acceleration of at least 0.12 mm/s^2 at 1.0 Astronomical Units (AU) from the Sun with the solar sail system fully deployed (Figure 3). For comparison, this equates to approximately 0.000012 Earth G's. Clearly, the Solar Cruiser vehicle is equipped with a radically low-magnitude and highly sensitive thrusting system and is planning on targeting a complex dynamical regime where multiple gravitational bodies act upon the vehicle at once. This calls for a required intricate connection to high-fidelity propagators and high-accuracy optimizers connected to one another.

Throughout the entirety of the mission timeline, ground-based tracking operations will be filtering out the true SRP forces impinged upon the spacecraft to verify mission requirements and to enable a best estimated state for planning future maneuvers. With specific filtering techniques applied to the ground-based orbit determination (OD) navigation filters, even off-nominal events (such as unexpected optical property shifts, improper sail deployments, etc.) can be detected to allow the mission operations team to take appropriate action during flight.

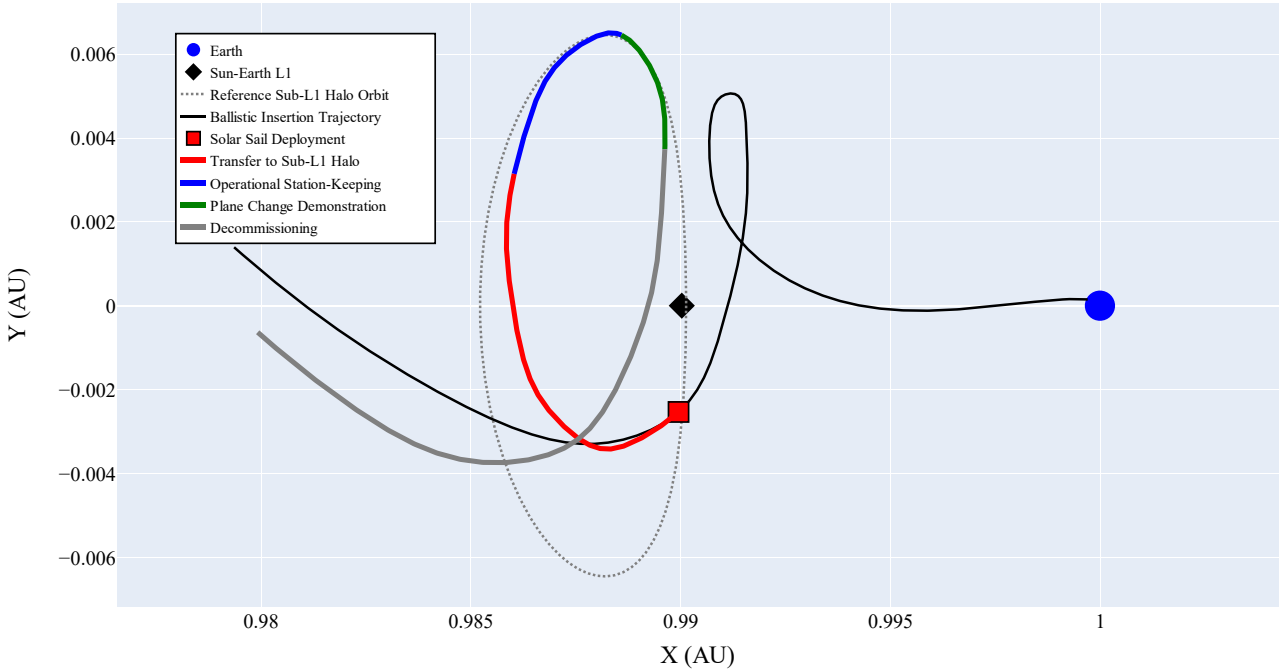

Figure 2. Solar Cruiser Design Reference Mission

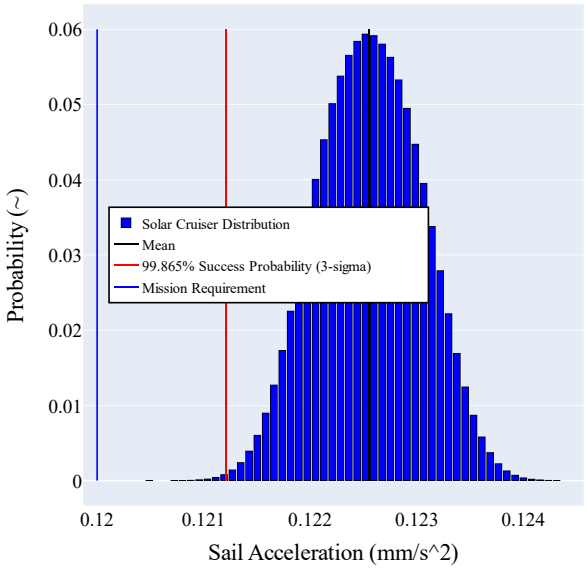

Figure 3. Solar Cruiser Acceleration Distribution

The Mission Design and Navigation Team

The Solar Cruiser Mission Design and Navigation (MDNav) team members wear multiple hats when it comes to the successful design and operational execution of the Solar Cruiser mission. During the design phases of the mission, the MDNav team is responsible for designing trajectories (and in general, mission solutions) that will enable Solar Cruiser to successfully complete all defined mission requirements in the allotted mission timeframe. During the operational phases of flight, the MDNav team will double as both an orbit determination team and a maneuver planning team, both filtering out a navigation solution based on incoming tracking data and subsequently designing new pointing histories to send to the Solar Cruiser attitude controls team to continue along a planned mission profile. The MDNav team is therefore in charge of software that can not only successfully detect very small magnitude SRP forces acting upon a deep-space vehicle, but to also design high fidelity trajectory solutions that enable flight of a very low thrust system in a complex dynamical region of space.

3. The Solar Cruiser Software Solution

The software architecture and development workflow of the Solar Cruiser Mission Design and Navigation (MDNav) tool suite has evolved through Solar Cruiser's preliminary design phases to become more helpful, efficient, and easy to use for all users who wish to use it, while ensuring that all of the issues identified in Section 1 are properly addressed. The tool suite is expected to change as the Solar Cruiser program nears launch, and the current architecture of the tool suite is designed to be flexible. The complexity and uniqueness of the mission design required for successful completion of this mission calls for an agile architecture to allow this tool suite to be used in several different environments.

The MDNav Tool Suite, Past Experiences

The Solar Cruiser MDNav tool suite did not originally employ containerization technology. The Solar Cruiser MDNav tool suite was originally a collection of tools completely independent from one another. Each tool was required to be installed separately, and the individual dependencies to be configured appropriately for each tool on each user's machine. The installation procedure was a time-consuming process, and each modification made to a component of the tools required sometimes an equally lengthy reconfiguration of dependencies and reinstallation of new versions of these tools. An analogy to the left half of Figure 1 for Solar Cruiser MDNav's *specific* predicament is shown in Figure 4.

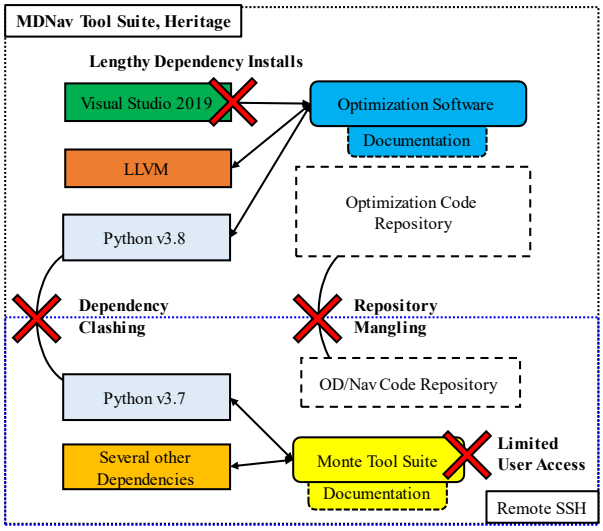

Figure 4. Solar Cruiser Heritage Software Architecture

The Solar Cruiser MDNav team utilizes JPL's Monte toolkit [7] for operational orbit determination and navigation, and utilizes a high-fidelity collocation optimizer written by the University of Alabama for trajectory optimization and mission generation routines. The original version of the collocation toolkit was only accessible/maintained on Windows operating systems. Monte, however, is developed and maintained strictly on Linux/RedHat machines. This immediately highlights the dependency problem identified in Section 1. There was no way for these two tools to communicate with one another on one computing machine, and the only communication possible between these two tool suites was by saving data to permanent memory storage and transferring from one OS to another OS.

The installation/configuration procedure for these two libraries required time and patience. The University of Alabama's optimization library was written in C++ and was compiled as a Python 3.x library (Python scripts are then how the optimizer is used). Compiling the original optimization library required a user to install all necessary dependencies of

the Microsoft Visual Studio suite (an installation of over 30 GB), as well as an Anaconda Python 3.x distribution that Visual Studio could then link to. With each individual user of the software potentially using a different minor version of Python (i.e. 3.6, 3.7, 3.8, etc.), and with each user potentially installing the Anaconda distribution at a different location on his or her local machine, the process of ensuring the Visual Studio solution was properly linking to the right version of Python was at times radically different for each individual user, with no way of developing consistent documentation that has the ability to address everyone's installation. This issue was partially mitigated by employing Windows environment variables in the Visual Studio solution and by having tool suite users create local environment variables that would link to the appropriate location of the Anaconda/Python install on their machines. Still, the process of creating and maintaining environment variables on a Windows machine is still a somewhat unfriendly process that a standard aerospace engineer should not have to deal with on a daily basis if using properly designed (and modern) software solutions.

The Monte v149 installation ships as an Anaconda environment itself. Monte was designed to work solely on RedHat Linux machines and requires a large list of dependencies before installation. A site-wide licensed version of Monte was installed on one of the operational RedHat Linux machines on NASA's MSFC's campus. For a user to be able to access Monte, it would require a user to apply for access to this individual Linux machine, the approval process of which took up to a week to finalize at times. After access was granted, the user gains limited access to the pre-installed Monte Anaconda environment on the machine, but only through an SSH connection into the Linux machine (yet another configuration process the user must complete). As one singular Monte instance was installed on the Linux machine, standard users only gained access to basic features of Monte and did not have control over sometimes critical customization features to improve process workflow. For example, if a user would like to install another Python library into the Monte Anaconda instance, he/she would have to request administrator approval, even if the library was something as trivial as a simple plotting or I/O library.

Monte documentation, which is a set of static HTML files that ships with Monte and is critical for understanding and learning the tool, was also somewhat complex for the user to access. The user could either download the HTML files from the Linux server on a local machine with a compatible web browser or apply for access to JPL's public-facing documentation website (yet another process that could take up to a week). Even then, the public-facing documentation website currently does not display the documentation from the version of Monte that MSFC currently maintains.

After all configuration and installation is complete, the process of generating and managing operational/analysis scripts still needs to be resolved. A set of optimization-related scripts were managed under source control in the same code repository that the optimization tool itself was stored. Then, a second set of operational scripts were stored under a different source control repository for orbit determination and navigation related activities that are ran in the Monte toolkit. This means that a user is also required to stay in-sync with two different repository code-bases, the data within which was also at risk of being repeated and duplicated across repositories without awareness of the current data state of the files (such as input text files, configuration files for plotting, etc.).

Thus, the process workflow for the original MDNav set of tools was clearly cumbersome, and while it served its purpose for the proposal and early design phases of Solar Cruiser, was not suitable for operational use. A new change was required to ensure that the tool suite that the MDNav team uses is suitable and efficient in an operational environment.

The MDNav Tool Suite, Current Architecture

As an analogy to the right half of Figure 1, Figure 5 displays the difference between the original MDNav tool suite (Figure 4) and the updated tool suite that is being constructed for operational use. The updated Solar Cruiser MDNav tool suite currently utilizes Open Container Initiative (OCI)-compliant containerization technology, described in Section 1 above, to help with dependency mitigation, simplicity of operational deployment, and ease-of-use.

At the core of the MDNav software suite is Docker [8], which is a container-level manager that allows a user to install individual *images*, instantiate images into *containers*, and manage/modify/close containers as desired. Other OCI-compliant container managers (such as Podman) were also tested, but evidently Docker was the most applicable and most accessible tool for users across all operating systems. Docker is compatible with Windows, MacOS, and Linux, which means that, if the user has access to a singular Docker image (which contains all relevant tools and dependencies), the user could immediately open the image into a container and begin development with absolutely no external dependency installation other than just installing Docker (a user-friendly process for all operating systems that Docker is compatible with).

An OCI-compliant *image* can be thought of as an instruction set which collects and compiles a set of dependencies and tools into a singular binary file. This image, if shared across groups, users, or operating systems, can then be loaded into the local container management system (Docker), and can be instantly instantiated into a *container*. The entire Solar Cruiser MDNav tool suite is currently distributed as two separate images: a main development image, and an auxiliary documentation image.

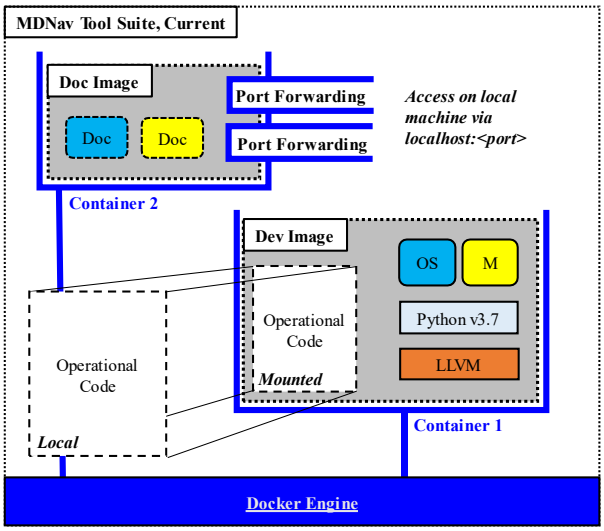

Figure 5. Solar Cruiser Updated Operational Software Architecture

The main development image contains Monte, all its dependencies, the UA optimization tool suite, and all its dependencies. To support the evolution of the MDNav tool suite, the UA software development team was able to generate and updated version of the optimization software that could be compiled onto Linux machines for a variety of Python major and minor versions. A version of the optimization tool suite was compiled to match the same minor version (and the same Linux OS version) of Python that v149 Monte currently operates under. Within the Docker image instruction set, the optimization tool suite is installed into the same Anaconda environment that Monte ships with. Thus, one of the main issues of the original MDNav tool suite (developing in two different operating systems) has been resolved.

It is important to note that all the image and instruction set configuration mentioned above is completely shielded from the user. Once a working Docker image is constructed and compiled, this image, containing all relevant tools and dependencies, can easily be passed to different team members in a "pre-built" configuration. The procedure to load in the image into the user's local Docker container management system is as simple as a one-line, 20-character console command. After which, the user will have complete access to the tool suite and will not have to worry about dependency re-installations or OS updates breaking the development environment – so long as Docker is installed on the client machine.

Another powerful component of containerization is the ability to "mount" a specific set of files into the container once the container is instantiated. This mounting of files allows a user to "mirror" a set of files from a local machine into a running container, thus giving the container access to modify and manipulate any files within this mounted directory. This means that if a user has a set of files/scripts on a local machine, but these scripts require tools and dependencies that are installed within a container to run, the user has the ability to give a running container access to these scripts in order to run them. Any changes made from within the container to this mounted directory will immediately be replicated/mirrored on the local machine.

Therefore, made possible from this capability, the Solar Cruiser MDNav team stores a set of minimal-footprint operational scripts to a centralized repository, and then mounts these scripts into a running container on an operational machine when the scripts are ready to be run. This format allows for minimal file bloating in code repositories (all the static data and tools are stored within the image – only the actual modifiable scripts are committed to the source repository).

The auxiliary image of the Solar Cruiser MDNav tool suite contains all the documentation necessary to learn and utilize both Monte and the optimization toolkit. Within the auxiliary image, two minimalistic web servers are installed on different internal ports that contain both the Monte documentation HTML files and the optimization software documentation HTML files. Utilizing Docker container's port-forwarding features, the user could then access either documentation set on a machine by opening a browser and navigating to http://localhost:<port>, where <port> represents the relevant forwarded port to each of the documentation servers.

When it comes to updates of either Monte or the optimization tool suite, the user does not need to re-install or re-configure anything. Although the owner of the Docker instruction set would have to reconfigure the new version of the image containing the updated software, this newly built image can then be immediately deployed to all users and operating systems of the tool suite, and users can continue work seamlessly. As mentioned above, updating the instruction set to contain new dependencies or modifications is just as simple as it would be to update the local machines with these updates. However, with containerization technology, these updates and reconfigurations need only occur once, and then can be easily distributed to users instantaneously.

The new version of the Solar Cruiser MDNav tool suite is now approaching a state where operational use of these tools can more easily be accepted. Installation procedures now turn into a simple install of Docker and a download of relevant images. The Solar Cruiser MDNav User's Guide can now officially be written in documentation form rather than purely existing as a collection of knowledge from trial and error on a variety of different machines and operating systems.

4. Extended Implications

Now that the Solar Cruiser MDNav tool suite is OCI-compliant, a wide variety of different use-cases can be employed with very low effort. For example, if an update occurs to one of the Linux machines that would require a re-installation of Monte into Linux, instead of installing Monte's dependencies and configuring access to each user on

the Linux machine, Docker can be installed onto the Linux machine and the MDNav tool suite images would instantly become compatible with the Linux machines without any futher intervention. This makes any planned or unplanned updates to Solar Cruiser's operating systems a non-factor in ensuring compatibility with the actual operational MDNav software.

If the MDNav team would like to eventually centralize reports or design outputs to a broader audience, an image could be constructed that contains a minimalistic web server. This image, when installed on an MSFC-available Linux machine, could be executed as a container, and a local directory containing documentation code could be mounted into said container. Port forwarding could then be used within the container to allow users across NASA to instantly view the reports or documentation. Updating the contents of the web server, then, simply involves updating the local version of the mounted documentation files. This documentation update could even be automated through a CI/CD pipeline. After the files have been updated, all requested users would instantly gain access to the updated documentation files by accessing the same link, because of the nature of the mounted repository.

If other complex mission design or navigation tools are required in future design phases or operational phases of Solar Cruiser, they could either be added to the same development image that has already been created, or added to a new image entirely and distributed to the team. As relationships between containers become more complex, automatic container management scripting language (such as Docker Compose [9]) would allow a user to instantly instantiate and manage multiple Docker containers at once without worrying about individual container management.

5. CONCLUSION

As new methodologies in the software engineering community continue to permeate into the aerospace community, more and more unique solutions will become common that will solve age-old inefficiencies across the software design lifecycle of aerospace projects. The Solar Cruiser MDNav team has proposed an architecture that solves several of these inefficiencies that relate to operational dependency mismatches and compatibility issues between several different computing machines. The Solar Cruiser MDNav tool suite will continue to mature as the Solar Cruiser mission quickly approaches flight. By utilizing the power of containerization technology, lengthy migration processes that previously would have had to be planned for (and purchased) can now be eliminated from the scheduling process.

The containerization technology demonstration of the Solar Cruiser tool suite is already being adopted by other simulations and tool suites within NASA's Marshall Space Flight Center. This will help contribute to getting a large range of users easy access to industry leading tools across the aerospace engineering industry.

REFERENCES

[1] Lugo, R, et. al., "Launch Vehicle Ascent Trajectory Simulation using the Program to Optimize Simulated Trajectories II (POST2)", 2017 AAS GNC Conference, Breckenridge, CO, AAS 17-274.

[2] Murri, D., et. al., "Improvements to the Copernicus Trajectory Design and Optimization System for Complex Space Trajectories", 28 February 2019, NASA/TM 2019-220247.

[3] J. S. Hale, L. Li, C. N. Richardson and G. N. Wells, "Containers for Portable, Productive, and Performant Scientific Computing," in Computing in Science & Engineering, vol. 19, no. 6, pp. 40-50, November/December 2017, DOI 10.1109/MCSE.2017.2421459.

[4] Pezent, J. B., et. al., "Preliminary Trajectory Design for NASA's Solar Cruiser: A Technology Demonstration Mission", Jun2 2021, Acta Astronautica, Volume 183, DOI 10.1016/j.actaastro.2021.03.006.

[5] McInnes, Collin, "Solar Sailing: Technology, Dynamics and Mission Applications", 1999, DOI 10.1007/978-1-4471-3992-8

[6] Thomas, Dan, et. al., "Solar Polar Imager Concept", 16 November 2020, AIAA Ascent Virtual Conference, DOI 10.2514/6.2020-4060

[7] Smith, J., et. al., "MONTE Python for Deep Space Navigation", 2016, 15th Python in Science Conference.

[8] Boettiger, Carl. "An introduction to Docker for reproducible research." ACM SIGOPS Operating Systems Review 49.1 (2015): 71-79.

[9] Docker Compose. https://docs.docker.com/compose/

BIOGRAPHY

Jason Everett *received a B.S. in Aerospace Engineering from The Pennsylvania State University in May 2018. Since his start in civil service at NASA's Marshall Space Flight Center, he has been working as a full time GNC engineer for several NASA projects including the Mars Ascent Vehicle, the Space Launch System, two solar sail missions, and two lunar lander designs. In September 2018, he received a Center Director's Special Service Award by supporting standalone GNC implementation for SLS critical liftoff and tower clearance assessments. He currently serves as the Mission Design and Navigation Lead for the Solar Cruiser project and supports DevOps and GNC development for both Solar Cruiser and the Mars Ascent Vehicle.*

Andy Heaton *has been employed by NASA since 1990. Since 2002, he has served in increasing positions of responsibility for solar sail propulsion technology development and flight missions. From 2002 to 2006 he was the GNC lead for the Solar Sail In-Space Propulsion Project. From 2010 to 2013, Mr. Heaton supported a number of solar sail mission proposals, which culminated in the selection of NEA Scout in 2013, for which Mr. Heaton is the Guidance and Control lead. Mr. Heaton currently serves as both the Mission Design and GNC Team Lead for Solar Cruiser.*

Aaron Houin *received a B.S. from Purdue University in 2018 and is currently a graduate student in the Astrodynamics and Space Research Laboratory under Professor Rohan Sood at the University of Alabama. He worked as a GNC engineer at Dynetics Inc. from 2018 until joining the NASA Marshall mission design team in January of 202. At NASA his work consisted mostly of low energy and multi-body trajectory design applied to manned and robotic spaceflight missions. He currently serves as the mission design team lead on the upcoming MoonBeam smallsat mission proposal.*

Kyle Miller *received his B.S. in Aerospace Engineering from Embry-Riddle Aeronautical Univ. (2017). He joined NASA's Marshall Space Flight Center in 2018. While at NASA, he has worked on several projects in the areas of Mission Design and Guidance, Navigation, and Control. Recently, his work has focused on Lunar descent and surface navigation, as well as solar sail mission design, navigation, and momentum management.*