Revolutionizing Space Mission Event Modeling with the Tychonis Framework

Thesis
submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in Computing

by
Marcel Llopis

Computer Science Department
Universitat Politècnica de Catalunya

October, 2023

PREFACE

NASA's Jet Propulsion Laboratory (JPL) is a cutting-edge research center that specializes in space exploration and development of space technologies. Based in Pasadena, California, it has been at the forefront of numerous missions that have revolutionized our understanding of the universe.

One such mission was the Dawn mission, which aimed to explore two of the largest objects in the asteroid belt, Vesta and Ceres. As a software developer for science planning software within the Dawn mission, I was part of a team of brilliant scientists and engineers who accomplished significant milestones in space exploration. The Dawn mission allowed us to study these massive bodies, providing insights into the formation and evolution of the solar system.

In my subsequent role at JPL, I led a team of engineers who created reusable tools for simulating spacecraft behavior at different levels to aid in mission planning. Drawing from my background as a science planning software developer and my growing understanding of multi-mission software, I recognized an opportunity to research and design a software-based framework for modeling geometric events in space. Such a framework would not only reduce the risk and cost of future missions but also enable a more accessible means of operating with space geometry.

Motivated by this newfound passion, I set a goal to pursue a PhD in this area of study. Years of diligent research and collaboration with Prof. Xavier Franch and Prof. Manel Soria enabled me to comprehensively develop and articulate the concepts presented in this thesis. It brings me immense pleasure to share these ideas with you.

Publications

During the doctoral program, the candidate authored and subsequently published the following scholarly works, which have contributed significantly to the research presented in this thesis.

1. M. Llopis, C. A. Polanskey, C. R. Lawler, and C. Ortega, "The Planning Software Behind the Bright Spots on Ceres: The Challenges and Successes of Science Opportunity Analyzer," in 2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT), 2019, pp. 1–8. doi: 10.1109/SMC-IT.2019.00005.
2. M. Llopis, M. Soria, and X. Franch, "Tychonis: A model-based approach to define and search for geometric events in space," Acta Astronautica, vol. 183, pp. 319–329, 2021, doi: 10.1016/j.actaastro.2021.01.057.
3. M. Llopis, X. Franch, and M. Soria, "Integrating the Science Opportunity Analyzer with a Reusable Opportunity Search Framework," in ASCEND 2020, doi: 10.2514/6.2020-4221.
4. M. Llopis, X. Franch, and M. Soria, "Assessing the Usability of Two Declarative Programming Languages to Model Geometric Events," Journal of Aerospace Information Systems, pp. 1–9, Apr. 2023, doi: 10.2514/1.I011207.
5. P. Betriu, M. Soria, J. L. Gutiérrez, M. Llopis, and A. Barlabé, "An assessment of different relay network topologies to improve Earth–Mars communications," Acta Astronautica, vol. 206, pp. 72–88, May 2023, doi: 10.1016/j.actaastro.2023.01.040.

Acknowledgements

It is with the utmost pleasure and amusement that I extend my heartfelt gratitude to my beloved daughter, Nia. You are always a relentless source of inspiration and motivation, keeping me on my toes with your cheerful and inquisitive nature. Your boundless enthusiasm infuses me with a sense of joy and purpose, and your questions constantly challenge me to think deeper and broader. Together, we have shared countless moments of warmth and comfort, snuggled up on the couch, with me clacking away on my computer, trying to complete my experiments or hammer out a paper. Balancing the rigors of being a dad, a researcher, and a working professional has not been easy, but your love and support have made all the difference. Thank you, my dear daughter, for being my rock, my sunshine, and my unknowing co-conspirator in this madcap academic adventure.

It is with great pleasure and a good chuckle that I express my deepest thanks to Xavi and Manel. These two outstanding gentlemen have been a constant source of assistance, encouragement, and insight, guiding me through the peaks and valleys of academia with their friendship and wise counsel. Our conversations have been so enlightening and enjoyable that I sometimes forget about the miles that separate us. While we may live far apart, I hope that our bond will only get stronger with time, and that we will keep meeting, ideally in person, for a series of well-deserved lunches and dinners. Thank you for putting up with my erratic schedule, and for always being willing to lend an ear and offer words of encouragement. I'm truly fortunate to have crossed paths with such fine human beings.

Agraïments

És amb el màxim plaer i amor que expresso el meu sincer agraïment a la meva estimada filla, Nia. Ets sempre una font incansable d'inspiració i motivació, mantenint-me alerta amb la teva naturalesa alegre i inquisitiva. El teu entusiasme sense límits em transmet una sensació de joia i propòsit, i les teves preguntes em desafien constantment a pensar de manera més profunda i ampla. Junts, hem compartit moments innumerables de calidesa i confort, recolzats al sofà, mentre jo clicava al meu ordinador, intentant completar els meus experiments o redactant un article. Equilibrar les exigències de ser pare, investigador, i professional no ha estat fàcil, però el teu amor i suport ho han fet tot possible. Gràcies, estimada filla, per ser la meva roca, el meu sol, i la meva còmplice involuntària en aquesta bogeria d'aventura acadèmica.

És amb gran plaer i una bona rialla que expresso el meu més profund agraïment a en Xavi i en Manel. Aquests dos senyors excepcionals han estat una font constant d'ajuda, encoratjament i visió, guiant-me a través dels cims i les valls de l'acadèmia amb la seva amistat i saviesa. Les nostres converses han estat tan il·luminadores i agradables que a vegades oblidava els quilòmetres que ens separen. Tot i que vivim lluny, espero que el nostre vincle només es faci més fort amb el temps, i que continuem trobant-nos, idealment en persona, per a una sèrie de dinars i sopars ben merescuts. Gràcies per suportar el meu horari erràtic, i per estar sempre disposats a escoltar i oferir paraules d'encoratjament. Realment tinc la sort de haver creuat el camí amb éssers humans tan excel·lents.

To Nia

Abstract

This thesis presents the Tychonis framework, a solution to the limitations faced by existing software packages used in space missions to identify geometric events. The framework is designed to integrate with current and future mission software, providing users with the ability to extend opportunities and search algorithms without modifying the tools that use it. The framework is built on the principles of separation of concerns, extensibility, reusability, and independent verification and validation. It is provided as a software library, allowing missions to add their own data structures and constructs, promoting cross-mission reusability. The text also includes a case study of Tychonis' integration with the Science Opportunity Analyzer (SOA) software, demonstrating its ability to be extrapolated to other tools that need to search for geometric events.

As a complement to the Tychonis framework, the thesis introduces two computer languages designed to make the process of modeling opportunities more accessible. Scientists, who may face difficulties with imperative programming languages or lack of available science planning tools, are the target audience for these languages. The readability and usability of the languages were evaluated through a comprehensive study involving a questionnaire with active exercises, statements with corresponding responses on a Likert scale, and open-ended questions to elicit qualitative responses. The results provide both relative and absolute quantification of the usability and readability of each language, as well as qualitative results to direct future language design decisions.

Emphasizing the importance of utilizing proven software principles and good design choices in space missions helps reduce risk and cost. Tychonis and our research on accessible computer languages embrace this concept.

RESUM

Aquesta tesi presenta el *framework* Tychonis, una solució a les limitacions que enfronten els paquets de programari existents utilitzats en les missions espacials per identificar esdeveniments geomètrics. El *framework* està dissenyat per integrar-se amb el programari de missió actual i futur, proporcionant als usuaris la capacitat d'estendre les oportunitats i els algoritmes de cerca sense modificar les eines que el fan servir. El *framework* es basa en els principis de separació d'interessos, extensibilitat, reutilització, i verificació i validació independents. Es proporciona com a biblioteca de programari, permetent que les missions afegiu les seves pròpies estructures de dades i construccions, promocionant la reutilització entre missions. El text també inclou un estudi de cas de la integració de Tychonis amb el programari Science Opportunity Analyzer (SOA), demostrant la seva capacitat per ser extrapolada a altres eines que necessiten buscar esdeveniments geomètrics.

Com a complement a Tychonis, la tesi presenta dos llenguatges de programació dissenyats per fer més accessible el procés de modelització d'oportunitats. Els científics, que poden trobar dificultats amb els llenguatges de programació imperatius o la falta d'eines de planificació científica disponibles, són el públic objectiu d'aquests llenguatges. La llegibilitat i la usabilitat dels llenguatges s'han avaluat a través d'un estudi exhaustiu que inclou un qüestionari amb exercicis actius, afirmacions amb respostes en una escala Likert, i preguntes obertes per obtenir respostes qualitatives. Els resultats proporcionen quantificació relativa i absoluta de la usabilitat i la llegibilitat de cada llenguatge, així com resultats qualitatius per dirigir les decisions de disseny de futurs llenguatges.

Posar èmfasi en la importància d'utilitzar bons principis de programació i bones eleccions de disseny en les missions espacials ajuda a reduir el risc i el cost. Tychonis i la nostra investigació sobre llenguatges de programació accessibles implementen aquest concepte.

Resumen

Esta tesis presenta el *framework* Tychonis, una solución a las limitaciones que enfrentan los paquetes de software existentes utilizados en las misiones espaciales para identificar eventos geométricos. El *framework* está diseñado para integrarse con el software de misión actual y futuro, proporcionando a los usuarios la capacidad de extender las oportunidades y los algoritmos de búsqueda sin modificar las herramientas que lo utilizan. El *framework* se basa en los principios de separación de intereses, extensibilidad, reutilización y verificación y validación independientes. Se proporciona como biblioteca de software, permitiendo que las misiones agreguen sus propias estructuras de datos y construcciones, promoviendo la reutilización entre misiones. El texto también incluye un estudio de caso de la integración de Tychonis con el software Science Opportunity Analyzer (SOA), demostrando su capacidad para ser extrapolada a otras herramientas que necesitan buscar eventos geométricos.

Como complemento a Tychonis, la tesis presenta dos lenguajes de programación diseñados para hacer más accesible el proceso de modelado de oportunidades. Los científicos, que pueden encontrar dificultades con los lenguajes de programación imperativos o la falta de herramientas de planificación científica disponibles, son el público objetivo de estos lenguajes. La legibilidad y la usabilidad de los lenguajes se han evaluado a través de un estudio exhaustivo que incluye un cuestionario con ejercicios activos, afirmaciones con respuestas en una escala Likert y preguntas abiertas para obtener respuestas cualitativas. Los resultados proporcionan una cuantificación relativa y absoluta de la usabilidad y la legibilidad de cada lenguaje, así como resultados cualitativos para dirigir las decisiones de diseño de futuros lenguajes.

Poner énfasis en la importancia de utilizar buenos principios de programación y buenas elecciones de diseño en las misiones espaciales ayuda a reducir el riesgo y el costo. Tychonis y nuestra investigación sobre lenguajes de programación accesibles implementan este concepto.

Impact Statement

The Tychonis framework and the two computer languages introduced in this thesis have significant potential to impact the space industry by providing a solution to the limitations faced by existing software packages used in space missions to identify geometric events.

The Tychonis framework's integration with current and future mission software, along with its principles of separation of concerns, extensibility, reusability, and independent verification and validation, provide a powerful change for space missions to search for geometric events. This is particularly important for organizations like NASA, where the accurate identification of geometric events is essential not only for mission planning, but also for spacecraft navigation and scientific data analysis.

NASA has a strong tradition of reusing software and technology from mission to mission to build on previous successes and ensure mission heritage. The Tychonis framework's focus on cross-mission reusability, coupled with its adherence to proven software principles, can help ensure that software developed for one mission can be used in future missions with minimal modifications, reducing the risk associated with software development and validation for each individual mission. This approach can lead to increased mission success rates and cost savings, making it a valuable contribution to the space industry. Additionally, our vision for textual computer languages to model geometric events can enable more scientists to participate in mission planning and execution, leading to a more diverse and inclusive workforce.

Declaració d'impacte

Tychonis i els dos llenguatges de programació presentats en aquesta tesi tenen un important potencial per influenciar la indústria espacial. Les nostres idees proporcionen una solució a les limitacions dels paquets de programari existents utilitzats en missions espacials per identificar esdeveniments geomètrics.

La integració de Tychonis amb programari actual i futur de les missions, juntament amb els seus principis de separació d'interessos, escalabilitat, reutilització, i verificació i validació independents, ofereix un canvi significatiu per a les missions espacials en la cerca d'esdeveniments geomètrics. Això és particularment important per a organitzacions com la NASA, on la identificació precisa dels esdeveniments geomètrics és essencial no només per a la planificació de la missió, sinó també per a la navegació de les naus espacials, i l'anàlisi de dades científiques.

La NASA té una forta tradició de reutilització de programari i tecnologia de missió a missió per donar continuïtat a èxits demostrats. L'enfocament de Tychonis en la reutilització transversal de les missions, juntament amb la seva adhesió a bons principis de programació, pot ajudar a garantir que el programari desenvolupat per a una missió es pugui utilitzar en futures missions amb modificacions mínimes, reduint el risc associat al desenvolupament i validació de programari per a cada missió individual. Aquest camí porta a un augment de les taxes d'èxit de la missió i estalvis de costos, fent-ho una contribució valiosa per a la indústria espacial. A més, la nostra visió dels llenguatges informàtics textuals per modelar esdeveniments geomètrics pot permetre que més científics participin en la planificació i execució de les missions, ajudant a tenir equips més diversos i inclusius.

Contents

1 Introduction

For millennia, humanity has been captivated by the celestial patterns formed by the stars in the sky. Our ancestors keenly observed the movements of the stars throughout the night, days, and seasons, seeking to gain knowledge of these celestial lights in the vast expanse of the Universe. From drawings on the walls of caves in Europe dating back 12,000-40,000 years ago, to the Ancient Egyptians who marveled at two bright stars in the Northern skies they referred to as *the Indestructibles*[1], the human fascination with the cosmos has been persistent. The Greek astronomers, who coined the term *astronomy*, made remarkable advancements through observations, deducing that the morning and evening stars were the same body we now know as Venus, and even discovering that Earth was round through the study of eclipses. Tycho Brahe, in the late 1500s, made significant contributions to positional astronomy, charting the positions of over 777 stars within 45 constellations and developing the Tychonic System, as depicted in Figure 1, leaving a lasting legacy of perseverance, systematic record-keeping, and a passion for the stars. Today, the dream of exploring the cosmos continues to inspire individuals of all ages, who envision themselves as the scientists and engineers of tomorrow, making groundbreaking discoveries in the great unknown. As we stand on the brink of sending humans to Mars and beyond, it is truly a remarkable time to be alive and witness the next chapter in our quest for understanding space.

[1] In addition, while the Greeks identified 15 stars as Indestructibles, different cultures throughout history have identified different sets of stars as fixed or eternal in the sky, demonstrating the cultural variability in our understanding and interpretation of the stars.

As humankind continues to explore the vast expanse of the universe, we have found new and innovative ways to gather information about the cosmos. While we may not have physically gone beyond the Earth-Moon system ourselves, we have sent robotic spacecraft to do the job for us. Equipped with a variety of instruments, these machines have given us a glimpse into the mysteries of the universe and allowed us to continue advancing our understanding of it. These instruments can either be active or passive, depending on the type of data they are designed to gather. Passive instruments such as magnetometers or visible-spectrum cameras simply collect information that is present in their environment, while active instruments like radar emit energy in order to gather data.

Figure 1. The Tychonic System. Tycho Brahe's model of the Universe, as presented in the 1661 edition of Andreas Cellarius' Harmonia Macrocosmica, offers a notable insight into the conceptualization of the cosmos in the seventeenth century. This illustration captures Brahe's proposal of a middle ground between the geocentric model of Ptolemy and the heliocentric model of Copernicus, where the earth takes center stage, and the Sun and Moon orbit around it, while the other planets revolve around the Sun.

The role of these robotic missions goes beyond simply observing the universe. By analyzing the data collected by these machines, we can learn about astrophysics, chemistry, and the geology of celestial bodies. They allow us to delve deeper into the mysteries of space, and provide us with the information we need to one day, hopefully, make human space exploration and colonization a reality. It is like a pre-flight check, so to speak, where we gather information about the environment we plan to explore before we venture into it ourselves. The true beauty of these missions is that they allow us to explore the universe and learn from it without having to leave the comfort of our planet, and they provide us with a wealth of information that continues to expand our knowledge of the cosmos.

The field of robotic space exploration emerged in the context of the Cold War, where the United States and the Soviet Union engaged in a competitive space race to demonstrate their technological prowess and military strength to one another and the world. This competition led to the launch of several groundbreaking missions, such as the Soviet Union's launch of Sputnik 1, the first Earth-orbiting satellite, in 1957, and the United States' launch of Explorer 1, the first satellite equipped with scientific instruments, in 1958. The earliest missions beyond the Earth-Moon system were part of the Mariner program, executed by NASA and the Jet Propulsion Laboratory (JPL) in Pasadena, California. Notable missions of the Mariner program include Mariner 2's Venus fly-by in 1962 and Mariner 4's Mars fly-by in 1965, which captured the first close-up images of another planet.

The aforementioned endeavors, although intricate, pale in comparison to JPL/NASA's Voyager program, which commenced in 1977 and marked the world's first multi-planet science-oriented flagship-level spacecraft launch. The initiative featured two identical spacecraft that capitalized on a rare planetary alignment that occurs every 175 years, thereby permitting both probes to transit effortlessly from one planet to the next in a meticulously orchestrated sequence[2],

[2] Both Voyager 1 and 2 used gravitational slingshot maneuvers around Jupiter to reach Saturn, Uranus, and Neptune. Voyager 1 reached interstellar space in 2012, while Voyager 2 continued its

as depicted in Figure 2. One can scarcely fathom the engineering intricacy of this undertaking, given the technological and computational limitations of the era, notably in the then-emergent discipline of orbital mechanics and the 11 sophisticated instruments equipped on each spacecraft.

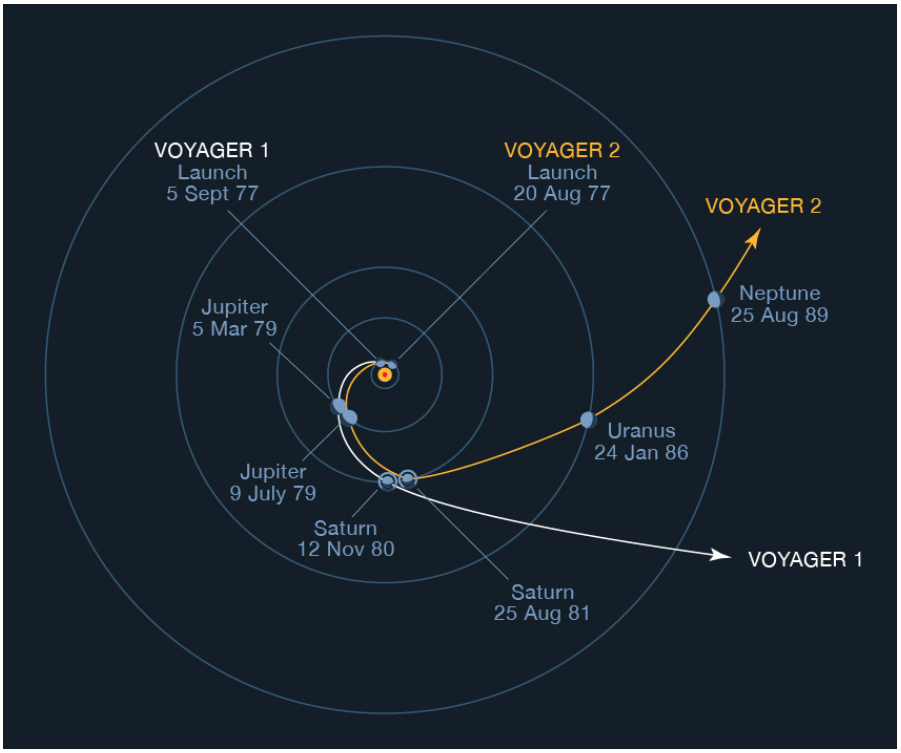

Figure 2. Trajectories of Voyager 1 and 2. This image, which showcases the intricate and detailed pathways traversed by the Voyager missions, highlights the remarkable scientific achievements and innovative technological advancements that have been instrumental in advancing our understanding of the universe. Image credit: NASA/JPL-Caltech.

What is all the more remarkable is that, as of this writing in 2023, after 45 years of tireless operation, the two Voyager spacecraft remain operational and are the most remote man-made objects in existence, having traversed the interstellar void beyond our solar system. While these venerable probes may not endure much longer, they continue to furnish invaluable data that enhances our comprehension of the cosmos. It is poignant to reflect that, from an engineering standpoint, robotic missions have undergone a metamorphosis from being revolutionary to

exploration of the outer solar system with a gravity assist from Neptune. Both spacecraft are still operational as of 2023 and continue to send valuable data back to Earth.

evolutionary, as exemplified by the Voyager missions. Today, the reader may readily discern more recent landmarks in the realm of audacious robotic spacecraft missions, such as Cassini, New Horizons, Mars Science Laboratory, Dawn, Juno, Mars 2020, and many others[3].

The operational management of space missions has experienced a gradual evolution, though its foundational principles remain steadfast. As the field of space mission management has grown and matured, it has given rise to well-established processes. Among the most crucial considerations in the design and management of a space mission is the identification of key geometric events. These events are vital not only to mission planning, but also to the adaptation of the spacecraft and its surroundings to changing conditions as it traverses through space. The process of determining the timing of these events, referred to as *Opportunity Search*[4], is the focal point of our investigation. Although this area has seen significant advancements over time, it has yet to be fully realized as a self-contained software capability. In the forthcoming sections, we shall explore the intricacies of Opportunity Search, preceded by a comprehensive overview of the fundamental principles of space mission management.

[3] Cassini explored the planet Saturn and its moons, while New Horizons visited Pluto and is now exploring the Kuiper Belt. Mars Science Laboratory, which includes the Curiosity rover, is exploring the surface of Mars, while Dawn explored the dwarf planet Ceres and the asteroid Vesta. Juno is currently orbiting Jupiter, while Mars 2020 is exploring Mars with the Perseverance rover and the Ingenuity helicopter. These missions have greatly expanded our understanding of the solar system and have provided valuable insights into the processes that shaped our planet and others.

[4] Opportunity search is a process of searching for opportunities or windows of time during which a spacecraft can perform a particular task or observation. It is an important aspect of mission planning in space exploration, as it helps scientists and engineers optimize the use of a spacecraft's resources and achieve mission objectives. Opportunity search can involve identifying periods of time when the spacecraft is in the optimal position relative to the target, such as when it is at the right distance or angle for a flyby, or when the target is in the right position relative to the sun or other celestial bodies.

1.1 Basics of Mission Management

1.1.1 Mission Classes and Phases

In order to elucidate the intricacies of managing a robotic space mission during its operations phase and the underlying processes, it is necessary to first delve into the conception and launch of such a mission. To this end, it is prudent to define the boundaries of the terms *mission* and *project*, which may possess semantic overlap. The former encompasses all resources, including funds, political relationships, purpose, hardware, software, staff, and processes, that are requisite for a spacecraft's journey and for obtaining the desired scientific data. On the other hand, *project* pertains to the planning, scheduling, and leadership necessary to judiciously utilize mission resources under the constraints of time, cost, and other risks, thereby enabling the attainment of the mission's objective. While the mission encapsulates the *what* of the endeavor, the project involves the *how* of the same. It should be noted that there exists a nuanced difference between these two terms, and while they may be used interchangeably in certain contexts, the most germane term will be employed in each discussion. Furthermore, it is important to clarify that our discourse will be restricted to NASA space missions, as the agency, and especially JPL within it, have exhibited leadership in this domain for several decades. Plus, the number, magnitude, and budget of the missions developed and managed within the United States far exceeds those of other space agencies and research centers.

Within the purview of NASA, a taxonomy of distinct robotic deep-space mission classes exists, predicated upon their genesis, funding sources, and budgetary limits. While some missions are solicited across multiple research centers, others are initiated and directed by the United States government. In the former instance, NASA will issue calls for proposals, specifying one or more scientific goals and a cost ceiling that must not be exceeded. These proposals will originate from disparate research centers and undergo a rigorous evaluation process comprising gate reviews, scrutinies, and downselections by both NASA and the

scientific community. Ultimately, NASA will designate a winning proposal and team, signifying the mission's transition from a proposal to a fully-fledged undertaking. Subsequently, NASA will furnish the mission with requisite funding and logistical support. This proposal-competition-selection pattern is a recurrent feature of programs such as the Discovery and New Frontiers initiatives[5].

Conversely, on occasion, specific branches of the US federal government collaborate with the scientific community to finance directed missions. Such missions may not undergo a competitive selection process and instead may be awarded directly to a NASA center or amalgamation of centers following a negotiation phase that considers the centers' abilities. This mode of operation characterizes what is currently termed *large strategic science missions*[6], formerly referred to as *flagship missions*. As the name suggests, these missions represent the largest and most expensive robotic undertakings within NASA's purview, with recent missions in this category surpassing the $1 billion mark in contemporary currency. Noteworthy examples encompass the Voyager, Cassini, Mars Science Laboratory (Curiosity Rover), Mars 2020 (Perseverance Rover and Ingenuity Helicopter), and Europa Clipper missions.

While the two aforementioned mission types based on their origin are the prevailing patterns, at times, either NASA or the executive and legislative branches of the US government may solicit a study to ascertain the feasibility of accomplishing a specific objective. Subsequent to the results of such an investigation, a mission may be commenced. It is noteworthy to recognize that although we have delineated the most typical NASA mission categories based on their origin, there have been and will continue to be other mission types that have not been expounded upon here. This non-exhaustive list is not comprehensive in

[5] The Discovery and New Frontiers programs are NASA initiatives that focus on developing innovative and cost-effective space exploration missions. The Discovery program focuses on missions that can be completed at a lower cost, while the New Frontiers program targets medium-class missions that address high-priority scientific goals.

[6] NASA's large strategic science missions program is a multi-billion dollar effort that focuses on developing and executing large-scale space missions that are designed to answer some of the most pressing questions in astronomy, astrophysics, and planetary science.

the sense that the US government and NASA must remain adaptable to evolving demands and contexts.

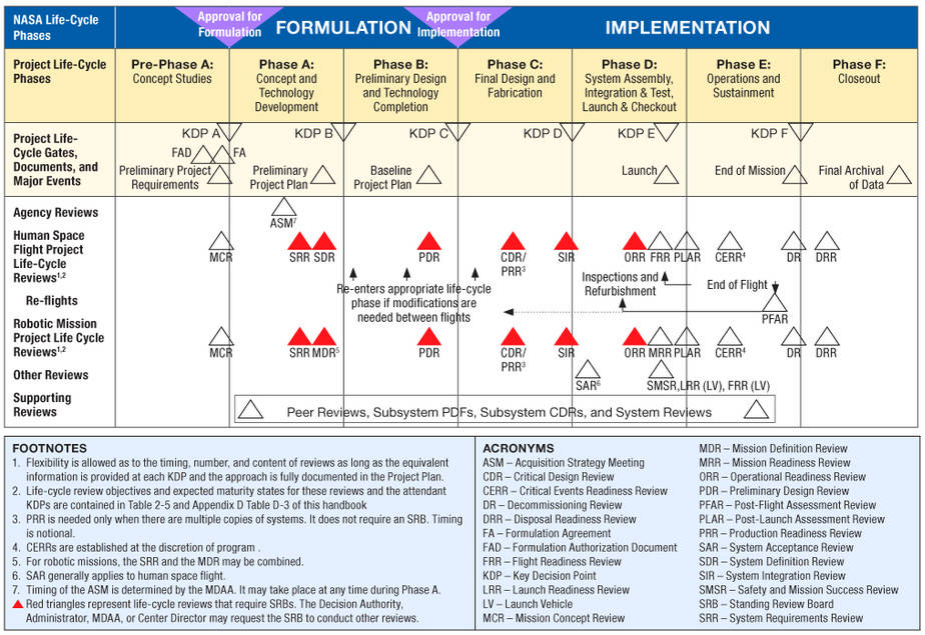

Figure 3. Lifecycle Phases for Flight Projects - NASA's Systems Engineering Handbook. Chart with an overview of the various phases encompassed within the life cycle of a NASA mission, delineating the key products and milestones that constitute each distinct phase.

Regardless of its inception origins, once a mission gets its approval, a project can start. And even if not always explicitly recognized, one could argue a project always ostensibly holds a provisional status, as funding can be paused, withdrawn, or withheld at any time according to scientific, technical, or political parameters. While progress in the project makes it increasingly cumbersome to halt, the possibility of funding cessation persists throughout the project's trajectory. Nonetheless, assuming nominal execution of the project, NASA's Systems Engineering Handbook [1] stipulates a series of distinct, rigorously controlled phases, as illustrated in Figure 3, that the project must pass through. At a macro level, these phases encompass a *Formulation* super-phase and an *Implementation* super-phase. The Formulation super-phase consists of two phases, A and B, both of which are devoted to delineating the mission's requirements / design and

encompass elements such as defining the software and hardware interfaces, ascertaining the heritage of the software / hardware / processes, undertaking trade studies, anticipating scientific returns, instrument selection, prototype development, mission simulation and analysis, and trajectory definition. These phases are critical to substantiate assumptions, delimit the development of the mission components, and create an initial design for the spacecraft's components and mission procedures.

The Implementation super-phase encompasses a sequence of four phases, namely C through F, that mark crucial milestones in the life cycle of a NASA mission. Phase C finalizes mission design and the development of spacecraft components, and commences or continues the development of mission operations processes that are implemented in the later phase E. Phase D completes the integration of spacecraft subsystems, including instruments, and finalizes the development of the launch version of the flight software, alongside the requisite preparations for launch and the launch itself. Phase E, which initiates post-launch, is responsible for the operational management of the spacecraft during its mission, a process colloquially referred to as *mission operations*. Its objective is to align with the mission plan and utilize the available resources to achieve the scientific or engineering objectives, while disseminating data of interest as it becomes available. This phase is subject to variability in its duration as it can be prolonged if the spacecraft is in good condition and NASA decides to continue funding the project, or curtailed owing to failure, an inability to achieve the mission objectives, or political exigencies. In light of this phase's significance to our inquiry, we shall expound upon it further in the ensuing section. Phase F is devoted to capturing, cataloging, and analyzing data, and sharing discoveries with both external and internal stakeholders. An important process within this phase is known as *capturing lessons learned*[7], in which the project team, with the assistance of NASA and other experts, reflect upon and articulate

[7] Capturing lessons learned during NASA mission Phase E provides benefits such as creating a historical record, identifying best practices and areas for improvement, improving mission management, and facilitating knowledge transfer and collaboration. The knowledge gained from these reports can be used to inform and improve future missions, leading to less risk and more efficiency.

the successes and failures of the mission as explicit knowledge, which can be leveraged in future missions.

1.1.2 Phase E - Mission Operations

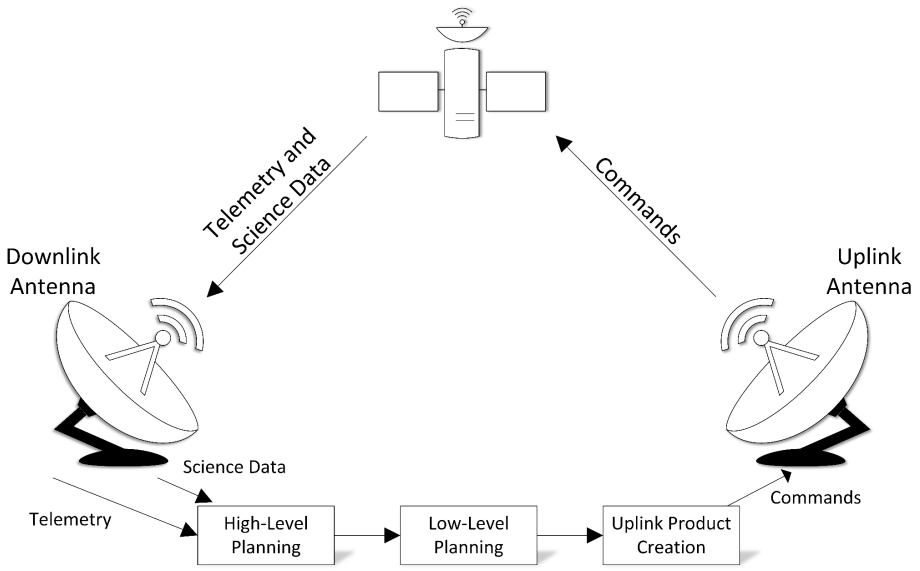

Figure 4. Uplink and Downlink: Two-Way Communication for Space Missions. Ground stations use powerful antennas to send commands to the spacecraft and receive data from it. This enables continuous communication between the spacecraft and the ground team, allowing for constant monitoring of mission progress, as well as the ability to remotely command the spacecraft.

As noted earlier, in Phase E the mission team employs available resources to accomplish the scientific or engineering objectives of the mission, while also releasing data of interest as it is processed and becomes available. Specifically, this phase involves controlling the spacecraft, which is situated in space, and directing it to execute a predefined set of actions aligned with the mission's defined goals. The mission team can utilize two systems - the flight system (which encompasses the hardware and software of the spacecraft) and the ground system (which encompasses mission operations personnel and the hardware and software utilized to manage the spacecraft from Earth). The means by which the mission team identifies the actions that the spacecraft will undertake is known as the

uplink process. Although the uplink process may differ across missions, it usually encompasses three core stages: (1) high-level planning, (2) low-level planning, and (3) uplink product creation. This overall process is depicted in Figure 4.

The high-level planning process is a crucial stage employed by large-scale missions in the field of spacecraft design and operation. Over time, this process has evolved due to the increased complexity of spacecraft that now include more advanced computer science techniques such as analysis algorithms, discrete-event simulators, and automated planners. The high-level planning process starts with the science planning sub-process, whereby scientists use software tools to create a model of the spacecraft and its surrounding environment. These tools [2, 3] provide a simulated environment for scientists to develop actions, primarily of remote sensing nature, and involving instruments, to determine if they can achieve a tactical science goal. On some missions, the use of automated science planning tools [4, 5] can replace the manual input of scientists to simulate actions. These automated tools can receive goals and constraints as inputs and provide activities that will satisfy the goals and constraints based on the spacecraft's environment and available resources. The output of the science planning process is known as the science plan, which encompasses a series of science activities that the spacecraft will perform at different points in time. This plan is frequently presented as a Gantt chart, as depicted in Figure 5.

Since the objective of the science plan is to contain and communicate the tactical activities that will help achieve the science goals of the mission, it often does not consider all the limitations from the spacecraft and ground systems, at least in the first iteration of the science plan. As a result, there is a continuous process to improve and refactor the plan given the current or expected state of the overall system and its resources. This iterative process is necessary as the development of the science plan is limited by the scientist's inability to comprehend the operations of all subsystems at any given point in time. In fact, the comprehensive nature of this task precludes the possibility that any single engineer or scientist could possess such knowledge. To address this limitation,

the broader activity planning process leverages computational power and modeling and simulation techniques.

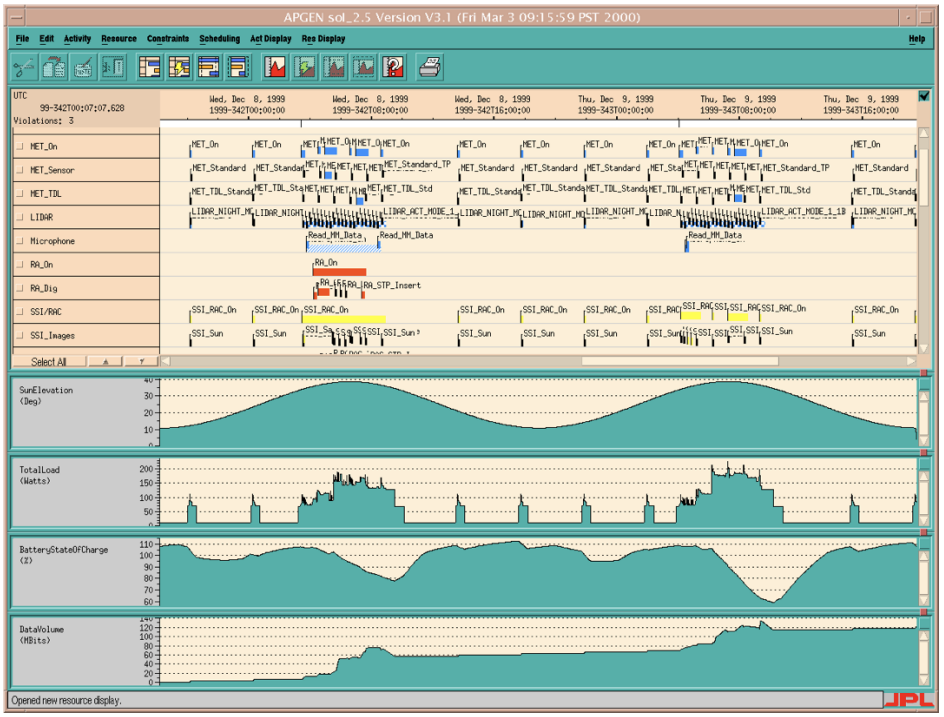

Figure 5. Activity Plan. The top half of the figure presents the activity plan, which depicts the spacecraft activities in a Gantt chart format, where each activity has start time, end time, and a predefined duration. The bottom half of the figure presents simulation data, which consists of time series data on a range of modeled resources, including sun elevation angle, power load, battery state of charge, and data volume.

In activity planning, a common practice is to subject a completed science plan iteration to a simulation system that forecasts the state of pertinent resources on the spacecraft and the ground system [6, 7]. The anticipated states are recorded in time series, as depicted in Figure 5. These time series are then evaluated both visually by humans and automatically by software constraint checkers to determine the feasibility of the plan in its current form. In case these time series present values in the modeled states that contradict the mission's parameters (e.g. the spacecraft exceeds power threshold, battery state of charge is below the safe threshold, invalid pointing angle for an instrument), modifications to the plan would be necessary. Engineers can revise the plan to circumvent undesired states

while ensuring that science activities can still be carried out. This requires a modification of the plan, followed by another simulation run and another check of the resource time series. This process repeats until all resources are in desirable states. However, if direct refactoring of the plan does not permit the completion of science activities, then engineers and scientists must collaborate in devising a new science plan that is executable.

The advent of computer-driven automated planning [4, 5, 8, 9] has provided a noteworthy counterpoint to the traditional, more manual and involved pattern. Over the past few decades, automated planning has gained increasing prominence, promising time savings and decreased risk. In this paradigm, a software system is tasked with receiving three inputs: (1) system states to be avoided, which reflect the constraints the engineer deems pertinent to the planning process; (2) an existing or empty plan; and (3) the goals that the spacecraft must accomplish. Utilizing these inputs, the automated planner executes a search algorithm to generate a new plan that satisfies the given goals while partially or fully adhering to the established constraints. This approach fundamentally replaces, either partially or entirely, the traditional planning and simulation process that was previously carried out by human operators, with an algorithmic approach. Today, automated planning tools are commonly employed in conjunction with manual processes for activity planning, serving to aid in the creation of an initial base plan that engineers subsequently expand upon, or when there exist specific and repetitive tasks where automation can prove beneficial.

One can think of the final product from the activity planning process, the activity plan, as an augmented and executable science plan. The augmentation includes the science activities developed during science planning, and engineering activities such as uplink and downlink of data, trajectory correction maneuvers, etc. that might not have been taken into account during science planning but are vital to sustain the spacecraft and the overall mission. However, activities, as they appear on an activity plan, are not the kind of construct that is understood by a typical deep-space spacecraft. Spacecraft implement computers and software that operate at a level lower than the human-oriented activities present in an activity

plan. The low level constructs spacecraft understand are called commands, which are the base instructions interpreted and acted upon by the flight software. From this, a sensible reaction would be to ask why there is a need to separate activities from commands, since after all, the only thing a spacecraft understands is commands. In response to this insightful question we might say the need is similar, but still somewhat different, to that of high level programming languages that execute, deep down, as machine code. One can code in C and then have a compiler create machine code for a specific computer architecture. In this case, the C code operates at a more humanly-understandable level, whereas the machine code generated will be different based on what computer it will be run on, and those are typically details a human will have trouble following and might want to ignore. The activity/command dichotomy is similar in that humans understand activities, but spacecraft understand commands.

The separation between activities and commands also makes sense from the perspective of the different frequencies in which activity types and command types are updated. Missions maintain a list of activity types described in an activity dictionary[8] and, analogously, they also maintain a list of command types in a command dictionary[9]. The dictionaries describe the characteristics and specifications of each activity or command, including their parameters, constraints, and other relevant data. The mission planning team may update the activity dictionary to enhance the planning of the spacecraft's actions by modifying, creating, or deleting activity types. In contrast, updates to the command dictionary are rare due to the high risk of modifying an integral part of the spacecraft's operation. This approach seeks to maximize the mission's efficiency by allowing updates to the activity dictionary while maintaining stability in the command dictionary.

[8] An activity dictionary is a tool for documenting and tracking tasks and actions in robotic space missions. It contains a list of activities, how to execute them, and necessary resources. It helps manage and coordinate operations.

[9] A command dictionary is a database that contains all possible commands that a spacecraft can receive, along with information on how to execute them. The command dictionary is highly dependent on the low-level implementation of the spacecraft.

The process of activity expansion or command expansion, depending on the terminology used, is responsible for connecting activities and commands. Command expansion, the preferred term in this text, refers to the process by which software logic takes an activity plan and transforms it into commands. The logic can be either static, implemented as a rule-based parser that creates a sequence of commands for each activity in the plan without considering external context, or dynamic, using algorithms to generate a sequence of commands based on the plan as a whole and the spacecraft's state at each point in time. There is an ongoing debate within NASA about the merits of each approach, with the static method providing better traceability between activities and commands and being more widely implemented in recent missions. However, it can be anticipated that future missions will likely adopt more stateful approaches as spacecraft mission planning becomes more goal-based. As a matter of fact, the past seems to point to a pattern similar to what took place in the spacecraft computing domain. Initially spacecraft contained custom electronics with limited computing capabilities, then they implemented low-complexity chips which eventually became fully programmable microprocessors. These microprocessors initially executed fully custom code, and now they implement fully capable operating systems which run the flight software. Earlier missions implemented their mission planning only with commands, whereas now they use activities which are expanded into commands. In the future missions will likely lean towards goal-based planning on the ground segment, which can then evolve into goal-based execution on the spacecraft itself, or at the very least, execution of activities by the spacecraft instead of commands. Perhaps stateful command expansion can be an interim and useful piece towards learning how to design a fully goal-based spacecraft system.

A discourse is ongoing within NASA centers on whether or not commands necessitate validation via simulation, akin to the requirement for activity plans. The debate presents two sides: (1) There is a requirement to validate commands as a form of double-checking to ensure that the expansion process functions as intended, specifically, that commands are in line with the activity plan's objectives

and respect the engineering constraints designated for the mission; and (2) the process of validating commands is redundant and costly, as commands can be inherently valid if the expansion process works as intended. In current practice, deep space missions continue to validate commands post activity expansion as a measure of risk mitigation, while proponents of constructing commands that are valid by design promote the thought that repeated validation efforts represent a significant expense that could be reduced by conducting a single validation at the activity level.

Command validation entails the use of tools to model the low-level behavior of the spacecraft. This task is commonly accomplished through the implementation of simulators [10] that integrate models or the utilization of the flight software [11] itself on either a testbed or an end-user workstation. However, both of these options entail significant costs for implementation and maintenance. First, the development of simulators that capture the low-level behavior of a spacecraft based on input commands is a nontrivial task that includes the creation of the simulator itself, which could also involve the implementation of a multi-mission generic component to facilitate the reusability of the simulator across missions, as well as the development of models that capture the physical and logical behavior of the spacecraft. The development of the simulator is a significant undertaking, and the development of models that offer sufficient fidelity to validate the commands is a complex and ongoing process over the duration of the mission. Second, the utilization of flight software to model the spacecraft's behavior poses significant obstacles that have been challenging to overcome historically, as it necessitates modifications to the code that could compromise the validation's fidelity. Indeed, when this approach has been employed, it has involved duplicating a version of the flight software code to enable modification and use in this capacity. In this case, the code used for validation is partly distinct from the canonical version of the flight software, and incremental incorporation of updates from the canonical flight software code into the validation code is required. Both modeling-based and flight software-based validation approaches have been effectively employed in flight projects, and it appears that the quality of

the chosen approach's execution might matter more than the approach itself in terms of risk and cost.

Once command validation has been completed, the subsequent stage involves converting the command list into a binary format that can be recognized by the flight software. This process can be automated by utilizing software. The binary file is then transmitted to the Deep Space Network (DSN), an internal organization within JPL responsible for the uplink and downlink of data between the spacecraft and Earth. The DSN possesses its own funding, workforce, and a network of antennas located in three distinct sites separated by roughly 120°: Canberra, Australia; Madrid, Spain; and Goldstone, California, United States, as depicted in Figure 6. The scheduling of uplink windows by the DSN entails the coordination of ground and spacecraft resources to transmit commands to the spacecraft. Due to the DSN's responsibility for communicating with JPL or NASA deep space spacecraft, as well as spacecraft from other countries or organizations, it is challenging to pre-plan communication windows. Priorities may shift rapidly, as in the case of a malfunctioning spacecraft requiring immediate communication for diagnosis. Additionally, other factors such as solar system geometry, spacecraft orientation, frequency availability, and others contribute to scheduling complexity. The DSN employs goal-based software to manage these constraints and plan its operations, as exemplified in [12].

Upon finding a transmission window, the DSN initiates the transmission of the binary command file to the spacecraft. Upon receipt, the spacecraft undergoes a processing step that includes decryption, as well as verification of the file's integrity through the use of checksums or similar mechanisms. If the file is not corrupted, the spacecraft schedules the commands for execution according to the instructions of the mission staff. The process of executing these commands is multifaceted and involves a range of factors related to spacecraft operation that are beyond the scope of this discussion. Nevertheless, it suffices to note that if the spacecraft executes the commands successfully, it will result in the acquisition of scientific data via the instruments aboard the spacecraft, such as magnetometer data, images, and other relevant scientific measurements.

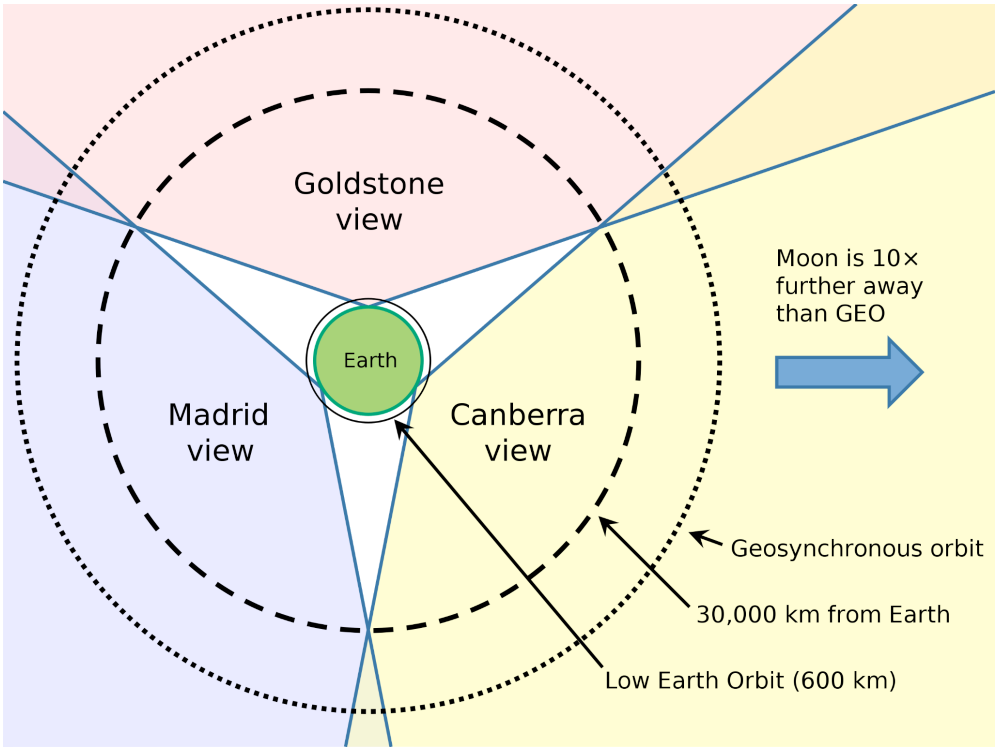

Figure 6. The Deep Space Network Locations. The diagram shows the Earth at the center and the three different locations with DSN antennas. Each location is roughly separated from the other two by 120°. "SimonOrJ" (U/T/C), CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons.

The downlink phase commences through a prearranged time window agreed upon between the DSN and mission staff. During the downlink stage, the spacecraft transmits the obtained scientific data to the DSN, accompanied by telemetry data that records the evolution of pertinent onboard resources over the course of command execution. Mission staff will then receive these two outputs, parse them, and distribute the relevant scientific data to internal and external partner science teams for analysis and interpretation. These scientific findings, in turn, contribute to the advancement of our understanding of the Universe and are published in scientific journals. Meanwhile, the resource telemetry will be cross-referenced with the simulated resources generated during the mission planning stage to ensure they correspond to the mission planning team's expectations. Any discrepancies that arise could be indicative of outdated or faulty models and assumptions employed in the planning process or a

malfunctioning spacecraft component. As soon as the downlink operations are finalized, the mission planning process begins anew, and the cycle continues.

The preceding description of the ground steps in the uplink-downlink process has been synthesized into a graphical representation denoted as Figure 7. This figure serves as an expanded version of the previously presented Figure 4, offering a more comprehensive overview of the ground steps involved in the communication between the spacecraft and the ground system.

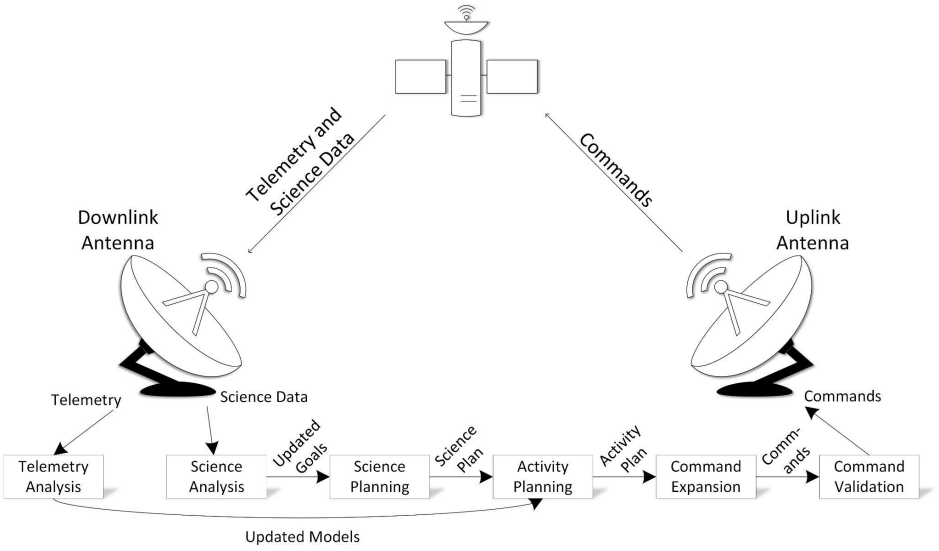

Figure 7. Uplink and Downlink II. The data received from the spacecraft is first processed by the ground team during the downlink step. Subsequently, the Science Planning phase commences, which involves the creation of a Science Plan. The Science Plan is then transformed into an executable Activity Plan, which further undergoes expansion into Spacecraft Commands. These commands are validated and eventually transmitted to the spacecraft.

1.1.3 Science Planning

Having gained a comprehensive understanding of the steps involved in developing a mission and commanding a spacecraft, it is essential to explore the concept of science planning through the use of an exemplifying software system, which is commonly utilized in orbital missions. It should be noted that science planning procedures may vary across different types of missions due to the

unique challenges posed by each mission type. Specifically, orbital and fly-by missions, which are categorized under orbital missions, are characterized by predictable orbital mechanics that enable mission teams to plan spacecraft actions well in advance over long periods of time, ranging from weeks to years. Conversely, other mission types are associated with greater uncertainty, making it infeasible to determine spacecraft actions with certainty. For instance, surface missions utilizing rovers or helicopters necessitate a different planning approach that is contingent on the level of uncertainty that comes with exploring unknown terrain, where the spacecraft may encounter unexpected obstacles that were not foreseen during mission planning. Although the science team on an orbital mission may need to adjust the science plan in response to new discoveries, this is not a frequent or predictable occurrence as it is in surface mobility missions.

Differences in the characteristics of orbital and surface mobility missions dictate the use of specialized science planning software. In the case of orbital missions, science planning software supports a long-term, geometry-focused process within a well-known space region, whereas surface mobility science planning software is tailored to a more reactive, environment-focused process within an unfamiliar space environment. NASA has developed two systems which are representative of these two models. The first is the Science Opportunity Analyzer (SOA) [2], a scientist-driven software tool for orbital missions, and the second is the COmponent-based Campaign Planning, Implementation, and Tactical (COCPIT) [3] software tool, which serves a similar purpose for surface mobility missions. Both software tools have been used in various robotic space missions, with SOA utilized in missions such as Cassini, Dawn, Psyche, and Europa Clipper, while COCPIT has been used in the Mars Science Laboratory (MSL) and Mars 2020 missions. In this article, we will focus on the science planning software for orbital missions, with a more detailed elaboration of SOA's functionality.

1.2 Background and Related Work

1.2.1 Science Opportunity Analyzer (SOA) and its impetus

In the field of orbital missions, the fundamental unit for science planning is known as *observation*. Specifically, an observation is a collection of remote sensing actions executed by an instrument with the objective of achieving a specific scientific goal. For example, an observation could entail capturing a swath of photographs across the sunlit surface of an asteroid through the use of a framing camera installed on an orbiting spacecraft. This would involve a range of components, including the frequency of the images captured, the positioning of the framing camera at each time point, and other pertinent parameters.

Prior to the advent of SOA, the process of developing observations was an arduous and time-consuming endeavor for science teams. This was largely due to the lack of integration between the many fragmented tools utilized in the mission planning process. Moreover, the design of observation tools was the responsibility of non-co-located teams, leading to issues of incompatibility and capability gaps. It was this very challenge that acted as the catalyst for the SOA development effort [13], which sought to create a user-friendly, multi-mission software solution that would enhance the productivity of the observation design process. Given the ambitious objective of the project, its original stakeholders and funding organizations were required to proceed with great methodological rigor to ensure that the initial versions of SOA achieved meaningful improvements on a range of aspects related to the status quo.

In 1998, a Quality Function Deployment[10] (QFD) [14] initiative was implemented by a task force within the JPL to narrow down the initial set of requirements and

[10] Quality Function Deployment (QFD) is a structured methodology used in product development and project management to translate customer requirements into specific engineering characteristics and specifications. It is a tool used to ensure that customer needs and expectations are met during the design and development stages of a project. QFD is commonly used in the aerospace industry, including by NASA, to develop and optimize the design of complex systems.

use cases for the Science Opportunity Analyzer (SOA) software [15]. The QFD process consisted of three steps, starting with the design of a closed-ended questionnaire that would be later validated by the answers provided to other open-ended questions. The questionnaire was then distributed to a group of 40 individuals consisting of scientists, mission planners, command validation engineers, configuration management engineers, software developers, cybersecurity engineers, line managers, mission designers, and systems engineers for completion. The final step in the process was the analysis of the results and the creation of the initial set of use cases and requirements for the software].

The QFD initiative yielded five main use cases for the SOA software: (1) Opportunity Search, (2) Observation Design, (3) Visualization, (4) Flight Rule Checking, and (5) Data Output. Since these use cases were met through different capabilities present in disconnected tools at the time, feedback obtained through the QFD indicated that these use cases and their implementation in SOA must be part of an integrated and iterative science planning process. The sequence of use cases in the planning process started with the Opportunity Search capability, which allows users to search for the times when an opportunity takes place. Within SOA, an opportunity is a combination of geometric events via Boolean logic that scientists are interested in planning observations around. As an example, two valid geometric events for an opportunity could be:

- E1: *There is an occultation of the Earth behind Ceres as seen from the Dawn spacecraft*
- E2: *The Dawn spacecraft is at a distance of 15,000 km or less from Ceres*

These geometric events are searchable independently or combined with a logical AND/OR and negated with a logical NOT. Notably, SOA's QFD initiative found that the software must be capable of combining a list of predefined atomic[11] geometric events that could grow over time given user needs.

Based on the time intervals identified through Opportunity Search, the user would then proceed to formulate and visualize the various observational actions

[11] Atomic refers to opportunities that can not be further broken down into smaller opportunities.

of the spacecraft's onboard instruments. To achieve this, SOA offers a means of defining the specific times, angles, and other relevant parameters that dictate when one or more instruments perform their observations. Additionally, SOA is be capable of visually representing the observation system in both 2D and 3D, taking into account factors such as time, spacecraft position, the observed body, trajectory, and instrument field of view[12] (FOV) projection onto a surface, among others. The Visualization capability serves the purpose of facilitating effective communication of observation design and assessing its quality visually. For instance, it enables users to determine whether an observation is targeting the right areas of the observed body. In numerous cases, a natural iterative process exists between the design of an observation and the associated visualization that SOA supports. Typically, the visualization of an observation prompts adjustments to its design, thus leading to a repetition of the Observation-Design/Visualization cycle. Figure 8 provides a clear representation of a typical usage of the Visualization capability for the high altitude mapping orbit of the Dawn mission. The different colors employed in the image help distinguish the imaging swaths obtained at varying time intervals, while the cratered surface of Ceres is evident from the texture of the shape model.

Subsequently, the user would employ an additional quality check, namely Flight Rule Checking, for the observation. This feature is able to validate that an instrument does not transgress a particular geometric or dynamic constraint, such as "the instrument shall not be directly pointed at the Sun" or "the spacecraft shall not exceed its maximum acceleration rates", respectively. Analogous to the representation of opportunities, Flight Rules are geometric constraints that can be combined using Boolean logic. Any infringement of these rules must generate a notification during observation design or visualization. Whenever a Flight Rule violation occurs, the user would be required to make a decision regarding whether to disregard the violation or modify the observation in order to avoid contravention of the constraint.

[12] A spacecraft's instrument Field of View (FOV) refers to the angular extent of the observable region that the instrument can detect. In other words, it is the area or volume of space that the instrument can *see* or observe. The FOV is affected by the instrument's design, such as its aperture size, and the spacecraft's orientation and position relative to the target being observed.

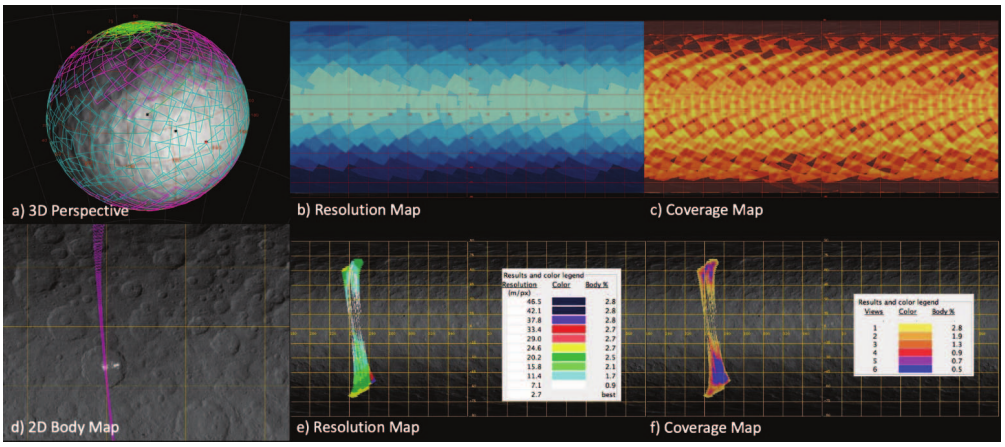

Figure 8. Examples of visualization options in SOA. a) 3D perspective view of Ceres imaging plan for the high altitude mapping orbit, b) 2D latitude/longitude map of the best image resolution from this imaging plan, c) 2D coverage map for images that contribute to the stereo topography campaign, d) 2D body map showing imaging plan for one elliptical orbit targeted to Cerealis facula in Occator crater, e) 2D latitude/longitude map of 25 elliptical orbits showing best imaging resolution, and f) 2D coverage map for imaging in 25 elliptical orbits that contribute to local optical topography solutions.

Lastly, SOA provides a capability to perform Data Output tasks. In this context, Data Output is understood as the ability to export data to a spreadsheet in text format and to create graphs that exhibit values which are subject to change over time. These values may relate to either the observation defined in previous stages or to the trajectory under consideration.

Upon realizing the software's use requirements, preliminary versions of SOA were made available to science teams for evaluation. The successful integration of these use cases in a user-oriented manner and the positive feedback received from science teams resulted in the adoption of SOA by several orbital robotic space missions, with continued development and improvements to this day. It is worth noting that while the way SOA handles science planning is not unique, it represents the refinement of a process that has taken place over the course of numerous orbital robotic space missions within NASA.

1.2.2 Tooling Taxonomy

The utilization of user research techniques in science planning, specifically the use of SOA, was exemplified in the preceding section. The resulting findings are typically used with JPL as a paragon for the design of orbital science planning software to this day. However, while SOA serves as an exemplar of a tool designed with comprehensive requirements, it is not the only option for opportunity search. This section aims to explore several such tools and classify them into two distinct categories.

The first category of contemporary software utilized for Opportunity Search is ad-hoc small mission tools, typically in the form of scripts[13]. These tools are commonly developed with toolkits such as MATLAB [16] or programming languages such as Python, and leverage supporting libraries such as Spacecraft Planet Instrument C-matrix Events (SPICE) [17] and MONTE [18], which respectively facilitate geometric calculations in space and provide astrodynamics functionality. In situations where a mission requires a search for an opportunity for which a search algorithm has not yet been developed, an engineer or scientist will typically clone and adapt an existing script or craft a new one. Typically, the lifecycle of these scripts is intimately linked to that of the mission they are designed to support; development begins when a mission is conceptualized and the scripts are maintained and utilized until the mission's conclusion.

Despite their utility, these scripts may be problematic in terms of reuse in other missions, as they are often customized to suit the specific requirements of the mission at hand. The limitations to their reuse are numerous and include, but are not limited to, their insufficient generality in defining opportunities, dependence on mission-specific frameworks or tools, and a tendency towards an ad hoc development approach that may limit extensibility in the future. More importantly, this approach can introduce programming errors by failing to differentiate between the concerns involved in the modeling of an opportunity

[13] Small script development in languages such as Python, MATLAB, sh, etc. typically involves writing short, focused programs that perform a specific task. These scripts are often simple and straightforward to write and can be easily modified and adapted to suit the needs of the user.

and the resolution algorithm that enables its discovery. This method thus violates the separation of concerns design principle [19], which stipulates that "software should be decomposed in such a way that different concerns or aspects of the problem at hand are solved in well-separated modules or parts of the software[14]".

The second family of tools is multi-mission, larger-scale, frameworks[15]. These software frameworks are designed to be employed across various missions and endow mission staff with a set of frequently used capabilities that can be extended. Such frameworks can be offered either as capabilities in a software library, for instance, SPICE's Geometry Finder [20], or as capabilities in an end-user software package, such as SOA, WebGeoCalc [21], Percy [22], SOLab [23], or MAPPS [24].

While some of these frameworks provide means for extension, they also show limitations. On the one hand, SPICE's Geometry Finder, Percy, and WebGeoCalc conflate the definition of an opportunity with the search of the same. This issue is evident when, for one opportunity type, there are multiple resolution/search methods with different parameters. Should parameters intrinsic to the resolution method be included in the opportunity definition? Ideally not since opportunity modeling and resolution are two separate concerns. Additionally, these tools only provide a limited set of opportunities for users to search for; thus, if a mission has a need to search for a new opportunity, mission staff can end up developing scripts or small tools akin to the mission-developed scripts option described previously.

[14] Separation of concerns is a fundamental principle in software design that advocates breaking a software system down into distinct parts, each responsible for a specific aspect of the overall functionality. This approach makes the software easier to understand, develop, test, and maintain. By separating concerns, software developers can minimize the impact of changes to one part of the system on the rest of the system. It also helps to improve the overall quality of the software, making it more reliable, scalable, and secure.

[15] Large-scale software development, compared to script development, involves creating complex software systems that require careful planning, design, and implementation. Such software systems are typically written in more elaborate languages such as Java, C++, or C# and require a significant investment of time and resources to develop. They must be designed to be modular, maintainable, and scalable, and must be extensively tested to ensure reliability and robustness.

On the other hand, while a tool like SOA does provide a separation between the definition of an opportunity and the search of the same, it does not provide a capability for end-users to add new opportunities or search implementations of their own. This could force users again towards the development of small-scope scripts, or towards the request for a development team to augment the software with additional opportunities. The latter approach could entail additional cost and a wait longer than the project can afford due to more protracted development, release, and deployment cycles linked to larger development efforts.

The forthcoming sections will elaborate on these ideas and provide a discussion of JPL-developed opportunity search frameworks, with a corresponding reduction in attention paid to any additional less-relevant functions the tools may possess. The present investigation will abstain from an in-depth discussion of mission scripts owing to the dearth of publicly available information regarding their operation and because they are predominantly constructed on a case-by-case basis. The focal point on JPL-generated software is attributable to the quality of the mission planning and space geometry software tools and publications that have been made available by the organization, as well as the overall commitment to openness from the organization. Though information pertaining to non-JPL mission planning and space geometry software applications is limited, it is commonly recognized that they draw on foundational libraries and frameworks developed by JPL, including the esteemed SPICE library. This scarcity of information serves as a notable obstacle in fully comprehending non-JPL applications.

1.2.3 SPICE

The SPICE library by the Navigation and Ancillary Information Facility[16] (NAIF) at JPL is a comprehensive collection of space geometry functions. SPICE, as a

[16] The Navigation and Ancillary Information Facility (NAIF) is a service at JPL that provides space mission navigation and data analysis support to the scientific community. It maintains a database of spacecraft trajectory and orientation information, as well as planetary and satellite ephemerides, which are used to navigate spacecraft and plan scientific observations.

software library, is not used independently; rather, it is integrated into other tools that make use of it. As such, integration with SPICE requires software development expertise and at times thorough understanding of low-level space geometry concepts.

Designed primarily for calculations involving spacecraft, spacecraft instruments, and target body geometry data, SPICE provides a range of functionalities, including calculations based on ephemeris, size, shape, orientation, and field of view. To supply geometric data to the calculation code, SPICE uses *kernels*[17], which are either provided by the developers of the library or by different missions that require SPICE functionality[18]. The NAIF team at JPL is responsible for the development of multi-mission kernels. On the other hand, individual missions are responsible for developing their own mission-specific kernels, which include spacecraft and instrument geometrical information, trajectory data, and other relevant information.

In recent years, NAIF has added a powerful new feature to SPICE called Geometry Finder (GF) [20], which offers a predefined list of higher-level calls that enable users to search for specific geometric events within defined time windows[19]. The calls that make up GF can be seen in Figure 9. Searching for any geometric event not on the list requires the implementation of numerical methods in conjunction with lower-level SPICE calls. GF not only provides a range of geometry-finding capabilities but also includes calls for performing set operations on time windows, which are a product of opportunity search.

[17] SPICE kernels are sets of data files containing information such as spacecraft ephemeris, instrument geometry, and other ancillary information that is required for mission design, planning, and analysis.
[18] As of 2023, public kernel data from various sources is compiled at https://naif.jpl.nasa.gov/naif/data.html
[19] As of 2023, an example on how to search for an distance opportunity with SPICE GF can be found here: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/gfdist_c.html

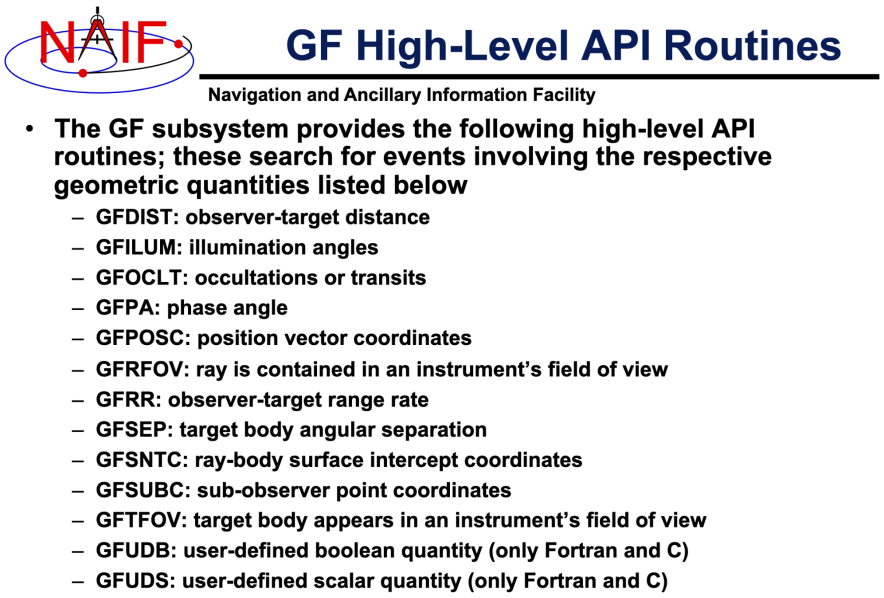

Figure 9. List of SPICE GF Functions. SPICE now provides various functions for searching geometric opportunities, which significantly simplifies the search process and makes it accessible to a wider range of users, regardless of their level of expertise in numerical methods.

1.2.4 WebGeocalc

WebGeocalc is a web-based tool[20] designed to provide scientists, engineers, and members of the general public with access to a range of high-level calculations offered by the SPICE library. WebGeocalc offers a distinct advantage over traditional approaches like SPICE, which often require software development and library installation. Instead, WebGeocalc enables users to initiate calculations directly from a web browser, which are then executed remotely on a server that implements calls to SPICE. The results of these computations are returned to the user via the web browser.

WebGeocalc offers three distinct families of calculations, including the *Geometry Calculator*, *Geometric Event Finder*, and *Time Calculator*. The Geometry Calculator is capable of executing nine different types of calculations that return a range of

[20] As of 2023, WebGeoCalc can be accessed from here: https://naif.jpl.nasa.gov/naif/webgeocalc.html

geometric values, such as angular separation, angular size, and illumination angles. The Geometric Event Finder provides time windows for ten different types of geometric events, including distance, position, and occultation. Meanwhile, the Time Calculator facilitates time conversion calculations between different time systems, such as UTC and Spacecraft Clock[21].

Figure 9. WebGeocalc Main Site. This interface presents a comprehensive list of the various operations that can be executed with WebGeocalc. Noteworthy among them, the *Geometric Event Finder* options, which stand out for their capacity to enable users to search for opportunities of interest.

[21] The term *Spacecraft Clock* denotes the onboard timing system of a spacecraft, which serves as the primary source of timekeeping for various spacecraft operations and related activities.

It is important to note that the calculation types offered by WebGeocalc are static and implemented on top of SPICE's GF. WebGeoCalc does not provide capabilities to add new types of events besides the ones available via SPICE's GF. Furthermore, the events provided by the Geometric Event Finder cannot be composed through boolean operations. A recent enhancement to the system is the inclusion of REST calls, providing users with the ability to execute these statically-defined calculations through web-based end-points. This new feature expands the capabilities of the tool and offers users increased flexibility in terms of the range of integrations that can be implemented with external software.

1.2.5 SOA

As discussed previously, SOA is a powerful tool optimized for streamlining the science planning process in orbital or fly-by missions. The software is designed to support various key aspects of the mission planning process, including Opportunity Search, Observation Design, Visualization, Flight Rule Checking, and Data Output. Among these use cases, the Opportunity Search feature is of particular significance as it enables users to identify and evaluate suitable time windows for specific scientific events based on user-defined geometric constraints.

SOA's unique Opportunity Search user interface, depicted in Figure 10, is designed to be intuitive and flexible, utilizing a tree structure that allows users to construct complex geometric constraints through the combination of basic building blocks. The software supports a range of concrete geometric constraints based on Distance, Occultation, Transit, Phase angle, Range rate, Angular separation, Elongation, and Quadrature calculations. In addition, the software also supports boolean operations, such as AND, OR, and NOT, making it possible to build even more sophisticated constraints.

SOA does not let users add new types of searchable geometric events without modifying the SOA codebase. As requested by users, the SOA development team will implement new search algorithms and include the new event in the UI. This

implies that a new version of SOA will need to be provided to users. Search algorithms are typically implemented in Java through external space geometry libraries such as SPICE.

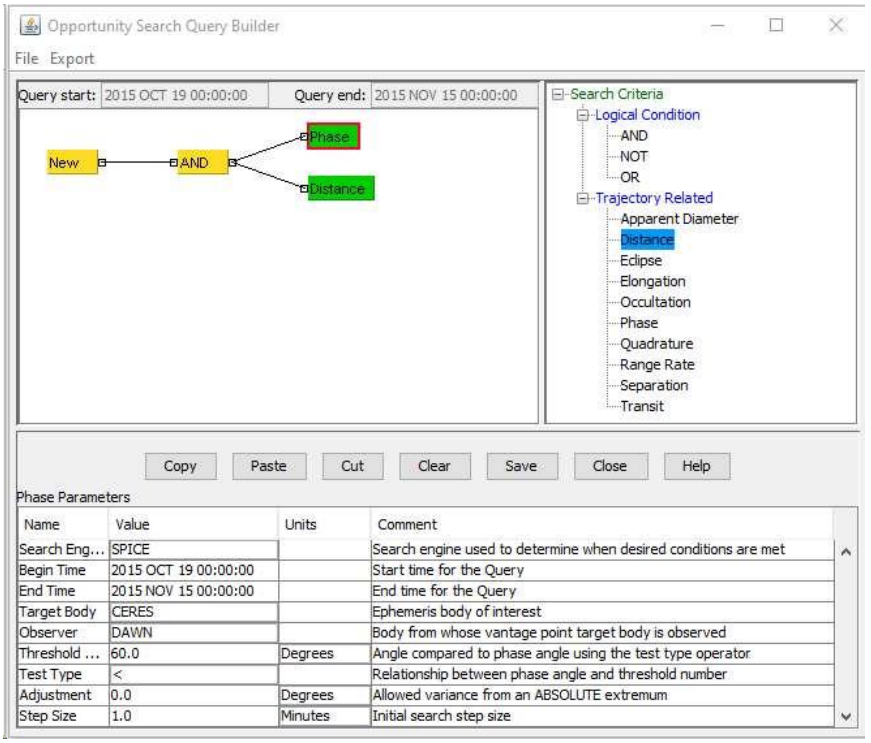

Figure 10. SOA's Opportunity Search Builder. This interface enables SOA users to construct their opportunities using a tree-based approach, which allows for Boolean relationships to be established among individual geometric constraints (top-left corner). Note the list of available opportunities (top-right corner) and the parameters that are accessible for the selected opportunity (bottom).

1.2.6 Research Question

Here we present a summary of our findings based on the existing multi-mission, larger-scale frameworks and introduce our research question. Again, our focus is on the larger-scale frameworks as we recognize that small-scale mission scripts have limited breadth and depth of information, and often lack proper software development practices. In brief:

1. The SPICE library is a comprehensive suite of space geometry functions for developers. SPICE enables the calculation of spacecraft, instrument, and target body geometry data, including ephemeris, size, shape, orientation, and field of view. SPICE has the GF functionality to search for a predefined set of geometric events. Other events would need their own implementation with the framework. While the SPICE library is a versatile tool, its use necessitates substantial software development expertise and a fundamental understanding of space geometry principles. Additionally, it requires integration into other software tools, which can be augmented by the creation of new kernels containing specific geometric information.
2. The WebGeocalc tool is a web-based platform with a user interface and RESTful services that are accessible to the general public. Unlike the SPICE library, it does not require software development expertise or library installation since calculations are carried out remotely via a web browser. Although the tool offers a limited selection of static geometric events aligned with SPICE's GF function, it does not provide constraint composition capabilities. The addition of new constraints is costly and entails recompiling the entire codebase and publishing a new version on a web server.
3. The SOA desktop tool is designed for non-developer scientists and features a user-friendly interface for query composition. SOA is the leading tool for opportunity search for a number of science teams in robotic space missions. It provides a predefined set of searchable geometric constraints. However, the tool's limited extensibility means that the addition of new constraints is costly and requires developers to recompile the entire codebase and publish a new version. The default constraints for the SOA tool are the result of a collaborative effort between developers and users.

Our findings are that each one of these tools serves a distinct purpose and audience, they suffer from technical shortcomings, such as the lack of separation of concerns between the modeling of an opportunity and its search, limited

reusability of opportunity definitions and search algorithms across missions, and a potentially ad hoc extensibility mechanism for opportunities and their search algorithms. These challenges lead to increased mission development costs and reduced knowledge capitalization across missions. To address these issues, we pose the main question of this thesis, which is to understand whether it is possible to design an opportunity search framework that:

A. Seamlessly integrates with contemporary and upcoming mission software and user teams.
B. Enables end-users to expand their opportunities and search algorithms without necessitating any modifications to the tools that leverage the framework.
C. Facilitates cross-mission reusability and adoption of developments and constructs implemented within it.

In response to the challenges faced in addressing the identified objectives, we have devised a novel approach, which we have aptly named Tychonis[22]. The following sections of this document are devoted to an in-depth exploration and evaluation of our approach, with a particular focus on the extent to which it satisfies the aforementioned criteria. We believe that through a thoughtful and intentional selection of design choices, Tychonis provides a highly innovative and practical solution that effectively addresses the identified challenges.

[22] In honor of Tycho Brahe, who was a Danish astronomer who lived in the late 16th century. He made many significant contributions to the field of astronomy, including the observation of a new star in 1572 and the creation of a comprehensive star catalog. He also wrote a book called "Astronomiae instauratae progymnasmata," which contained his observations of the positions of the planets and stars.

2 Methodological and Design Principles

As we have seen, current Opportunity Search software can be categorized as belonging to one of two categories: mission-developed scripts or multi-mission frameworks such as SPICE GF, WebGeoCalc, and SOA. Software in both categories might lack desirable qualities such as extensibility, reusability, and a separation of concerns between the definition of events and the search for the same. The existence of these shortcomings, as stated in the previous section, presents an opportunity to develop a software package that can advance the state of the art. In response to the challenge, Tychonis should emerge as a framework that can integrate with any mission software and user teams, enable end-users to extend opportunities and search algorithms without modifying the tools that use the framework, and promote the cross-mission reusability of the framework and developments implemented in it.

These features can be facilitated through a judicious application of software design's best practices and the construction of an object-oriented metamodel[23] which will be described in depth in Section 3. Tychonis' metamodel is currently fashioned using the Eclipse Modeling Framework (EMF) [25] and its Ecore metamodeling capability. In conjunction, EMF and Ecore can generate Java classes that can replicate the metamodel at the code level and both can be easily integrated with mission software. These Java classes, as well as their corresponding representation in the metamodel, have been structured to encapsulate the modeling of an opportunity and its subsequent resolution, with each aspect being assigned to a distinct class. In addition, while all these classes are amenable to user extension, Tychonis also provides inbuilt implementations that can be readily employed by missions.

[23] For now, let us state that an object-oriented metamodel is a model used in software engineering to define and describe the structure and behavior of an object-oriented system. It consists of a set of abstract classes, interfaces, and their relationships, which form the building blocks for the implementation of the system. The metamodel defines the syntax and semantics of the language used to describe the system, and provides a framework for modeling and managing the system's structure, behavior, and evolution.

To satisfy the requirements imposed by the research question, Tychonis adheres to several well-defined and instrumental design principles. These principles include, but are not limited to, separation of concerns, user extensibility, mission reusability, and independent verification and validation. The ensuing subsections will provide an overview of these principles and explain how our research has met them. Note that a detailed elaboration of these goals will be presented in subsequent sections; hence, a complete understanding of each individual concept by the reader is not imperative at this juncture.

2.1 Separation of Concerns

Separation of concerns [19], promotes the idea that "software should be decomposed in such a way that different concerns or aspects of the problem at hand are solved in well-separated modules or parts of the software". Tychonis follows the separation of concerns principle in the design and implementation of the framework and obtains benefits such as the following [26]:

1) A higher level of abstraction that permits thinking about concerns in isolation.
2) Improved understanding of the code. Concerns are easily distinguishable.
3) Weak coupling of concerns. Increased flexibility and reusability of single concerns.

Tychonis incorporates its most relevant concerns as (1) modeling, where opportunities are defined, (2) resolution, where opportunities are searched for, and (3) results, where the resulting time intervals are captured. These concerns, which are premised on our understanding of SOA, are also applicable to other software systems and can address a variety of use cases involved in the opportunity search process. Additionally, to meet the extensibility requirements, the software design for each of these concerns has been deliberately structured to provide end-users with the ability to expand the default capabilities of the framework.

The Tychonis framework has been realized as a Java-based, object-oriented [27] software package. As a result, each of the major concerns associated with the framework is implemented through a distinct class type. Specifically, `Query` classes are responsible for defining (or modeling) opportunities, while `Solver` and `SolverStrategy` classes are tasked with developing search algorithms that operate on opportunities. Finally, the `Result` class captures the precise time windows during which a given opportunity can be accessed.

Given their similar names, it is important to separate the concerns addressed by `Solver` and `SolverStrategy`. `Solver` and `SolverStrategy` differ in that each `Solver` contains a search algorithm for a specific `Query`, while `SolverStrategy` searches for composite opportunities defined as a tree of `Query` instances. In essence, a `Solver` returns a `Result` instance for a `Query` instance, whereas `SolverStrategy` returns a `Result` instance for a tree of `Query` instances contained under a root `Query` instance node. These trees are typically a result of the user's actions in connecting atomic `Query` instances with Boolean operations to model composite opportunities. Given `SolverStrategy` owns the responsibility of resolving composite opportunities, its default but extensible implementation executes a post-order tree traversal algorithm[24] that, for each `Query` instance node, determines what `Solver` applies to such instance, executes the `Solver` on such `Query` instance, and saves a `Result` object within the `Query` instance. Both Boolean operations and atomic opportunities are modeled with `Query` classes; therefore, each has its own mapping to a `Solver` class within a `SolverStrategy` implementation. Eventually, all `Result` instances are merged and propagated upwards through the tree to the root node, which contains the overall `Result` instance for the composite opportunity. It is noteworthy that after successful integration with Tychonis, the host application will not need to interact with `Solver` classes by name. Rather, the host application's search needs will be satisfied by an chosen implementation of `SolverStrategy`. The host

[24] The post-order tree traversal algorithm is a way to visit all nodes of a tree data structure in a specific order. It works by recursively traversing the left subtree, then the right subtree, and finally visiting the root node.

applications only need to be aware of `Query`, `SolverStrategy`, and `Result`.

2.2 User Extensibility

In order to improve over the usual static packaging of classes into a library that users integrate with, we chose to leverage EMF and its Ecore metamodeling capability. EMF is a code generation facility for developing applications based on a structured data model. Ecore, a subcomponent of EMF, provides a convenient way to define a model that is independent of its final implementation with a programming language[25]. Ecore models can be created using various methods such as hierarchical package-class trees, UML diagrams [28], annotated Java classes, and domain-specific programming languages [29]. An Ecore model is ultimately saved as an eXtensible Markup Language (XML) [30] Metadata Interchange[26] (XMI) file. EMF and Ecore have been useful in decreasing the design time dedicated to Tychonis since without them, we would have had to develop our own metamodeling capability.

The use of EMF and Ecore offers numerous benefits, including the creation of a reusable and extensible Tychonis model that captures the `Query`, `Solver`, `SolverStrategy`, and `Result` classes and their subclasses. This approach is particularly beneficial in the context of space missions, where users may need to add a new opportunity type to the Tychonis package without the need to write Java code. To accomplish this, a user would:

[25] In summary, Ecore provides a way to define object-oriented models, while EMF provides a framework for generating code and tools for working with instances of these models.
[26] XMI provides a standard, XML-based syntax for serializing metadata that is independent of the tools used to create and manipulate that metadata. XMI is used to exchange metadata in a variety of contexts, including software modeling and design, and has become an important part of the modeling ecosystem, especially in the Object-Oriented Programming (OOP) community.

1) Open the Tychonis Ecore XMI model file for editing.
2) Add a new opportunity type that inherits from a base Query class.
3) Add attributes pertinent to the new opportunity.
4) Auto-generate Java classes.
5) Compile and package Java classes in a Tychonis Java ARchive[27] (JAR) file.

The extensibility process would also apply to the `Solver` and `SolverStrategy` classes. To achieve this, it would be necessary to add search algorithms to the generated Java classes within the `solve()` method before packaging the JAR file. Unlike the extension process for `Solver` and `SolverStrategy`, the extensibility of the `Result` class does not involve modifications to the Ecore model. Instead, it is accomplished through `Result`'s ability to contain rows of results that, in turn, contain lists of objects whose classes implement the `Resultable` interface. This approach enables `Result` to be agnostic with regard to the type of data it can contain, allowing `Solver` to make the appropriate decisions.

The extensibility processes discussed herein provide a notable advantage in that applications that integrate with Tychonis can readily incorporate changes made to the model. Specifically, modifications to `Query`, `Solver`, `SolverStrategy` classes, and data within `Result` can be observed without the need for recompiling the code of the application[28]. This feature renders Tychonis a fully autonomous component, whereby capabilities can be evolved independently from other software. This is particularly relevant in scenarios where new geometry libraries or algorithms are to be implemented for search opportunities. To exemplify, SOA has changed its opportunity search engine over the years, and it did so by modifying its own code. If SOA had originally implemented Tychonis,

[27] A JAR (Java Archive) file is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform. JAR files use the ZIP file format and have a .jar file extension.

[28] Recompiling software after making changes to the code can be a significant burden for developers because it can be a time-consuming and error-prone process. Even a small change to the code can require recompiling the entire software system, which can take a long time, especially for large and complex systems. This can have a significant impact on users, as it can delay the release of new features or bug fixes, and can also make it more difficult for users to obtain the latest version of the software.

then this change could have been limited to just developing new Solver classes, and the release of an updated version of SOA with the changes would not have been required. It is reasonable to think that this scenario will present itself again in the future, e.g., when a new version of SPICE provides new capabilities SOA wants to leverage, or when a mission's parallelized version of a search algorithm will provide improved search performance.

2.3 Mission Reusability

Tychonis provides a range of default opportunities, which are delineated in Table 1, along with their associated class names and parameters. The selection of these defaults was informed by the aim of furnishing a feature set that is comparable to both SOA and WebGeoCalc. Notably, each of the Query classes within Tychonis can model a family of opportunities, thanks to the parameters that they make available to the host applications, thereby promoting reusability at the opportunity type level. Furthermore, Solver classes can also search for a family of opportunities, as they define a set of criteria that dictate the type of Query classes they accept. If a space mission finds the currently available opportunities, parameters, or search algorithms inadequate, they have the option to develop and incorporate their own into the Tychonis framework. In this way, these novel additions can be subsequently employed by other missions, thus transforming Tychonis into a repository of algorithms and opportunities that grows over time. This scenario paves the way for Tychonis to serve as a multi-mission platform that allows for coordinated, divide-and-conquer development of capabilities that can be shared among several missions.

Assuming a serial but iterative augmentation-and-use cycle for Tychonis in the context of a mission, it is probable that the initial iterations of new opportunities and search algorithms may require enhancements. Nonetheless, as the mission progresses through its phases, reviews, and frequent utilization of the software, the Tychonis classes will undergo improvements and fixes. By the time the mission attains a more advanced stage, the codebase will have demonstrated

robustness and effectiveness, rendering the incremental additions of high quality and potentially reusable by subsequent missions. Such development will serve as an incentive for later missions to abstain from developing capabilities that are already operational within the framework, thereby reducing cost and risk. It is accepted within NASA that employing code that has been validated and is readily available is a less costly and less risky proposition than crafting code from scratch at the outset of a new mission[29].

Opportunity	`Query` Class Name
Angular separation between two bodies, from an observer, satisfies a condition.	`AngularSeparationTimeQuery`
Distance between target and observer satisfies a condition.	`DistanceTimeQuery`
Observer sees a target occulted by another body.	`OccultationTimeQuery`
Observer - target position vector satisfies a condition.	`PositionTimeQuery`
Range rate between target and observer satisfies a condition.	`RangeRateTimeQuery`
Coordinate of the subobserver point on a target satisfies a condition.	`SubPointTimeQuery`
Coordinate of a surface intercept vector satisfies a condition.	`SurfaceInterceptTimeQuery`
Target enters the field of view of a spacecraft's instrument.	`TargetInFOVTimeQuery`
Logical AND between two opportunities (union of resulting time intervals).	`AndTimeQuery`
Logical OR between two opportunities (intersection of resulting time intervals).	`OrTimeQuery`
Logical NOT of one opportunity (complement of resulting time intervals).	`NotTimeQuery`

Table 1. List of default Tychonis opportunities and their class names. The most relevant attributes or parameters for each opportunity are underlined in each entry of the Opportunity column. The last three opportunities are Boolean operators applied to one or two other opportunities.

[29] NASA often reuses and adapts *heritage code* from previous missions in new missions. This approach can lead to cost savings and reduce development time, as the code has already been tested and proven in real-world scenarios. Additionally, using familiar code can minimize the risk of errors and improve reliability.

2.4 Verification and Validation

The verification process of Tychonis is, in part, implicitly facilitated by EMF's code generation capabilities, which ensures adherence to the constraints specified in the Ecore model. Ecore-based classes are created by users, which contain parameter types, inter-class relationships, cardinalities, and other model constraints. These constraints are directly reflected in the code generated by EMF and can be evaluated using the `Diagnostician`[30] class provided by EMF. The `Diagnostician` class includes a `validate()`[31] method that confirms whether the parameters of an EMF-generated Java class conform to the constraints defined in the Ecore model. For example, in an Ecore-based class designed to model opportunities involving two bodies within a certain distance, it is essential to ensure that a valid distance constraint is always present. If verification fails, the calling code is notified with either a warning or an error that specifies the exact location of the failure. This information is highly detailed and is particularly useful when applications that integrate with Tychonis need to validate user-created `Query` instances against their constraints prior to them being subjected to a `Solver`. It is worth noting that such code would typically be developed manually by users. However, EMF offers this functionality without requiring any additional human input.

Tychonis' default `Solver` classes and their algorithms are verified through unit testing by means of JUnit[32]. The most important checks the unit tests perform are to ensure opportunity search algorithms, for a known input, always return the same known output (e.g., time windows, error codes). This approach effectively guards against any deviations from the expected output, be it due to

[30] The `Diagnostician` class is a diagnostic reporting utility in the Eclipse Modeling Framework (EMF) that helps identify potential issues with model instances during validation. It is used to diagnose and report any issues with a given model instance against a set of validation rules specified in the corresponding Ecore metamodel.

[31] The method returns a `Diagnostic` object that contains the validation results, including any errors, warnings, or information messages that were generated during the validation process.

[32] JUnit is a popular open-source testing framework for Java programming language that is widely used in software development to write and run repeatable and automated tests. JUnit provides a set of annotations and assertions that help developers write test cases and ensure that their code is working as intended.

modifications to existing algorithms or the inclusion of new algorithms for a given opportunity type. It is expected, if users develop new `Solver` classes, that they would write unit tests that can become part of the Tychonis framework. The ability to develop unit tests solely for Tychonis speaks to the inherent capacity of the framework to be tested independently of its integration with any external applications.

Over time, Tychonis users are likely to introduce novel capabilities and refine existing ones. As these capabilities are applied in the context of missions, they may be iteratively enhanced. Notably, Tychonis provides a clear separation of concerns such that scientists and developers can discuss the parameters of an opportunity and the corresponding search algorithm for the opportunity as distinct conceptual entities. This represents a significant departure from some existing opportunity search software that treats modeling and search as a singular step. Tychonis' separation between opportunities and algorithms enables the explicit definition of requirements that can be verified through unit testing and validated through the science tools that integrate with the framework. In situations where opportunities or algorithms do not align with the scientist's original intent, the developer and scientist may collaborate to refine these components, which can then be independently deployed without impacting any software integrated with Tychonis.

2.5 Textual Language

While Tychonis alone demonstrates the feasibility of creating an extensible and reusable opportunity search framework, there can be challenges to its adoption by individuals without a background in software development. To search for a geometric event, an individual could generate custom code that employs the Tychonis framework, compile the code, and execute it. However, this approach can result in the code having a transitory existence (*à la* missions scripts) since it may only be applicable to a single event. Another alternative is to integrate a geometry software library, such as Tychonis, with a host application that can

templatize geometric events. In such a scenario, the user can initiate the execution of numerous event searches via the host software. This pattern will be described in Section 3.5, where SOA users model their events via a Graphical User Interface[33] (GUI) and then request the software to search for the events using Tychonis. However, space missions often operate in cost-constrained environments, which can limit the availability of personnel and access to software developers who can create code to search for specific events. Additionally, science teams may lack access to advanced science planning software like SOA that enables the templating of events for streamlined searches.

Given these limitations, we propose textual computer languages in a subsequent chapter to model and search for geometric events. Our goal is to develop a language that (i) provides unambiguous textual representations of space-based geometric events and (ii) can be utilized by space mission scientists and engineers with a moderate level of programming experience. In the pursuit of this objective, we designed two approaches for the textual language and evaluated their usability through a user-centric study.

[33] A GUI, or Graphical User Interface, is a type of interface that allows users to interact with a software application or an electronic device through graphical elements, such as icons, menus, windows, and buttons, rather than through text-based command lines.

3 Tychonis: Metamodel

Tychonis' approach to user extensibility is consistent with other multi-mission tools employed in the domain of mission planning and execution, which typically separate the core functionality of the software from mission-specific applications that employ the software. For instance, APGen [6] is a tool that simulates spacecraft activity plans, evaluates the impact of those plans on spacecraft subsystems, and schedules corresponding activities for execution. The software architecture for APGen includes a *core* set of capabilities that encompasses the simulator, the language used to define spacecraft behavior models, and the user interface, among other features. Additionally, the software incorporates an *adaptation* layer that enables mission-specific capabilities such as unique spacecraft behavior models, spacecraft states, and mission schedulers. Development of the core capabilities is the responsibility of multi-mission developers who are tasked with releasing new versions of APGen, whereas adaptation layer development and execution of the simulator and schedulers are the purview of individual space missions. By leveraging a similar approach, Tychonis is able to provide a customizable and adaptable framework that allows for efficient and effective mission planning and execution.

Tychonis allows for extensibility and reusability via an object-oriented metamodel [31]. The choice of the term *metamodel* is not arbitrary, as Tychonis' metamodel functions as a means of constructing ontological types[34] of opportunities and search logic. It is important to note that while Tychonis incorporates a metamodel, it also includes models implemented with its metamodel. Unless users are extending the framework, they interact within model space, where they (or software under their control) develop models that conform to generalized

[34] An ontological model of a given domain describes the types of entities that exist in that domain, their properties, and the relationships between them.

constructs specified in the metamodel[35]. In this document, we will refer to both the metamodel and model components of Tychonis as the larger metamodel.

As previously described, Tychonis uses EMF to define its metamodel. EMF, in turn, provides the Ecore modeling capability[36], which is similar to UML in that it lets developers create relationships between structured classes that represent different concepts. EMF, however, goes beyond the creation of abstract models and also provides capabilities to generate Java classes directly from a model. In the Tychonis framework, the generated Java classes serve as a faithful reproduction of the metamodel, allowing for the creation of geometric opportunity types that conform to the templated representations within the metamodel. EMF offers a number of additional advantages, including the ability to validate user objects, package a model and its generated classes as a Java library, modify a model during run-time, and introspect the model definition and its relationships. These capabilities make it easier for end-users to extend Tychonis' definition of opportunities or its search algorithms, or for external software to integrate with Tychonis.

There are four main types of constructs within the Tychonis metamodel:

- `Query` classes define opportunities, their parameters, and their relationship to other Tychonis classes. A `Query` class can represent purely geometric opportunities, but it could also connect two or more `Query` classes through Boolean relations (AND, OR, NOT). The use of Boolean relations is pertinent when modeling composite opportunities and will result in a tree of `Query` instances. `Query` classes are typically

[35] To elaborate further, a metamodel is a model that describes the structure and constraints of other models. It defines the concepts, relations, and rules that apply to a particular domain of interest. In contrast, a model is an instance of a metamodel that represents a particular system, process, or artifact in that domain. Tychonis is the metamodel that provides the blueprint for creating models, and users create models that capture the specific details and characteristics of a particular opportunity.

[36] In a sense, Ecore is Tychonis' metamodel, hence Ecore can be considered a meta-metamodel. This is analogous to a computer language to construct computer language grammars, which can also be considered a meta-metamodel.

instantiated by software that implements Tychonis and are developed by mission users who need to add new opportunity types to the framework.

- `Solver` classes contain the algorithm that searches for time windows in which an opportunity, defined within a `Query` instance, occurs. `Solver` classes create `Result` instances that contain these time windows. A `Solver`, in general, only applies to a specific `Query` class. Solver classes are developed by individuals knowledgeable about geometric methods or algorithms and may use libraries such as SPICE or MONTE.
- `SolverStrategy` classes (1) create and assign `Solver` instances to `Query` instances, (2) execute the `Solver` opportunity search class code, and (3) return a `Result` instance back to the calling software. Given opportunities can be composite and defined as a tree data structure, `SolverStrategy` classes also include the algorithm that walks a `Query` tree hierarchy and executes the previous (1), (2), (3) sequence for each `Query` instance. `SolverStrategy` classes are instantiated by software that uses Tychonis in order to obtain the time windows that correspond to an opportunity, be that opportunity atomic (a single `Query` instance), or composite (a tree of `Query` instances). `SolverStrategy` classes are developed by users who understand what `Solver` classes apply to specific `Query` classes, and who know how to develop or optimize a resolution algorithm for composite opportunities.
- The `Result` class describes the values returned by `Solver` and `SolverStrategy` instances in their search for opportunities modeled by `Query` instances. Result instances are created by `Solver` classes and later processed by other software in order to provide users with information about searched opportunities. They typically contain time windows that describe when a `Query` instance takes place, but they can also contain other types of contextual data relevant to the time windows such as celestial bodies or spacecraft involved.

The partitioning of modeling and resolution concerns in Tychonis software is a direct response to the imperative of addressing our research question. `Query` classes embody the definition of opportunities and their interrelationships, while `Solver` classes encapsulate the algorithmic logic that searches for atomic opportunities modeled in `Query` instances. `SolverStrategy` classes are responsible for assigning `Solver` instances to `Query` instances, executing `Solver` opportunity search class code, and resolving composite opportunities. The `Result` class encapsulates data returned after an opportunity has been searched. By delineating the software into these separate constructs, we are able to establish clear boundaries between different functionalities and support the diverse use cases for which the software was designed.

3.1 Use Cases

Tychonis facilitates three categories of use cases, namely Integration use cases, End-user use cases, and Maintenance use cases, which are illustrated in Figure 11. The users in Integration and Maintenance cases are developers, whereas the targets for End-user cases are mission engineers that need to search for opportunities. These use cases were derived from the inspection of current opportunity search software such as SOA, SPICE, Percy, and WebGeoCalc.

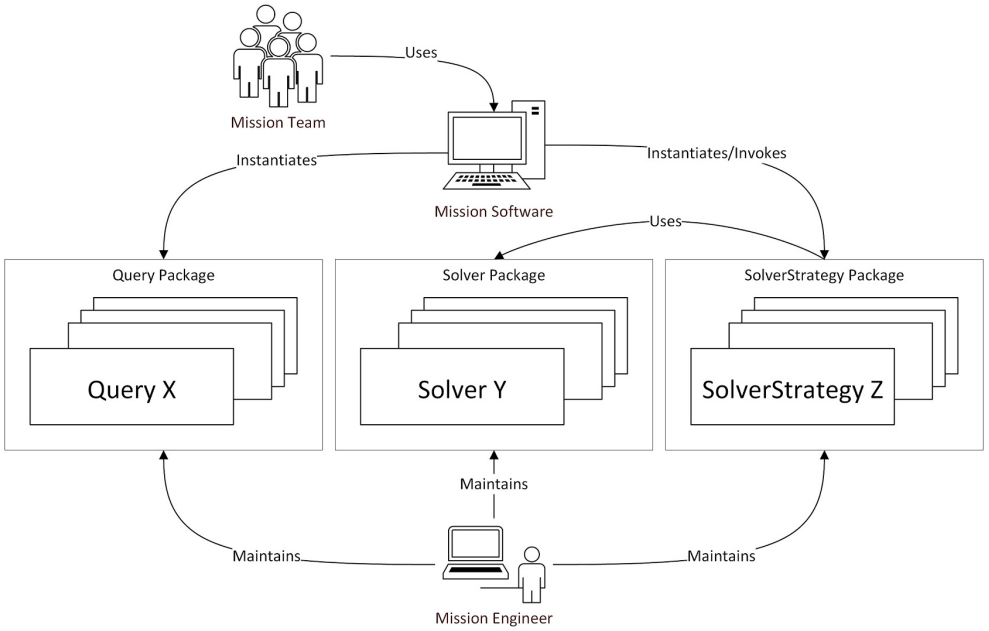

Figure 11. Metamodel Use Cases. End-User Use Cases (top) A mission team operates mission software that integrates with Tychonis. Mission software instantiates `Query` classes that represent an opportunity. The same or other software invokes a `SolverStrategy` to resolve the opportunity. Maintenance-Driven Use Cases (bottom): An engineer maintains - adds, deletes, or modifies - `Query`, `Solver`, and `SolverStrategy` classes based on the needs of a mission. Changes are propagated without the need to recompile mission software.

Integration use cases cater to developers and involve acquiring the Tychonis framework from a repository and writing computer code to facilitate the connection between mission software and Tychonis. Mission software can accommodate diverse means of input and representation of opportunities, which include textual descriptions, form-based descriptions, and projectional editors[37] (also referred to as structured editors or syntax-directed editors) [32], that enable a visual representation of the opportunity's structure. Integration necessitates either the development of code that employs Tychonis' `Query` class-based representations directly or the implementation of logic that transforms the constructs of mission software into Tychonis' representations. Upon completion

[37] A projectional editor is a type of software tool that allows users to edit code directly through a projection of the abstract syntax tree of a programming language, rather than through a text-based representation of the code. This approach is in contrast to traditional text-based editors, which require the user to enter code using a specific syntax and structure. Projectional editors can make it easier for developers to work with complex languages or domain-specific languages, as well as facilitate automation and code generation.

of integration, mission software will no longer require updating and recompiling with each modification of the metamodel and the subsequent generation of new Java classes. Software that integrates with the framework can query the framework to determine the availability of `Query` or `Solver` classes and, therefore, update its comprehension of the opportunities it can search for, and present the same to the user.

End-user use cases are interaction-oriented and can be seen in Figure 11, top half. These use cases outline how Tychonis is invoked by other software as a result of user actions. A mission engineer, using mission software that has already been integrated with Tychonis, will model opportunities on a user interface. This process results in the instantiation of `Query` classes by the mission software. Following this, the user can instruct the mission software to conduct a search for an opportunity, which will trigger the instantiation of a `SolverStrategy` class and commence the resolution process of the opportunity. Subsequently, Tychonis returns an instance of a `Result` class, which provides all the relevant data pertaining to the discovered opportunity. The data contained within the `Result` instance can then be utilized directly by the mission team or displayed on the UI by the calling software.

Maintenance use cases are developer-oriented and aimed at enhancing Tychonis' capabilities by adding, modifying, or deleting `Query`, `Solver`, and `SolverStrategy` classes. The framework offers several built-in `Query` classes that define commonly used opportunities, along with a number of `Solver` classes and a pre-existing `SolverStrategy`. However, if users are not satisfied with the built-in classes, they can make the necessary modifications by first altering the Ecore metamodel, followed by generating new Java versions of the modified classes or developing entirely new classes via EMF, and eventually supplementing code to the generated `Solver` or `SolverStrategy` Java classes. It is crucial to note that `Query` classes solely represent a data model with no algorithms, hence limiting the need for modifying the generated Java classes.

3.2 Opportunities

Semenov [21] expounds on a number of opportunities that can be examined using the WebGeoCalc software, such as Position, Angular Separation, Distance, Sub-Point, Occultation, Surface Intercept, Target in Field of View, and Ray in Field of View[38]. To facilitate requirement development and testing, Tychonis is equipped to model and resolve these opportunities by default. While the aforementioned opportunities within WebGeoCalc are atomic, enabling users to search for only one opportunity at a time, authors of [2] outlined how SOA can amalgamate atomic opportunities into larger opportunities using Boolean logic. Tychonis has followed this notion and developed a set of `Query` classes within its metamodel that reference and relate `Query` instances hierarchically in a tree. It is worth noting that, presently, only Boolean `Query` classes contain such relationships, as the built-in geometric `Query` classes are considered to be terminal within the tree structure. Figure 12 shows an example of a composite opportunity named Qx that contains several Boolean relationships.

[38] The Position event in spacecraft operations indicates the location of the spacecraft at a specific time, whereas the Angular Separation event represents the angle between two celestial objects as observed by the spacecraft. The Distance event specifies the distance between the spacecraft and a target object, while the Sub-Point event indicates the location on the surface of a celestial body directly below the spacecraft. The Occultation event is the period when the spacecraft is obscured by another object, such as a planet, while the Surface Intercept event represents the moment when the spacecraft passes over a specific location on the surface of a celestial body. The Target in Field of View event occurs when a specific target is visible in the spacecraft's sensor field of view, while the Ray in Field of View event indicates the moment when a sensor ray intersects a celestial object's surface.

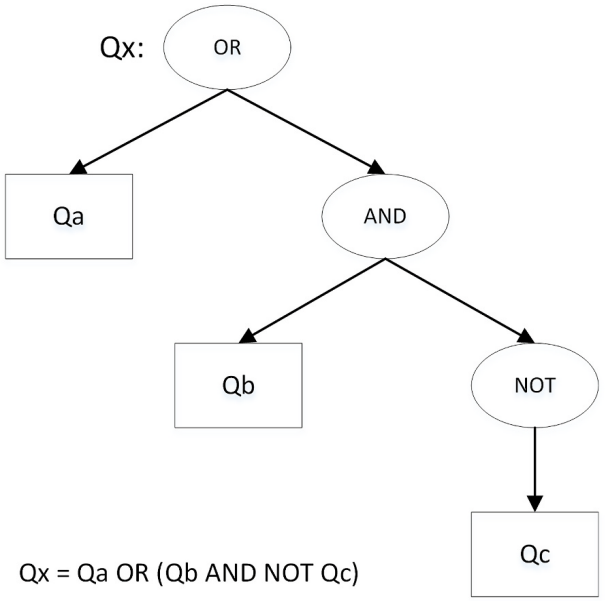

Figure 12. Example of a Composite Opportunity. Ellipses represent Boolean `Query` instances whereas rectangles represent terminal geometric `Query` instances.

3.3 Structure and Extensibility

This section describes in more detail the most relevant classes involved in Tychonis' use and extension from an object-oriented modeling perspective. Generally, extending Tychonis components entails a repeatable process whereby the user modifies the Ecore metamodel by including classes that either implement an interface or extend an existing class, then generates Java classes through EMF. Next, the user proceeds to develop algorithms within the generated classes before compiling and packaging the new classes into a JAR file. It is noteworthy that the said workflow obviates the need for recompiling mission software to avail of the new additions.

The subsequent discussion will focus on the key relationships and core classes. It should be noted that the Ecore modeling framework provides its own terminology to define metamodels, such as `EClass` for Ecore classes and `EReference` for a reference from an `EClass` to another `EClass`. Although

these Ecore-specific names will not be discussed directly, it is expected that the use of these object-oriented constructs refers to their Ecore equivalent. For a more comprehensive understanding of the varied constructs that Ecore users can use to generate their models, please consult Figure 13. Additionally, to enhance reading fluidity of the text, the *class* suffix may be omitted when referring to a class (e.g., `Query` instead of `Query` class), except when necessary. References to instance names will be followed by the word *instance* (e.g., `Query` instance).

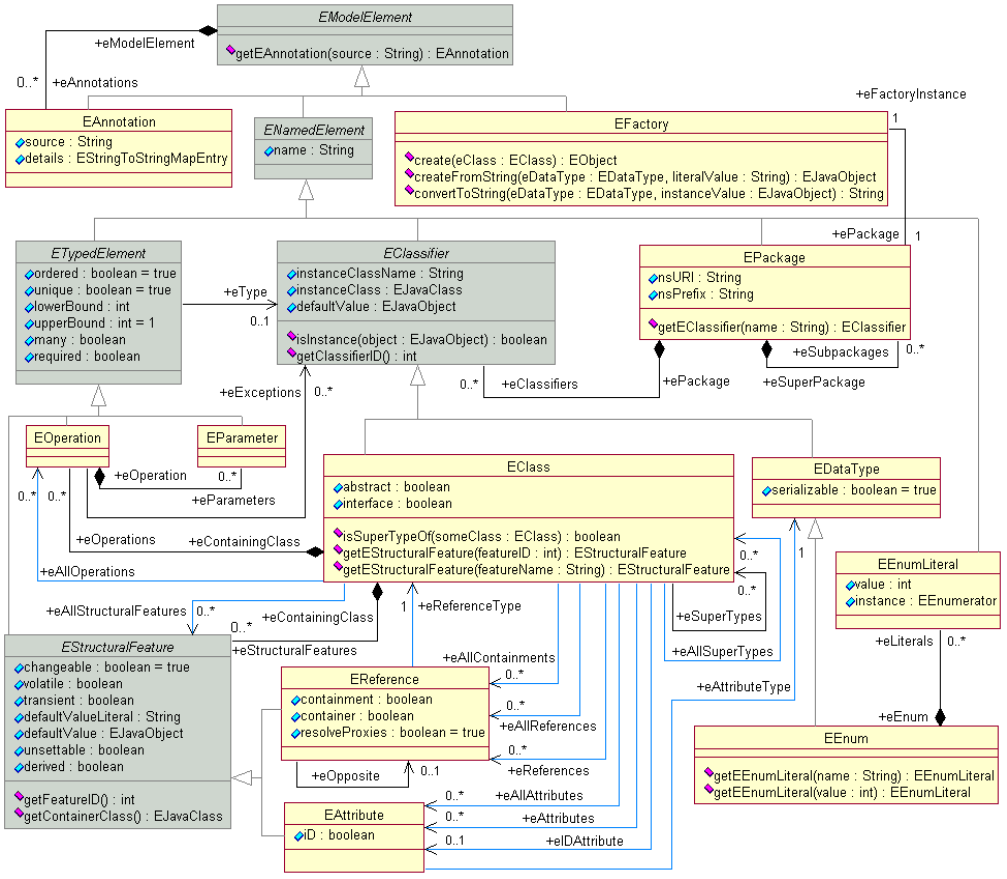

Figure 13. The Ecore classes and their relationships. Ecore offers its users a range of classes that can be utilized for creating customized models. Once users have implemented their models, EMF has the capability to generate Java classes that encapsulate the user-defined model. These generated classes possess advantageous methods that enable verification of instance validity, and facilitate introspection of the internal relationships within the model.

3.3.1 Query

In its definition of the `Query` metamodel, as depicted in Figure 14, Tychonis provides the `Query` base class, which contains the coordinate system used by parameters of the query, along with a reference to `Result`. `Result` is the class that defines how results of a `Query` instance are propagated back to software that integrates with Tychonis. A `Result` object is referenced from a `Query` object only after a `Solver` instance has searched for the opportunity modeled by that `Query` object.

`Query` is extended by `UnboundedTimeQuery` and, in turn, `UnboundedTimeQuery` is extended by `TimeQuery`. The distinction between `UnboundedQuery` and `TimeQuery` is that `UnboundedTimeQuery` does not establish a time frame for defining an opportunity, whereas `TimeQuery` bounds an opportunity with a start time and an end time. The latter is essential to avoid exploring an infinite search space. Moreover, Boolean `Query` classes do not need to specify a time boundary, given that it is implicit in the query's reference to other queries and in the execution of the Boolean operation. Therefore, `TimeQuery` serves as the parent class for the built-in atomic geometric `Query` classes, while `UnboundedTimeQuery` acts as the parent class for the built-in Boolean `Query` classes. If a user opts to include a new `Query` type, the initial step would be to assess the desirability of expanding the `UnboundedTimeQuery` or `TimeQuery` classes, built-in geometric or Boolean `Query` classes, or mission-developed `Query` classes. The selection would be contingent on the degree of code reuse the user desires.

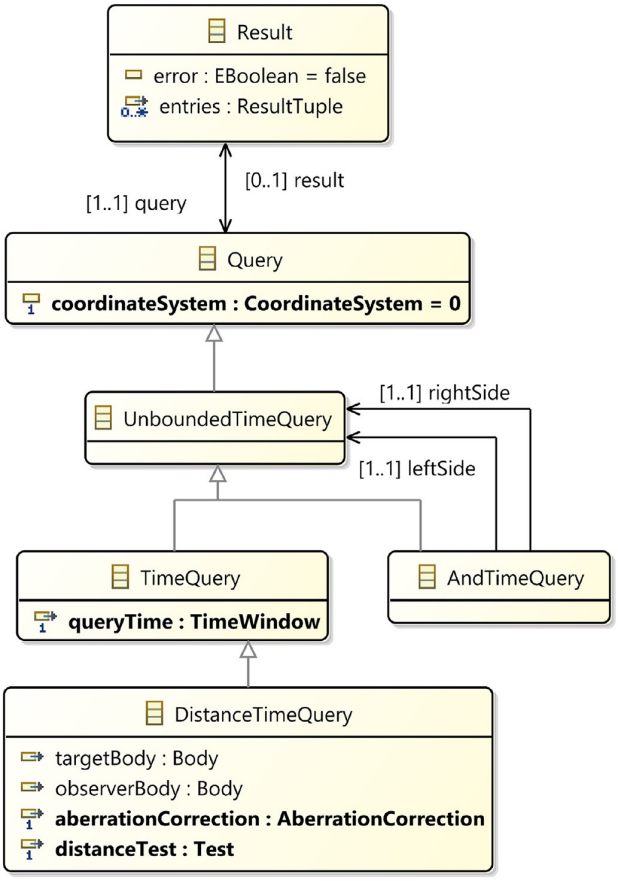

Figure 14. The `Query` Class Hierarchy. The Query class hierarchy with two example extensions, DistanceTimeQuery and AndTimeQuery.

Some concrete examples of extensibility could be:

1. `DistanceTimeQuery` is a geometric query that models opportunities in which a body is located at a specified distance relative to another body. This query extends the `TimeQuery` class and references two `Body` instances (target and observer), a `DistanceTest` instance that defines the distance inequality, and an `AberrationCorrection` instance for light speed and relative velocity magnitude corrections.
2. `AndTimeQuery` is the built-in Boolean query that models an AND relationship between two other `Query` instances. As such, `AndTimeQuery` extends `UnboundedTimeQuery` and references two other `UnboundedTimeQuery` instances that capture the two

sides of the AND operation. The selection of `UnboundedTimeQuery` as the parent class for `AndTimeQuery` is motivated by the fact that the Boolean `Query` classes have no need to explicitly define a search time window.

3.3.2 Solver

`Solver` contains the code that resolves a `Query`. Typically, there should be at least one `Solver` class for each `Query` class, although in certain cases multiple `Solver` classes may be available for a specific `Query` class. For example, when users are experimenting with new `Solver` implementations that are more efficient or use new libraries, it may be necessary to maintain multiple `Solver` classes until the new implementation is confirmed to be an improvement over the previous one in terms of accuracy and performance. The decision on which `Solver` to use for a specific `Query` is made by `SolverStrategy`.

`Solver` is an interface that provides an extension point used by Tychonis' built-in `Solver` classes and `Solver` classes developed by end-users. Classes that extend the `Solver` interface must implement both the `solve()` and `validate()` methods. The `solve()` method defines the search algorithm for the given `Query` instance, while the `validate()` method verifies that the `Query` instance can be resolved by the `Solver` class. The `validate()` method returns a list of `ValidationResult` objects containing information on the validation results, such as whether validation passed or not, any sources of errors, and human-readable messages. To ensure a viable resolution, the `solve()` method internally calls the `validate()` method.

Figure 15 shows the described structure, which includes two extensions, `DistanceTimeQuerySPICESolver` and `AndTimeQueryBasicSolver`, both of which are built-in `Solver` classes. The former employs the SPICE toolkit to identify time windows for `DistanceTimeQuery` objects, while the latter implements an intersection of

time windows from the two sides of an AND relationship defined in an `AndTimeQuery` object. It is worth noting that although there are other means to implement a `Solver` for `DistanceTimeQuery` objects, SPICE was chosen for implementation, as with other `Solver` classes provided in Tychonis. Furthermore, `DistanceTimeQuerySPICESolver` has the capability to search for distance opportunities even when either one or both `Body` parameters in a `DistanceTimeQuery` instance are unknown, as indicated by a `null` Java value. When a `Body` instance is set to `null`, the resolution algorithm within `DistanceTimeQuerySPICESolver`'s `solve()` method will search for all bodies loaded by `Solver` that satisfy the `distanceTest` constraint within the `DistanceTimeQuery` instance.

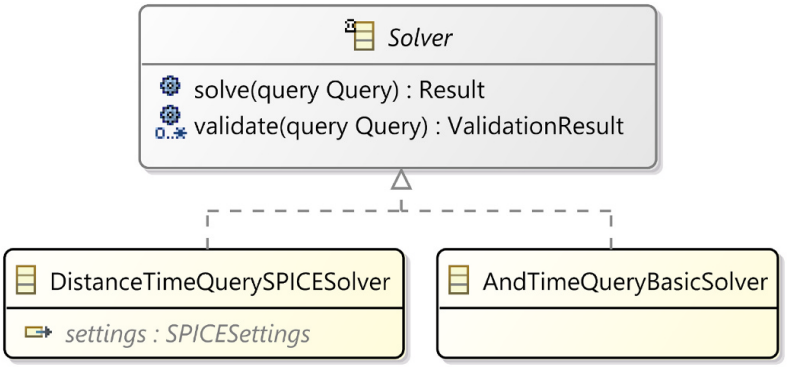

Figure 15. The `Solver` Interface. The `Solver` interface with its two methods and two built-in `Solver` classes: a `DistanceTimeQuerySolver` implemented with SPICE and an `AndTimeQuerySolver`.

3.3.3 SolverStrategy

The `SolverStrategy` interface constitutes a pivotal extension point that facilitates the development of the underlying logic for `Query`-to-`Solver` assignments, tree traversal algorithms for multiple `Query` instances, and the execution of `Solver` code for each `Query` node. The implementation of this interface requires the incorporation of a `solveFrom()` method that receives

the root `Query` instance within the tree structure to generate a `Result` instance. A graphical representation of this concept is presented in Figure 16.

Tychonis, as a software framework, offers a built-in implementation of SolverStrategy referred to as `SimpleSolverStrategy`. This pre-existing implementation, along with its associated helper classes, serves as a valuable exemplar of `SolverStrategy` extensibility. Further elaboration on this implementation can be found in the following points:

- The implementation of `SimpleSolverStrategy`, which adheres to the `SolverStrategy` interface, entails the resolution of a `Query` tree through a hierarchical process. This process first resolves the leaf nodes followed by the upward propagation of the resultant values. This resolution procedure continues iteratively, culminating in the resolution of the top `Query` node passed as a parameter. However, it is pertinent to note that the current implementation of this resolution algorithm operates in a serial fashion. Despite the potential for parallelization, given the data independence across nodes, parallel processing is not currently incorporated.
- `QuerySolverMappingProvider` is an interface used by `SimpleSolverStrategy`. The purpose of `QuerySolverMappingProvider` is to provide a `Solver` instance for a specific `Query` instance through a method known as dependency injection [33]. Dependency injection proposes that the implementation of at least one of two or more dependent components should be realized at runtime via another component[39]. In this case, `QuerySolverMappingProvider` is the component that provides a `Solver` instance for a `Query` instance at runtime. `QuerySolverMappingProvider` is an interface in order to enable

[39] Dependency Injection allows developers to create more modular, maintainable, and testable software by decoupling object creation and object usage. Instead of having objects create their dependencies directly, those dependencies are *injected* into the object by an external framework or container.

user extensibility, as missions might want to provide `Query`-to-`Solver` dependencies through various means.

- `CSVQuerySolverMappingProvider` is a class that implements the `QuerySolverMappingProvider` interface. It retrieves the mappings from a Comma-Separated Values[40] (CSV) file and provides a `Solver` through the `getSolver()` method defined in the `QuerySolverMappingProvider` interface.

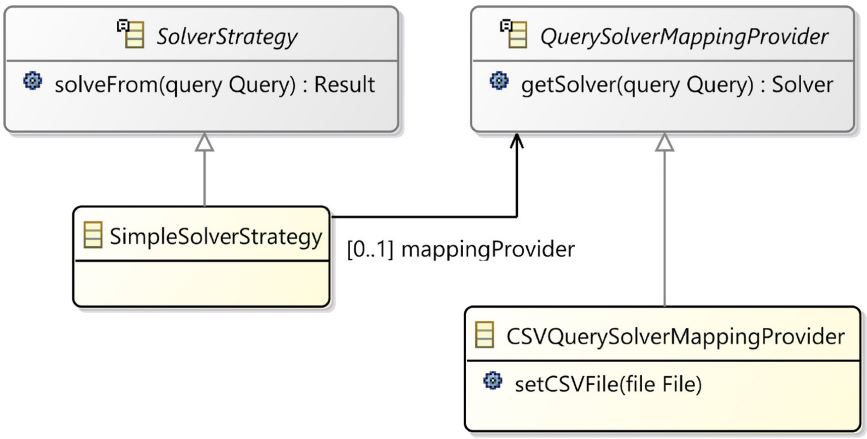

Figure 16. The `SolverStrategy` Interface. The SolverStrategy interface with `SimpleSolverStrategy` as its default implementation. Tychonis also provides extensibility for `SimpleSolverStrategy` by letting users implement their own dependency injection provider via `QuerySolverMappingProvider`.

3.3.4 Result

`Result` is used to store the findings of opportunity search algorithms within a `Solver` class. Within each `Result` instance resides a reference to the `Query` instance it contains result data for. In turn, each `Query` object also contains a reference to its `Result` instance, thereby establishing a reciprocal relationship between the two. It is important to note that the responsibility of defining this

[40] Comma-Separated Values (CSV) is a file format used to store tabular data, in which each row represents a record, and each column represents a field within that record. In a CSV file, the values are separated by commas, and each record is separated by a newline character.

bidirectional association between a `Query` instance and a `Result` instance lies with `SolverStrategy`, since the creation of `Result` objects falls under the purview of `Solver`.

The `Result` class is not intended for user modification or extension. Nevertheless, it is endowed with the capacity to reference distinct data types or classes. This is attributable to the fact that, while contents within a `Result` instance typically contain time windows in which an opportunity takes place, they can also describe other data depending on the `Solver` class used by an implementer of `SolverStrategy`. The ability to remain agnostic to data types is achieved through the employment of a tabular format to describe results, with each cell of the table potentially represented by an instance of a class that implements the `Resultable` interface. The end result is that `Solver` developers are empowered to determine the nature of the data returned by a given `Solver` implementation with minimal constraint.

In order to structurally describe the result table, `Result` contains a list of references to `ResultTuple` objects that function as the rows of the table. Each `ResultTuple` object within a given `Result` instance, in turn, references instances of classes that implement the `Resultable` interface. The implementation of the `Resultable` interface is established via a <key,value> pair utilizing a `String` key within each `Resultable` instance, and a `String` value via the `stringValue()` method, which necessitates implementation by the user. Typically, the key in the <key,value> pair is utilized to clarify the type of data that a `Resultable` instance contains, while the value offers a serialized textual representation of the object that can be accessed by external software. This is exemplified in the `Body` class utilized to designate space bodies, which itself implements the `Resultable` interface. Specifically, `Body` instances are utilized to define relationships between bodies in `Result` instance rows, such as <"*target*","*Venus*"> and <"*observer*","*Cassini*">. Additionally, Tychonis provides methods for searching data types within `Result` instances, as will be expounded upon in Section 3.4.3.

3.4 Searching for Opportunities

After laying the groundwork by elucidating the relevant classes involved in Tychonis' application and extension, we present a systematic approach for utilizing the framework to locate opportunities. This approach comprises three steps: (i) opportunity modeling, (ii) resolution, and (iii) result parsing. At a user level, Tychonis-enabled software typically affords assistance for these steps in a more interactive manner: users manipulate a user interface to specify opportunities (modeling), choose an option to initiate a search (resolution), and subsequently view the search results (parsing). All such user interactions lead to internal instantiations of EMF-generated Java classes supplied by Tychonis.

An important improvement offered by the separation of the distinct software concerns (namely, modeling, resolution, and parsing) in Tychonis is its provision of a framework for treating needs as components that vary at different times for different reasons [34]. From a practical standpoint, scientists and engineers often find it challenging to comprehend code that intermixes modeling and resolution because they require expertise in both the scientific or engineering definition of an opportunity and in how a software library (e.g., SPICE, MONTE) carries out its search. Tychonis presents a solution to this issue.

To contextualize the preceding observations within the scope of the following discourse, this text provides examples of the three phases mentioned. Although these examples are instructive, they will be static, denoting that the code exhibited will not be parameterized in the same manner as it would be if integrated with real-world mission software. A seamless integration between Tychonis and mission software facilitates programmatic modeling of opportunities from user input, without necessitating the manual generation of code, as presented here.

In the interest of supplementary contextual background, integration with real-life mission software could involve the direct utilization of Tychonis' EMF-generated Java classes. Alternatively, an alternative path for integration would entail utilizing

an Ecore XML description file (tychonis.ecore), which encapsulates the complete Tychonis metamodel. Software reliant on this description file could leverage reflection [35] to instantiate classes from the Tychonis metamodel. This approach possesses the advantage that no modification (and no recompilation) of the software employing Tychonis would be necessary if the Tychonis metamodel is updated and its Java classes necessitate regeneration. The latter approach in question will be elaborated upon in a subsequent section.

3.4.1 Modeling

The goal of this step is to model or define an opportunity in unambiguous terms. To achieve this, the metamodel provides pre-built `Query` classes and allows users to define their own `Query` classes. The issue of ambiguity in defining opportunities is a crucial concern, particularly in fields where accuracy is paramount. The use of natural languages like English, which are inherently ambiguous, to describe opportunities can lead to varied interpretations by different recipients. For instance, consider this opportunity: "there is an occultation between Venus and the Sun as seen from Earth". A mission engineer might interpret this statement as Venus being in front of the Sun, while another person might think the front-back relationship is with the Sun in front of Venus. Moreover, the time window during which the opportunity can be searched is also a point of discussion. The role of a metamodel in these circumstances is to provide clarity as to what it means to capture domain-specific knowledge and set regulations about such a domain.

In continuation with the previous occultation opportunity example, it should be noted that the `OccultationTimeQuery` class, which is bundled within Tychonis, defines an atomic occultation opportunity as requiring a front body, a back body, an observer, a type (full, annular, or partial), an aberration correction, and a time window via its inheritance of `TimeQuery`. Given the stringent criteria stipulated by `OccultationTimeQuery`, engineers can formulate a more succinct rendition of the prior opportunity named O1 defined as: "there is

an occultation of any type between Venus - as back body, and he Sun - as front body, as seen from Earth anytime in 2019". The Java code that represents O1 is present in Listing 1, and it presupposes the utilization of default aberration correction and occultation type (i.e., any). It is important to note that, while the presented code accurately models an opportunity, it does not embody an integration with mission software that accounts for variable user input and parameterization. Furthermore, Listing 1 showcases several ancillary EMF idioms, capabilities, and auto-generated constructs, such as `TychonisFactory`, an auto-generated class that facilitates the creation of Tychonis class instances, which are subsequently populated with data via getters and setters. These getters and setters conform to the metamodel and are also generated automatically by EMF.

```java
Body frontBody = TychonisFactory.eINSTANCE.createBody();
frontBody.setName("SUN");
Body backBody = TychonisFactory.eINSTANCE.createBody();
backBody.setName("VENUS");
Body observerBody = TychonisFactory.eINSTANCE.createBody();
observerBody.setName("EARTH");
OccultationTimeQuery anOccultationQuery =
      TychonisFactory.eINSTANCE.createOccultationTimeQuery();
TimeWindow searchWindow =
      TychonisFactory.eINSTANCE.createTimeWindow();
TimeInstant startTime =
      TychonisFactory.eINSTANCE.createTimeInstant();
startTime.setTimeFromString("01/01/2019");
TimeInstant endTime =
      TychonisFactory.eINSTANCE.createTimeInstant();
endTime.setTimeFromString("12/31/2019");
searchWindow.setStartTime(startTime);
searchWindow.setEndTime(endTime);
anOccultationQuery.setFrontBody(frontBody);
anOccultationQuery.setBackBody(backBody); //line 16
anOccultationQuery.setObserverBody(observerBody);
anOccultationQuery.setQueryTime(searchWindow);
```

Listing 1. Modeling an Occultation Opportunity. Java code that models the O1 opportunity, defined as "there is an occultation of any type between Venus - as back body, and the Sun - as front body, as seen from Earth anytime in 2019".

Lastly, it is pertinent to remark that if a user opts not to specify one of the `Body` parameters in an `OccultationTimeQuery` instance, then that parameter will remain unknown and will necessitate further exploration. For example, if the developer omits line 16, the opportunity defined in `anOccultationQuery` would be articulated as "there is any type of occultation between any back body and the Sun, as viewed from Earth, at any point in 2019".

While O1 serves as a clear example of the Tychonis metamodel, it is plausible that it represents a comparatively simpler structural perspective of the potential possibilities enabled by the framework. For instance, one could conceive of another opportunity, denoted as O2, which reads as follows: "the distance between Earth and the Moon is between 370,000 km and 390,000 km in the last three months of 2019". This opportunity could be converted into a composite opportunity, such as "the distance between Earth and the Moon is more than 370,000 km in the last three months of 2019, AND the distance between Earth and the Moon is less than 390,000 km in the last three months of 2019". This composite opportunity is modeled in Listing 2 using two `DistanceTimeQuery` objects and one `AndTimeQuery` object.

It is worth noting that this particular example is not coincidental, as it raises the prospect of extending the metamodel beyond `Query`, `Solver`, and `SolverStrategy`. For example, as Figure 14 demonstrates, `DistanceTimeQuery` contains a reference to an instance of the `Test` class. Since the `Test` class is currently restricted to accommodating less-than or greater-than inequalities, a user may extend the class within the metamodel, designate it as the `BetweenTest` class, and integrate an upper and lower bound value. This would establish an *is between* relationship, obviating the need for the composite distance opportunity with an AND operator.

```Java
Body body1 = TychonisFactory.eINSTANCE.createBody();
body1.setName("EARTH");
Body body2 = TychonisFactory.eINSTANCE.createBody();
body2.setName("MOON");
TimeWindow searchWindow =
       TychonisFactory.eINSTANCE.createTimeWindow();
TimeInstant startTime =
       TychonisFactory.eINSTANCE.createTimeInstant();
startTime.setTimeFromString("09/01/2019");
TimeInstant endTime =
       TychonisFactory.eINSTANCE.createTimeInstant();
endTime.setTimeFromString("12/31/2019");
searchWindow.setStartTime(startTime);
searchWindow.setEndTime(endTime);
GreaterThanTest greaterThan370 =
       TestsFactory.eINSTANCE.createGreaterThanTest();
greaterThan370.setValue(3.7e5);
DistanceTimeQuery query1 =
       TychonisFactory.eINSTANCE.createDistanceTimeQuery();
query1.setTargetBody(body1);
query1.setObserverBody(body2);
query1.setDistanceTest(greaterThan370);
query1.setQueryTime(searchWindow);
LessThanTest lessThan390 =
       TestsFactory.eINSTANCE.createLessThanTest();
lessThan390.setValue(3.9e5);
DistanceTimeQuery query2 =
       TychonisFactory.eINSTANCE.createDistanceTimeQuery();
query2.setTargetBody(body1);
query2.setObserverBody(body2);
query2.setDistanceTest(lessThan390);
query2.setQueryTime(searchWindow);
AndTimeQuery andQuery =
       TychonisFactory.eINSTANCE.createAndTimeQuery();
andQuery.setLeftSide(query1);
andQuery.setRightSide(query2);
```

Listing 2. Modeling a Composite Opportunity. Java code that models the O2 opportunity with two `DistanceTimeQuery` objects and one `AndTimeQuery`.

3.4.2 Search

The process of searching for an opportunity involves the selection of one of the available `SolverStrategy` classes and the subsequent invocation of the `solveFrom()` method within the chosen `SolverStrategy`. In this context, the `solveFrom()` method receives the root `Query` node in the opportunity tree that the user intends to initiate the search from. An illustration of this process is demonstrated in Listing 3, where the code presented solves opportunity O2 from Listing 2. The responsibility of resolving the composite opportunity tree and linking each `Query` node with a corresponding `Result` instance that encapsulates the data generated by the `Solver` classes utilized, rests with the chosen `SolverStrategy`. It is pertinent to note that there are numerous approaches to implementing this logic, and as such, Tychonis offers users the flexibility to develop their own `SolverStrategy` beyond the pre-existing implementation.

Tychonis' built-in implementation is provided by the `SimpleSolverStrategy` class. It employs a post-order tree traversal algorithm [36] that systematically visits and resolves all `Query` nodes. Whenever the algorithm encounters an unsolved `Query` node that is solvable, i.e., all its dependent child `Query` nodes have already been solved or the node is a leaf node, it performs a lookup within a `QuerySolverMappingProvider` object. This object facilitates the binding of `Query` classes and `Solver` classes through the `getSolver()` method. Upon invocation, this method inspects the passed `Query` instance and subsequently returns a `Solver` object to the `SimpleSolverStrategy` class. The `Solver` object received is then utilized to solve the `Query` instance. This process continues successively from left to right and upwards in the tree structure until the root `Query` node - the one passed to `solveFrom()` - is also resolved.

Furthermore, it is important to discuss `QuerySolverMappingProvider`'s built-in implementation. The primary purpose behind the creation of

`QuerySolverMappingProvider` is to enable users to choose whether the mapping between a `Query` class and a `Solver` class should be statically defined in compiled code or dynamically defined via dependency injection. In the case that the mapping should be dynamic, `QuerySolverMappingProvider` specifies where this mapping should reside. A default implementation of `QuerySolverMappingProvider` is provided as `CSVQuerySolverMappingProvider`. This implementation reads a CSV file containing a row for each `Query` class name, followed by the name of the EMF factory class responsible for creating a `Solver` instance, and the name of the method in that factory that returns the `Solver` instance. This enables `CSVQuerySolverMappingProvider` to create a `Solver` instance via reflection when the framework requests a `Solver` instance with the `getSolver()` method.

```
Java
SimpleSolverStrategy strategy =
      TychonisFactory.eINSTANCE.
            createSimpleSolverStrategy();
CSVQuerySolverMappingProvider mappingProvider =
      TychonisFactory.eINSTANCE.
            createCSVQuerySolverMappingProvider();
mappingProvider.setCSVFile(new File("config/mappings.csv"));
strategy.setQuerySolverMappingProvider(mappingProvider);
strategy.solveFrom(andQuery);
```

Listing 3. Searching for Opportunity O2. Example code that resolves the O2 opportunity with the built-in SolverStrategy implementation.

3.4.3 Parsing Results

Once the search step is complete, the opportunity tree's `Query` objects contain references to corresponding `Result` objects generated by the `Solver` classes. At this point, mission software accesses the `Result` instance for the entire

opportunity tree by invoking the `getResult()` method on the root `Query` node. Thereafter, the `Result` object can be parsed by mission software to scrutinize its constituent `ResultTuple` instances, which can be further decomposed into `Resultable` objects. This structure, resembling a table, lends itself to hierarchical parsing. A concrete example of this approach can be found in Listing 4, which illustrates how the results for opportunity O2 can be parsed and displayed on a computer terminal.

In terms of extensibility, the `Result` table's flexibility arises from the fact that each cell holds references to objects implementing the `Resultable` interface. This feature allows engineers to decide what type of data is included in the table. Thus, should a mission require a new data type, an engineer would define a class implementing the `Resultable` interface and then modify an existing `Solver` or create a new one to include the new data type in the table. While this flexibility offers advantages, it might make retrieving data by type more complex. To address this issue, the `Result` and `ResultTuple` classes provide methods to query their contents. For instance, the `public <T> EList<T> getAll(Class<T> aClass)` method can be used to obtain all `TimeWindow` instances within a `Result` instance if invoked this way: `result.getAll(TimeWindow.class)`.

```java
Result result = andQuery.getResult();
for (ResultTuple tuple : result.getEntries()) {
	for (Resultable resultable : tuple.getEntries()) {
		printKeyValue(resultable.getKey(),
		resultable.stringValue());
		System.out.println();
	}
	System.out.println();
}
```

```
[key: target, value: EARTH]
[key: observer, value: MOON]
[key: result_window, value: {(2019-OCT-01 20:30:17.259, 2019-OCT-05 03:40:38.670)}]

[key: target, value: EARTH]
[key: observer, value: MOON]
[key: result_window, value: {(2019-OCT-18 01:35:32.289, 2019-OCT-22 19:10:37.216)}]

[key: target, value: EARTH]
[key: observer, value: MOON]
[key: result_window, value: {(2019-OCT-29 18:35:11.525, 2019-NOV-02 07:06:54.701)}]

[key: target, value: EARTH]
[key: observer, value: MOON]
[key: result_window, value: {(2019-NOV-13 17:51:00.865, 2019-NOV-20 13:11:38.668)}]

[key: target, value: EARTH]
[key: observer, value: MOON]
[key: result_window, value: {(2019-NOV-25 17:42:09.218, 2019-NOV-30 05:22:11.256)}]

[key: target, value: EARTH]
[key: observer, value: MOON]
[key: result_window, value: {(2019-DEC-10 11:50:46.855, 2019-DEC-27 15:25:18.377)}]
```

Listing 4. Parsing Results. Top section: Code that parses a `Result` instance and prints the table it contains out to the terminal. Bottom section: Terminal printout that results from the execution of the code above. Results might differ depending on `Solver` settings used (e.g., SPICE kernels).

3.5 Case Study: Integration with SOA

This section presents a case study for a potential future integration of SOA with Tychonis, aiming to evaluate Tychonis as a reusable and extensible framework. We will commence by discussing the integration patterns that Tychonis supports, followed by an analysis of the potential integration between SOA and Tychonis. To accomplish this, we will divide SOA's opportunity search capability into three major concerns, describe their operation, and present a likely integration path illustrated in Figure 17. The Modeling concern produces a `Query` tree, the Search concern produces a `Result` instance, and the Results Table yields a User Interface (UI) component displaying the contents of the `Result` instance. Integrating SOA with Tychonis necessitates the development of code that satisfies the actions in italics, as will be detailed in this text. This integration

approach is transferable to other tools that require the identification of geometric opportunities, such as ESA's Solar System Science Operations Laboratory (SOLab) [23], which has a capability named *Event Finder*, akin to SOA's opportunity search, or JPL's WebGeoCalc tool.

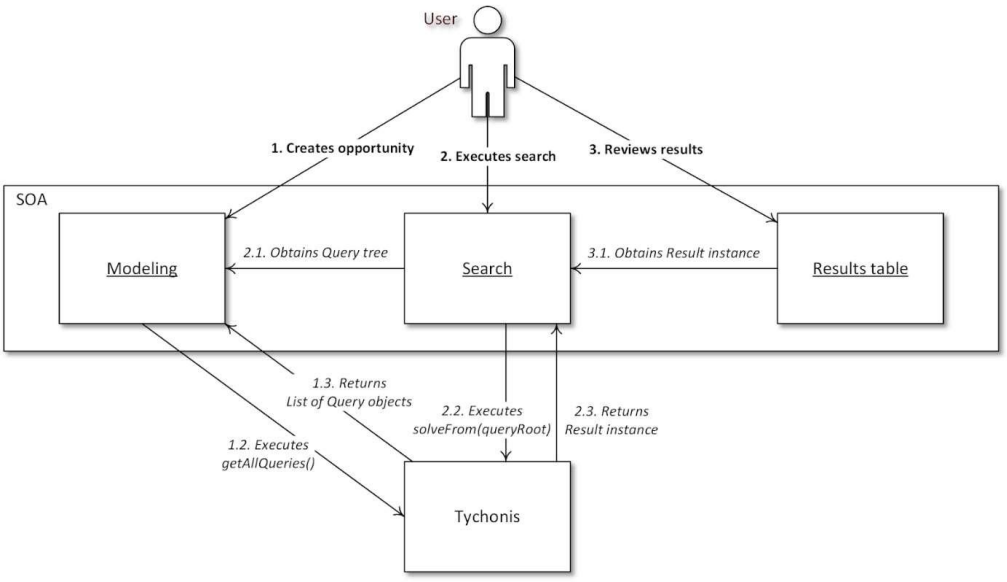

Figure 17. Searching for Opportunities in SOA. The SOA opportunity search process post-integration with Tychonis. The main user actions with SOA are in bold text. SOA's *Modeling*, *Resolution*, and *Results* table concerns are underlined.

3.5.1 Integration Pattern: Static vs. Dynamic

When integrating with an external library, developers typically instantiate classes and call methods and functions from the library by name. However, when using Tychonis, this approach does not automatically transfer updates to the metamodel to the host application. To illustrate this point, suppose that a Tychonis version lacks the capability to model and search for phase angle[41]

[41] Phase angle is an angular measurement defined by the position of three bodies in space: an observer, a target, and an illumination source. It is the angle between the direction of the observer as seen from the target, and the direction of the illumination source as seen from the target. The phase angle is important in astronomy and space exploration as it is used to determine the geometry of bodies in space, such as the position and orientation of a planetary or lunar surface with respect to the illumination source.

inequalities between illuminator, target, and observer bodies. To address this issue, a user may create a `PhaseAngleTimeQuery` class to model the opportunity, along with a `PhaseAngleTimeQuerySolver` to search for it. These classes would then be added to the Tychonis Ecore model, and their attributes defined. Following this, EMF would generate several Java classes, and the search algorithm would be added to a `PhaseAngleTimeQuerySolver` Java class.

However, how would a tool like SOA know that there is a new opportunity type it can search for? With this static API paradigm, SOA would only be aware of this new opportunity type if a developer augmented the SOA code that contains all opportunities by class name and added `PhaseAngleTimeQuery`. However, modifying the SOA code would entail a significant time commitment, as a new software release would need to be certified and provided to users. To solve this, Tychonis offers facilities for dynamic integration, which ease this burden in exchange for a more involved initial integration effort. These facilities involve writing code in the user application to ask Tychonis what opportunity types it supports and then offering those to users for their perusal. The following subsection will discuss how this occurs in greater detail.

3.5.2 Modeling

The *Opportunity Search Query Builder* is a tool provided by SOA that enables users to visually design opportunities. The tool consists of a palette that displays the available opportunity types and Boolean operators, a canvas where users can place opportunity types from the palette and link them with Boolean operators, and a table where users can set attribute values for the selected opportunities from the canvas. Creating an atomic opportunity involves dragging an opportunity type from the palette, dropping it on the canvas, and setting its parameters on the attribute table. To create composite opportunities, users repeat these steps for as many atomic opportunities as required and link them with Boolean operators. An illustration of this interaction can be found in Figure 18,

where SOA's Opportunity Search Query Builder UI is shown modeling the opportunity "within the period of April 23, 2015 to June 30, 2015, the phase angle formed by the celestial bodies of Dawn, Ceres, and the Sun exceeds 60 degrees, and simultaneously, the distance between Dawn and Ceres remains under 3,000 miles".

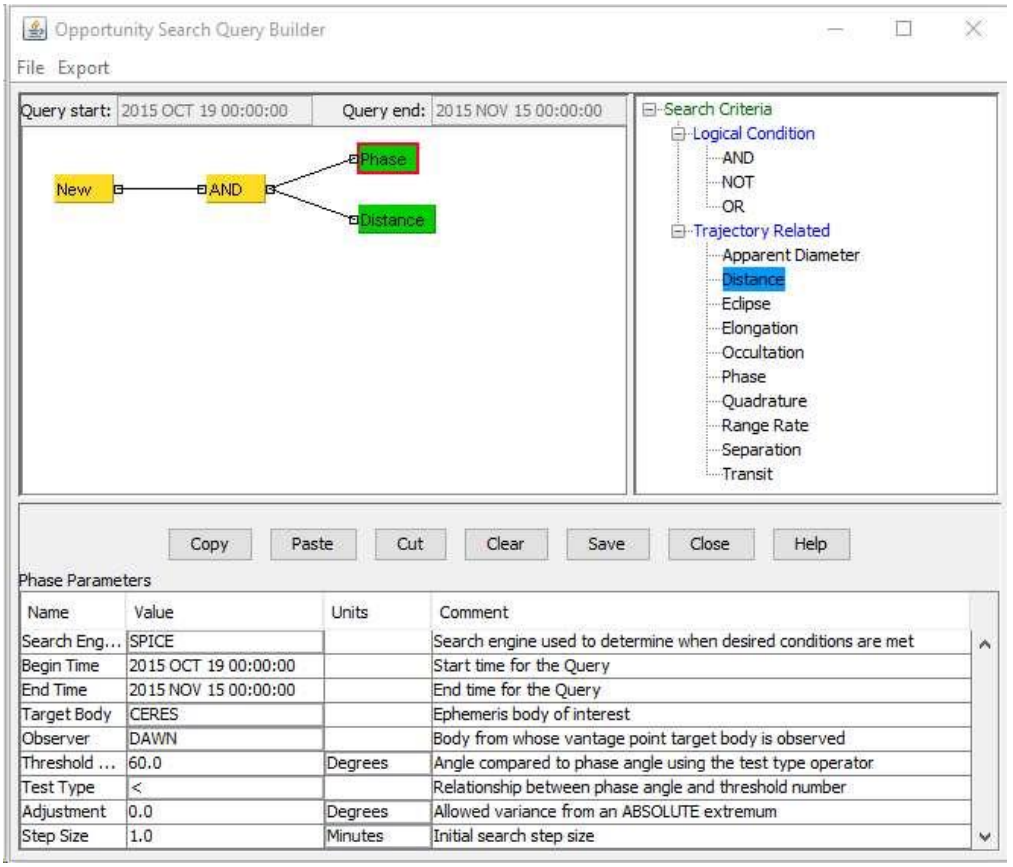

Figure 18. SOA's Opportunity Search Query Builder. The palette of atomic opportunities and Boolean operators (top-right corner) contains items that can be dragged and dropped into the opportunity design canvas (top-left corner) in order to create a composite opportunity. The attributes of a selected (red outline) atomic opportunity in the canvas can be modified from the attributes table (bottom).

If a static integration is desired, a developer needs to identify the opportunity types that need to appear on the UI. For this, the developer would gather the names of `Query` classes and the attributes they provide from the Tychonis version specifications. These opportunities would then be displayed on the

palette, and their corresponding attributes would be shown on the attribute table. However, as discussed, this approach would require the recompilation of SOA for any modification in opportunities.

Tychonis, as an innovation, promotes a more flexible and dynamic integration. It provides a facility to obtain (1) the names of all available opportunities - to show them on the palette, and (2) their attributes - to place them on the attributes table. To get the opportunities, Tychonis provides the `TychonisUtils.getAllQueries()` method, which returns a list of `Query` instances that comprises all atomic opportunities and Boolean operators. Using this method, a developer who integrates with Tychonis would be able to obtain all available opportunity types within the framework and show them on the palette. The UI-friendly name for each opportunity type can be obtained via `Query`'s `getHumanReadableName()` method, whose exemplifying return value for the `PhaseAngleTimeQuery` class is the string "*Phase Angle*". As for the internal operation of `TychonisUtils.getAllQueries()`, it parses the Tychonis Ecore XMI model file and reflectively instantiates all classes that extend Query. In order to obtain the attributes of each opportunity, a developer can rely on the `EClass` class from EMF, which provides methods to access attributes, structural features, references, etc. of classes developed through Ecore. The attributes obtained via `EClass` can then, for each atomic opportunity, be placed onto the attributes table. This is a much more introspective approach to integration that is key to Tychonis' independent maintenance and deployment.

When the user places atomic opportunities onto the canvas and sets attribute values from within the attributes table, SOA can already instantiate Tychonis `Query` classes. Instantiation can occur with the empty instances returned previously by `TychonisUtils.getAllQueries()`, and with the EMF-provided getters and setters for each class. The Tychonis class instantiation process can also trigger invocations of EMF's `Diagnostician` class to check that the user-created opportunities are well-formed, satisfy the constraints of the Tychonis model, and are thus ready to be searched. If this verification fails, users

can be informed via the UI that the opportunity they created is not valid. Information about the failure of this verification can point to detailed specifics such as "*attribute value is not in the desired range*", "*attribute value is missing*", or others.

3.5.3 Search

After the instantiation of `Query` classes, the Science Opportunity Analyzer (SOA) is able to conduct a search for the modeled opportunity. This search process involves the use of `Solver` and `SolverStrategy` classes, as previously discussed. Upon integration with Tychonis, however, SOA's code will not contain references to `Solver` classes. Instead, the logic for selecting the appropriate `Solver` for a given atomic opportunity or Boolean operator is embedded within `SolverStrategy`. The objective is to ensure that any new `Query` or `Solver` added to Tychonis does not require SOA to have knowledge of the logic for choosing a `Solver` for a `Query`. This separation of concerns is necessary since the resolution domain should be confined to Tychonis and not to external tools that employ the framework, such as SOA.

It is important to note that Tychonis provides a default implementation of `SolverStrategy` known as `SimpleSolverStrategy`. As described in section 3.3.3, this class utilizes a dependency injection mechanism to allow users to select the `Solver` class they prefer to use for each `Query` class. In order to implement this, users first modify an external CSV file that contains pairs of the `Query` and `Solver` classes to define a mapping between them. Next, the file is read by code that instantiates the appropriate `Solver` based on the matching `Query` found as `SimpleSolverStrategy` walks the opportunity tree. Essentially, `SimpleSolverStrategy` builds `Query`-to-`Solver` mappings from a file, traverses the opportunity tree in a post-order manner, and invokes the matching `Solver` for each `Query` instance based on the mappings. This traversal returns a `Result` instance that includes all the time windows belonging to the modeled opportunity. This entire process is initiated through a call to the

`solveFrom(Query node)` method of `SimpleSolverStrategy`. A simplified recursive version of this method's implementation is provided in the top section of Listing 5. Lines 9-12 determine the appropriate `Solver` for the current node (line 9), search for the opportunity defined in the subtree (line 10), place the result in the current node (line 11), and return the result (line 12). The `getSolverForNode()` method is responsible for reflectively instantiating the `Solver` class injected in the dynamic `Query`-to-`Solver` dependency file. The result variable contains the results obtained from solving the `Query` tree under the `queryRoot` node. An example invocation of `solveFrom()` is provided in the bottom section of Listing 5, which is the only line of code SOA needs to call to initiate the search for an opportunity defined under the `queryRoot` node.

```
Java
Result solveFrom(Query node)
{
       if(node==null)
              return null;

       solveFrom(node.left);
       solveFrom(node.right);

       Solver solver = getSolverForNode(node); // line 9
       Result result = solver.solve(node);     // line 10
       node.setResult(result);                 // line 11
       return result;                          // line 12

}
```

```
Java
Result result = SimpleSolverStrategy.solveFrom(queryRoot);
```

Listing 5. `SimpleSolverStrategy`'s `solveFrom()` Method. Top: The most relevant parts of `SimpleSolverStrategy`'s implementation of its `solveFrom()` method. Bottom: Invoking `solveFrom()` to search for an opportunity defined under a `queryRoot` node.

The `solveFrom()` method is included in the `SolverStrategy` interface to enable other implementations beyond `SimpleSolverStrategy`. Although users can opt to design their own `SolverStrategy` with their own `solveFrom()` method, `SimpleSolverStrategy`'s versatility in allowing for the injection of dependencies between `Query` and `Solver` classes is expected to be sufficient for most integrators. It is worth noting that future creative implementations of `SolverStrategy` may emerge, such as ones that allocate the execution of `Solver` code across different computer cores for data-independent `Query` instances in the `Query` tree. Additionally, since changes to a `SolverStrategy` implementation are not anticipated to be frequent, no provisions have been made for dynamic integration with multiple `SolverStrategy` classes. Consequently, integration with Tychonis at the resolution level is accomplished by SOA's knowledge of the name of the `SolverStrategy` class to be employed.

3.5.4 Showing Search Results

The Science Opportunity Analyzer (SOA) provides a user interface that displays the results of searching for opportunities in a table, as depicted in Figure 19. This table includes information such as the name of the user-defined opportunity, the time interval number, and the start and end times of each time interval. Additional data can be obtained from the `Result` instance returned by the `SolverStrategy` selected for the query or from the root node of the `Query` tree. `SolverStrategy` saves all partial results in each node of the tree, which means that each node contains the results of its subtree.

Query Name	Window	Begin Time	End Time
ceresPhase	1	2015-292T00:00:00.000	2015-292T01:25:55.387
ceresPhase	2	2015-292T10:47:59.078	2015-292T14:15:09.655
ceresPhase	3	2015-292T23:27:54.245	2015-293T02:51:33.919
ceresPhase	4	2015-293T11:55:23.119	2015-293T15:15:22.536
ceresPhase	5	2015-294T00:10:29.601	2015-294T03:26:57.303
ceresPhase	6	2015-294T12:13:41.789	2015-294T15:26:45.774
ceresPhase	7	2015-295T00:05:35.174	2015-295T03:15:38.819
ceresPhase	8	2015-295T11:46:44.036	2015-295T14:53:38.274
ceresPhase	9	2015-295T23:17:12.608	2015-296T02:21:04.189

Figure 19. SOA's Results Table. Note that from all the data potentially available within a Result instance, SOA only shows the opportunity's time intervals.

As explained in Section 3.3.4, the `Result` class is designed to store any objects of any class that implement the `Resultable` interface, providing flexibility in the data that can be included in the table. A `Solver` can choose what data to include in each cell, which may vary depending on the particular opportunity being modeled. As such, it is crucial for each `Solver`'s documentation to list the items that will be included in the `Result` instance beyond time intervals. The extensibility of `Result` occurs at the cell level, as the data placed within each cell is at the discretion of the `Solver`.

Retrieving the data from a `Result` instance can be done in two ways. One method is to iterate over each cell in the table, parse the results, and populate SOA's UI table. The other method is to call the `getAll(Class<T>aClass)` method, which returns a list of all objects of type `aClass` within the `Result` instance. While the iterative method is agnostic to the classes stored in the table, the `getAll()` method requires the integrator to know what types of objects they wish to retrieve from the table. For instance, to add data to the "*Begin Time*" and "*End Time*" columns with `getAll()`, code similar to that found in Listing 6 can be used.

```
Java
void placeTimeWindowsOnUI(Result result, UITable table) {
      for (TimeWindow window :
      result.getAll(TimeWindow.class)) {
            String start_time =
                  TimeUtils.
                  stringFormat(window.getStartTime());
            table.addRowToColumnID("begin_time",
            start_time);
            String end_time =
                  TimeUtils.
                  stringFormat(window.getEndTime());
            table.addRowToColumnID("end_time", end_time);
      }
}
```

Listing 6. Building SOA's "*Begin Time*" and "*End Time*" columns. Starting from a `Result` instance and a UI component, the code iterates over the TimeWindow instances returned by `getAll()`. It then adds a new row to each column for the start time and end time of each time interval.

4 Textual Language

Usability, which is defined as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" [37], has been a crucial concern for NASA since the early days of space exploration. The practical implications of this concern are clear, as improved usability should lead to better user performance, resulting in increased efficiency and productivity for personnel of all types. Additionally, usability is a critical risk mitigation strategy, as more user-friendly systems reduce the likelihood of operational errors impacting the crew or spacecraft. However, mission planning software for robotic unmanned space missions has only recently started to incorporate more inclusive applications of usability assessment and guiding techniques. This is due to the fact that the idea of multi-mission software, software that can be reused across missions with minimal to no changes, is a relatively recent development for governmental space agencies. For instance, the Human-Computer Interaction software group within JPL was established only in the mid-2010s. Despite being a newly formed group, it has made remarkable contributions to the area of mission planning software by improving software design and architecture, implementing successful design principles within mission processes, and developing interfaces that mission staff can use to interact with mission software. Consequently, JPL has adopted a new approach of engaging usability experts and engineers to implement established industry-wide usability practices into software projects. This signifies the organization's dedication to ensuring the functionality and safety of space mission software. Our commitment is similar, and this section details a research endeavor aimed at integrating usability practices into the design and development of mission planning software, akin to the initiatives undertaken by NASA. Specifically, we assess the usability of two different textual languages to enable engineers and scientists to model opportunities effectively, easily, and satisfactorily.

Although previous sections demonstrate that Tychonis is an extensible and reusable opportunity search framework, its use by individuals who are not experienced software developers poses challenges. If a geometric event needs to be searched, custom code can be written that uses the Tychonis framework, compiled, and executed *à la* mission scripts described in Section 2. The outcome of the execution step would be a time window in which the geometric event of concern takes place. However, the code produced could have an ephemeral existence, as it might only be useful to search for one event. An alternative is to integrate a geometry software library such as Tychonis with a host application that will templatize geometric events. With such an application, the user would be able to trigger the execution of a multitude of event searches through the host software. This pattern was described for one of NASA's software tools, the Science Opportunity Analyzer in Section 3.5. However, space missions frequently encounter themselves in cost-constrained environments that result in limited availability of staff, and this may cascade in science teams not having access to software developers that can write code to search for a specific event. Additionally, it is possible that science teams might not have access to elaborate science planning software such as SOA to search for events in a templatized manner.

Therefore, we propose the idea of a textual computer language that can be used to model and search for geometric events. The goal is to design a language that (i) can provide univocal textual representations of geometric events in space, and (ii) can be used by space mission scientists and engineers with just enough programming experience. In pursuit of this goal, we designed two approaches for a textual language and compared their usability through a survey. This section details our approach to the two language options, the survey's design, the results from the same, and its practical implications.

4.1 Background

The creation of a textual language with the objective of modeling geometric events in the realm of space necessitates an extensive examination of pertinent research in two fundamental domains. The first domain encompasses the relevance and coherence of varied programming language paradigms, while the second concerns the practicality and effectiveness of software systems, which must undergo rigorous evaluation and assessment. This discourse expounds upon these two critical considerations, with the objective of formulating a textual language that is effective, efficient, and capable of accurately modeling geometric events within the complex domain of space.

4.1.1 Programming Paradigm

The declarative programming paradigm promotes the idea that there are types of computer programs that are more apt to be modeled in terms of the description of a problem than in terms of the algorithm needed to solve the problem [38]. The former vision is known as declarative programming and can be exemplified by functional, logic, and constraint languages, but also by Domain-Specific Languages (DSL) such as the Structured Query Language (SQL) [39], which found objective success to model data operations within the realm of Relational DataBase Management Systems (RDBMS) [40]. The latter vision is known as imperative programming and it is promoted by procedural and object-oriented programming general-purpose programming languages. Typical examples of these sub-approaches to imperative programming include storied languages such as Pascal, FORTRAN, C/C++, Java, and Python.

The debate between declarative and imperative programming has persisted for several decades, and it is widely accepted that, for scoped and appropriate problems, declarative approaches result in programs that are more comprehensible, learnable, accessible, and communicable. The author in [41] asserts that "declarative programming entails specifying what is to be computed,

but not necessarily how it is to be computed"; the author further explains that programs that benefit from declarative programming are those in which (i) the system performs deductions, that is, there is no need to specify how knowledge will be used by the system, and (ii) knowledge consists of independent facts - control flow is not determined by the facts stated in a program. In contrast, imperative programs are characterized by the need to orchestrate the utilization of functions, variables, classes, etc. within their control flow. From this, and considering Kowalski's equation *algorithm* = *logic* + *control* [42], it can be inferred that the absence of control leaves logic alone, which is declarative in nature, and could still be the minimal expression of a geometric event that does not incorporate its search method. Conversely, [41] explains that the advantages of imperative programming are realized (1) when capturing processes, (2) when expressing second-order knowledge, and (3) in capturing heuristic knowledge.

In the present study, we are not attempting to capture a process, nor second-order or heuristic knowledge, as we believe a geometric event to be a declarative description of an event that exists as a logical statement, with no algorithmic interpretation. By way of proof by contradiction, if we were to accept that an imperative language is more suitable for modeling geometric events in space, we would be developing a language that offers little value over existing languages such as C in conjunction with a library such as SPICE. Learning such a language would impose a significant burden on scientists and engineers working in space missions, similar to the difficulty of them learning both C and SPICE. Additionally, resulting programs could entangle the definition of a geometric event with the algorithmic method used to search for the time window in which the event occurs. In conclusion, reading and writing events in such a language would require specialized training in computational geometry and programming, as users interested in modeling an event would need to be familiar with algorithmic descriptions. Based on the considerations discussed, it has been determined that the design of an imperative language shall not be pursued within the scope of this study.

4.1.2 Programming Language Usability

The concept of usability is defined in various academic works. In [43], usability is described as "the capability in human functional terms to be used easily and effectively by the specified range of users, given specified training and user support, to fulfill the specified range of tasks"'. This definition is consistent with the ISO 9241-11 standard [37], which defines usability as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use". These definitions highlight the significance of not only task completion, but also factors such as effectiveness, ease of use, and user satisfaction in evaluating the usability of a system.

In light of these definitions, the authors of [44, 45] present several methods for evaluating the usability of a software system, including Laboratory Testing, Thinking Aloud, Formal Modeling, Guidelines/Checklists, and Heuristic Evaluation[42]. However, as pointed out in [46], these methods can be challenging to apply or are dependent on the evaluator's expertise. To address these limitations, the authors of [47] proposed the System Usability Scale (SUS), a low-cost, yet high-return method for determining the overall level of usability of a system compared to its competitors or predecessors.

The SUS is a questionnaire composed of ten statements, available in Table 2, with five of them framed in a positive manner and the remaining five in a negative manner. Respondents rate each statement on a Likert scale[43] of 1-5, where 1 is

[42] Laboratory Testing involves testing the software in a controlled environment to assess its functionality and user experience. Thinking Aloud involves observing the user as they use the software and asking them to think aloud as they interact with it. Formal Modeling is a mathematical approach used to model and analyze the user interface of software. Guidelines/Checklists are a set of rules and guidelines that must be followed to ensure that the software meets usability standards. Heuristic Evaluation involves testing the software against a set of heuristics or guidelines for usability to identify potential usability issues.

[43] The Likert scale is a type of rating scale used in surveys or questionnaires to measure respondents' attitudes, opinions or perceptions. The scale usually consists of a statement that expresses a particular view or opinion, followed by a range of response options, such as "strongly agree", "agree", "neutral", "disagree", and "strongly disagree". The Likert scale is designed to capture the intensity and direction of an individual's attitudes or feelings towards a particular statement or topic. It is named after its developer, psychologist Rensis Likert.

Strongly Disagree and 5 is *Strongly Agree*. The SUS method then employs a calculation that produces a number on a 0-100 scale, where higher numbers indicate higher usability. This number is determined by arithmetic operations that penalize high scores on the Likert scale for negative statements and favor high scores for positive statements. [48] found that a large-scale analysis of SUS scores showed that the SUS is a highly robust and versatile tool for usability professionals.

SUS Statements
I think that I would like to use the language above frequently
I found the language above unnecessarily complex
I thought the language above was easy to use
I think that I would need the support of a technical person to be able to use the language above
I found the various functions in the language above were very well integrated
I thought there was too much inconsistency in the language above
I would imagine that most people would learn to use the language above very quickly
I found the language above very cumbersome (awkward) to use
I felt very confident using the language above
I needed to learn a lot of things before I could get going with the language above

Table 2. SUS Statements. These are the different statements respondents to SUS questionnaires are presented in relation to a system they are evaluating. Respondents are asked to provide answers to these statements on a Likert 1-5 scale where 1 is "Strongly Disagree" and 5 is "Strongly Agree".

Given the positive industry sentiment and the advantages of the SUS for our study, where we compare programming language options, it is reasonable to consider the use of the SUS. In addition to its low cost and ease of execution, the SUS also provides unambiguous quantitative measures to compare different language options via the Likert scale responses and the overall 0-100 score.

Furthermore, by asking potential users for their thoughts as they complete the study, the SUS can also serve as a tool for qualitative data collection, similar to the Thinking Aloud method.

4.2 Language Options

In this study, we examine two distinct declarative language options for the purpose of modeling geometric events. One option follows a natural language approach similar to that of SQL, enabling users to define events in a way that resembles the English language. The other option is more structural in nature and uses key-value tuples, which is reminiscent of JavaScript Object Notation (JSON) [49]. The selection of these two options was based on two important factors. Firstly, both SQL and JSON are designed to capture human-readable parameterized statements. While SQL's stricter grammar constrains the user to describe a database query in a certain way, JSON allows for more flexibility when used without a schema [50]. Despite not defining a database query, it is beneficial to adopt a strict grammar in defining an event, as this would lead to a concise and unambiguous representation of the geometric event. Secondly, we sought to adapt to the communication styles of scientists and engineers in their daily work within space missions. While scientists and engineers typically communicate geometric events to each other in a natural language, a SQL-like natural language approach would be similar to the way they typically express themselves to each other. Furthermore, many of these users frequently operate mission planning software, and are therefore familiar with reading files in XML, JSON, or YAML [51] formats, which are based on the same structural principles as JSON. Hence, the JSON paradigm is not unfamiliar to our target user base.

Consideration of the event "Between the start of year 2000 and the end of year 2005, the distance between the Earth and the Moon is less than 400,000 km" entails a comprehensive and logically complete distance range event between two celestial bodies. The event's semantics are explicated through the utilization of

the English language, where its constituents are clearly represented as nouns, and units of measurement are explicitly defined. In this regard, the natural language approach, denoted as Natural-Language-Based (NLB), and the more structural key-value approach, referred to as Key-Value Pair (KVP), are viable options for modeling the aforementioned event. Table 3 captures the modeling of the event through the NLB and KVP approaches, demonstrating their respective advantages and disadvantages. Specifically, the NLB approach renders the event more readable, albeit making parameters such as "*Moon*", "*Earth*", "*400000km*", and "*01/01/2000:12/31/2005*" less visible. In contrast, KVP makes event parameters more apparent, albeit requiring additional effort to comprehend the event as a whole. This complementarity in the strengths and weaknesses of the two approaches provides a strong foundation for a usability study aimed at discerning the more appropriate language for modeling geometric events, based on user feedback.

NLB	KVP

DEF event1 AS
DISTANCE FROM Moon TO
Earth LESS THAN 400000km
DURING 01/01/2000:12/31/2005
	DistanceQuery event1{
	observer: "Moon"
	target: "Earth"
	test: <
	amount: 400000km
	start_time: 01/01/2000
	end_time: 12/31/2005
	}

Table 3. The event "Between the start of year 2000 and the end of year 2005, the distance between the Earth and the Moon is less than 400,000 km.", modeled with the two proposed languages.

4.3 Study Design

Following the Goal-Question-Metric[44] [52] paradigm, let us state that (i) our goal is to determine a usable language approach to model geometric events in space;

[44] The Goal Question Metric (GQM) framework is a structured approach to defining and measuring software quality. It involves setting specific goals, defining questions to assess progress towards those goals, and selecting metrics to provide answers to the questions.

(ii) the question is *what is the relative usability, including readability, of two distinct declarative languages?*; and (iii) the metric is a questionnaire with active exercises, statements with corresponding Agree-Disagree responses on a Likert 1-5 scale and qualitative statements from the respondents. Aspects (i), (ii), and (iii) will be discussed in this section.

4.3.1 Instrument

To assess the relative usability of the two languages, we designed a survey to collect both quantitative and qualitative data. The survey comprises several distinct parts, including a demographics section (Section 1), an exercise where respondents model events in both languages based on given examples (Sections 2 and 3), and a section where respondents rate the readability of each language option on a Likert 1-5 scale (Section 4). The question pertaining to the readability[45] of each language option is particularly relevant to the study as it is believed to be a key factor in determining the success of the language. This is due to the involvement of stakeholders within a mission who may not write events in the language but will be responsible for determining whether an event makes sense within a mission's scientific context. Following the completion of the language evaluation, respondents are asked to fill out the SUS questionnaire for each language option (Sections 5 to 14). The questionnaire utilized in the study is provided in its entirety in Appendix A and [53]. Through the use of this survey, the study aims to gather valuable insights that will inform future decisions regarding the implementation of a language to model geometric events.

4.3.2 Population and Sample

The population of interest for a textual language intended to model geometric events comprises space mission engineers and scientists responsible for planning

[45] Readability in the context of usability refers to the ease with which a user can read and comprehend the content of an interface or document.

the actions that spacecraft will perform to achieve mission objectives. An important consideration in assessing the usability of such a language is the level of computer programming proficiency possessed by users within this population. Based on our experience, individuals who have recently graduated from college tend to have greater ability in using programming languages than those who graduated years ago. This phenomenon is likely due to a more pronounced emphasis on software usage and development in contemporary science and engineering college programs.

Given the trend towards an increasing user base with higher levels of software knowledge, our study focuses on designing a sample that considers the characteristics of users who will be utilizing such a language for the next few decades. This could include recent JPL hires and interns, as well as college students with relevant backgrounds who are not currently affiliated with JPL but may be hired into it. To this end, we selected samples from two distinct sources. The first was JPL, where we identified recent hires and interns with Aerospace Engineering backgrounds, most of whom worked within the Planning and Execution Systems[46] (PES) section of the organization. This team is responsible for staffing and managing large-scale space missions during the operations phase, with a particular focus on the uplink and downlink processes. Notably, the Science Planning group within the PES section utilizes geometry as an input for the planning of spacecraft actions, and thus represents the primary group of users who would benefit from a proposed language to model geometric events. Assuming that the PES section will provide most of the mission operations staff, and given that the Science Planning group currently comprises 11% of the PES section's personnel, we could start with an estimate that roughly 10% of mission operations staff could benefit from the use of a textual language to model geometric events. We anticipate that a similar proportion of mission staff would benefit in missions not managed by JPL or NASA.

[46] The PES Section at JPL is responsible for developing and maintaining software systems for the planning and execution of robotic space missions. These systems are used to create detailed plans for spacecraft operations, including mission timelines and instrument commands, as well as monitor and analyze the health and status of the spacecraft during operations.

The second sample was drawn from students within the Aerospace Engineering program at the Universitat Politècnica de Catalunya[47] (UPC). This sample had similar interests, backgrounds, and training as the JPL sample, but less knowledge of space mission planning. Furthermore, while the individuals in the UPC sample had a very good professional command of the English language, it was not their native language. This diversity within the sample is advantageous, as it is not necessarily the case that users of a proposed textual language should be native English speakers.

Taken together, these two samples represent individuals who possess the requisite experience, educational caliber, and background to be candidates for roles in institutions where they would be modeling geometric events for space missions. A key observation we make between these samples and the current population in an institution such as JPL is that the individuals in our samples possess a more robust background in computer programming than past graduates. As intended, this reflects the trend of relevant university graduates entering the workforce with a greater programming background, and underscores the importance of studying the language usability with individuals who are representative of future space mission staff and will use such a language for years to come.

4.3.3 Execution

To prepare for the execution of our study, a survey was piloted with five JPL interns possessing relevant backgrounds. These individuals were not included in the actual study but served to identify faults in the survey and assess the quality of the survey format. The survey questions were presented via a web-based questionnaire and conducted live through one-on-one video conferencing. One author guided the survey, which took approximately one hour per respondent. The guide explained the study's objectives and research background, sent the web

[47] The Universitat Politècnica de Catalunya (UPC) is a public research university located in Barcelona, Catalonia, Spain. It was founded in 1971 and is well known for its engineering and architecture programs, as well as its research activities in fields such as telecommunications, renewable energy, and aerospace engineering.

link for the survey response interface, observed the respondent's completion of each question, and inquired about any feedback. The objective of this piloting phase was not to collect data but to practice the interview execution and obtain qualitative data about the survey's structure in case modifications were necessary. The following observations were made during the piloting phase:

1. Respondents noted that the definition of the term *use* in Sections 5, 7, 8, 11, and 12 might be ambiguous. Respondents expressed that it could mean (a) writing geometric events with the language, (b) reading geometric events described with the language, or (c) a combination of both writing and reading. It was agreed that future respondents should be informed that the term *use* involves spending 50% of the time writing geometric events using the language and 50% of the time reading geometric events written in the language.
2. Respondents inquired whether they would use an Integrated Development Environment[48] (IDE) to write statements in real life. They mentioned IDEs that could auto-complete JSON-based text based on a JSON schema, but they were unaware if such capability was available for a custom language. Future respondents were informed that both languages would be used with intelligent code completion capabilities that were equivalent in both languages.

Multiple respondents provided these comments during the piloting phase. The survey guide's responsibilities during the study's execution phase were modified to explain these points to respondents as they arrived at relevant sections. This was done to minimize assumptions or bias and normalize mental models.

After completing the piloting phase, we conducted the questionnaire with JPL and UPC samples based on respondent availability. The study followed a format similar to the piloting phase, using a web-based questionnaire completed via

[48] An Integrated Development Environment (IDE) is a software application that provides comprehensive tools for software development. IDEs can help developers in various ways, including code editing, debugging, and testing. They can also provide features such as code completion, syntax highlighting, and refactoring. IDEs can help developers be more productive, efficient, and consistent in their work. They can also help reduce errors and provide a more streamlined development process.

video conferencing. The guide introduced the study's purpose and observed respondents as they answered questions. If respondents had any doubts or questions, the guide provided clarification. If respondents provided any qualitative feedback, the guide recorded it. At the end of the survey, the guide asked if there were any additional qualitative comments about the two language options or the survey. The guide also asked questions specific to answers to questions in the questionnaire, such as why one language was more readable than the other. All comments and responses were transcribed by the guide.

4.3.4 Statistical Analysis

In this study, an integral component of the numerical investigation was an analysis of the responses to questions contained within Sections 4-14. Such responses were deemed statistically significant, as they were structured as a Likert ordinal scale[49]. It was possible to interpret the results of each individual question as a distribution, wherein the proportion of respondents who agreed with a given statement could be represented by a percentage. However, this study focused on the central tendency[50] of the overall sample and the subsamples (JPL and UPC) for each question and language option, when analyzing the Likert-based results. Notably, Likert data is inherently ordinal, thereby restricting the use of parametric analyses such as the statistical mean [54]. Consequently, the statistical mode was used per question, per language option, for the overall sample and for the subsamples.

In addition, the SUS scores were utilized as another key aspect of our statistical analysis. These scores were presented as a numeric value between 0 and 100 for each participant in the study. To determine usability, arithmetic means were calculated for the SUS scores of the overall sample and the subsamples. As the

[49] An ordinal scale is a measurement scale that orders objects or events based on their relative position. It does not provide an equal interval between values, and it does not measure the distance between them. Instead, it identifies the position of an object or event in relation to others in the same group.

[50] The tendency of data to cluster around a central value or typical value. This is often measured using statistical measures such as mean, median, and mode, which can provide insight into the distribution of data and help summarize large amounts of data into a single value.

SUS is an interval score[51] for usability, arithmetic means were considered appropriate in this context.

4.3.5 Threats to Validity

One area of the study in which we believe there was a potential internal threat to validity is regarding the progression and mental framing of the respondents as they complete the questionnaire. More specifically, this threat involves a conscious or subconscious preference for one language versus another as a result of the structure of the questions being asked. At the SUS level, this is handled by alternating positive-framed questions, i.e., questions where *Strongly Agree* is positive for usability, with negative-framed questions, i.e., questions where *Strongly Agree* is negative for usability. As a result, no action was taken from the point of view of our study design in relation to SUS question ordering. An action was taken, however, regarding the fact that our study considers the same sequential questions for the two language options. In order to prevent a possible situation in which a respondent might think a language option is better because it was presented first or second, we chose to alternate the language options within each question, e.g., the questionnaire will ask Question 1 for language NLB first, and KVP second, and then the questionnaire will ask Question 2 for language KVP first, and language NLB second, and so on. Note that in the questionnaire we did not use the names NLB and KVP for the languages, but we did name languages as *Language A* and *Language B* within each section, and alternated what Language A and B represented as language options. In effect, *Language A* in Section *n* would be different from what *Language A* was in Section *n+1*. This was done to avoid name affinity or likability, and recency bias.

A criticism could be made in relation to the fact that we did not distribute language options across the sample, that is, we did not randomly assign language options across the sample in a way that one respondent responds to questions

[51] An interval scale is a measurement scale where the distance between two points is meaningful and consistent throughout the scale. It allows for meaningful comparisons of the differences between the values on the scale.

about only one language option, unaware there is another option. We believe this to be a sensible comment on validity, however, we also believe there is value in respondents being exposed to the two options, as they not only can factor in a scoring in their answers that is relative to each language option against the other, but can also provide qualitative comments about the two options in relation to each other.

In terms of the sample used, there were threats about target population representation for the languages we are evaluating. We initially believed that using the JPL sample was enough to depict the kinds of individuals involved in modeling geometric events in space. However, as the study design evolved, our ideas also evolved. We thought that while the JPL sample was well-versed in the uses our languages would need to operate in, JPL as a whole is a microcosm of a larger realm that can involve users from different organizations and nations outside of the United States. To that end, we recruited individuals from UPC in Catalonia, Spain with a relevant background who could also be users of the languages in their professional or research lives. This action increased sample diversity, making a more robust sample, and implied the benefit that qualitative comments would be expected to be richer as a whole. One point of view is that this could create a threat in case the two subsamples provide quantitative results that substantially differ from the other subsample, but if that occurred, that would be an input to iterate on more research to design a better language proposal.

4.4 Study Design

The subsequent paragraphs present an overview of the findings from each survey section. These include an examination of participant demographics in Section 1, followed by an evaluation of the exercises in Sections 2 and 3, an analysis of readability in Section 4, and a detailed exploration of the SUS questions in Sections 5-14.

4.4.1 Section 1 - Demographics

The survey questionnaire administered to the participants comprised several demographic questions. These questions requested respondents to furnish their personal details, including their name, college major for their highest degree, educational level, time since graduation, and years of experience in the aerospace domain. For the question on college major, results indicated that a majority of the respondents, specifically 77.78%, graduated or were in the process of graduating from an Aerospace Engineering program. Further, 14.81% and 7.41% of the participants held a major or had majored in Computer Science and Applied Mathematics, respectively. Concerning the educational level, 70.37% of the respondents were graduating or en route to graduating from a Bachelor's-level program, while 29.63% had completed or were to complete a Master's-level program. Furthermore, the analysis revealed that 66.67% of the participants were yet to graduate, and the average time since graduation for the remaining 33.33% was 1.67 years.

The experience level in the aerospace domain was ascertained by providing clarity to respondents that experience refers to any form of involvement with the aerospace domain in general. Hence, internships, work experience, and aerospace-related education were considered valid criteria. The analysis indicated that the mean level of experience in the aerospace domain was 3.61 years, considering both samples. The assessment of software development background was designed to measure experience as a binary and more subjective value for each respondent. Specifically, respondents were asked to evaluate if they considered themselves software developers or not. This approach was adopted due to the diverse nature of the sample population, including respondents from varying majors and interests. Measuring experience in terms of length of time could have provided a wide range of values, as programming could be a part of a career, major, interest, or hobby. In addition, measuring programming knowledge more comprehensively would have necessitated designing additional questions that would have been less relevant to the study's objectives. The results showed

that 92.59% of the participants identified themselves as software developers. Further details and comprehensive data can be accessed from Table 4.

	Total	JPL Sample	UPC Sample
Sample Size	27	18	9
Time since graduation, arithmetic mean (years)	0.63	0.78	0.33
Time since graduation, range (years)	[0-5]	[0-5]	[0-1]
Experience in the aerospace domain, arithmetic mean (years)	3.61	3.33	4.17
Experience in the aerospace domain, range (years)	[0-9]	[0-9]	[3-5]
Think of themselves as software developers (sample %)	92.59%	100.00%	77.78%

Table 4. Demographics of the total sample and the two subsamples.

4.4.2 Sections 2 and 3 - Exercises

The purpose of Sections 2 and 3 was to enhance respondents' comprehension of each language option and cultivate critical perspectives on their usage. In Section 2, an example event was presented, namely "Between the start of year 2000 and the end of year 2005, the distance between the Earth and the Moon is less than 400,000 km.", which was modeled using the two distinct language options. Respondents were then requested to model another event related to the example, namely "In all of 2021, Earth and Mars are at a distance of more than 70M km from each other". All participants provided an objectively correct answer for each language option. Exemplary responses are available in Table 5.

NLB	KVP

DEF event_name AS
DISTANCE FROM Earth TO
Mars MORE THAN 70Mkm
DURING 01/01/2021:12/31/2021
SEARCH FOR event2
	DistanceQuery event2{
	observer: "Earth"
	target: "Mars"
	test: >
	amount: 70Mkm
	start_time: 01/01/2021
	end_time: 12/31/2021
	}
	SEARCH FOR event2

Table 5. Potential solutions for exercise in Section 2. Respondents were asked to model and search for the event "In all of 2021, Earth and Mars are at a distance of more than 70M km from each other".

On the other hand, Section 3 showcased language grammar that could serve as documentation to be read by users of each language, to model an occultation event in each language option. To clarify, an occultation occurs when the line of sight between two celestial bodies is disrupted by a third body. For instance, a solar eclipse is a form of occultation, as the line of sight between Earth and the Sun is disrupted by the Moon, either fully or partially. The documentation was followed by the request to model the event "Earth and Mars are at a distance of more than 70M km, and at the same time, Mars and Earth are in full solar conjunction. Use the 2010-2020 interval". Respondents were required to utilize the documentation in Section 3 and examine their previous answers in Section 2, which was possible through the web-based form. Similar to Section 2, none of the participants failed to provide the correct answer for the exercise within Section 3. Exemplary responses for the question in Section 3 are available in Table 6.

NLB	KVP

DEF event1 AS
FULL OCCULTATION
BETWEEN BACK
BODY Mars
AND FRONT BODY Sun
OBSERVED BY BODY
Earth
DURING
01/01/2010:12/31/2020
 	
DEF event2 AS
DISTANCE FROM Moon TO
Earth MORE THAN 70Mkm
DURING 01/01/2010:12/31/2020
 	
SEARCH FOR event1 AND event2
	DistanceQuery event1{
	observer: "Earth"
	target: "Mars"
	test: >
	amount: 70Mkm
	start_time: 01/01/2010
	end_time: 12/31/2020
	}
	OccultationEvent event2{
	type: "full"
	back_body: "Mars"
	front_body: "Sun"
	observer: "Earth"
	start_time: 01/01/2010
	end_time: 12/31/2020
	}
	SEARCH FOR event1 AND event2

Table 6. Potential solutions for exercise in Section 3. Respondents were asked to model and search for the event "Earth and Mars are at a distance of more than 70M km, and at the same time, Mars and Earth are in full solar conjunction (solar conjunction is when the Sun is between two bodies). Use the 2010-2020 interval".

4.4.3 Section 4 - Language Readability

In Section 4, respondents were presented with a statement, "I think the language above is readable by a scientist or engineer", and provided with sample events in each language option. A Likert 1-5 scale was utilized to record respondent perceptions, where 1 indicated *Strongly Disagree* and 5 indicated *Strongly Agree*. The NLB results indicated a mode of *Strongly Agree* for both subsamples and the overall sample. In contrast, the KVP results displayed a mode of *Neutral* for the overall sample and the JPL subsample, while the UPC subsample demonstrated a tie between *Agree* and *Neutral*. The mode outcomes for this question are presented in Table 8, whereas the complete results can be found in [27] and Appendix B.

4.4.4 Sections 5-14 - System Usability Scale (SUS)

The average SUS score across all respondents for NLB was found to be 89.81. The JPL sample had an average SUS score of 90.00 for NLB, while the UPC sample had an average SUS score of 89.44. For KVP, the average SUS score across all respondents was 90.00, with the JPL sample having an average SUS score of 89.72, and the UPC sample having an average SUS score of 90.55. It was observed that the respondents favored KVP's usability by a very small margin of 0.19 points compared to NLB. Interestingly, the JPL sample gave a higher average score to NLB, albeit by a small margin of 0.28 points. In contrast, the UPC sample gave a higher score to KVP by a larger, but still small margin of 1.11 points. These results can be found in Table 7.

Sample	NLB, Mean SUS Score	KVP, Mean SUS Score	Difference Magnitude for Mean SUS Scores
Total	89.81	**90.00**	0.19
JPL	**90.00**	89.72	0.28
UPC	89.44	**90.55**	1.11

Table 7. SUS Scores per sample as arithmetic means and the difference magnitude between means. High scores in each row are marked in bold lettering.

When examining the response modes for the SUS questions, it was observed that both subsamples provided comparable answers. In fact, out of the 20 total combination questions for two language options, the JPL and UPC samples only differed in their response modes on two occasions. To verify these discrepancies through inferential statistics, a Mann-Whitney U test was conducted on the responses from both subsamples. The null hypothesis (H0) was that the two subsamples were drawn from the same population, thus producing comparable results. On both identified discrepancies, the null hypothesis was rejected at a significance level (α) of 0.05. No other descriptive (via mode analysis) or

inferential (via Mann-Whitney U test[52]) discrepancies were detected. The two discrepancies are outlined below:

- For NLB and question #1, "I think that I would like to use the language above frequently", the JPL sample's mode was *Strongly Agree*, whereas the UPC sample's mode was *Agree*. The p-value of the Mann-Whitney U test was 0.040.
- For NLB and question #8, "I found the language above very cumbersome (awkward) to use", the JPL sample's mode was *Strongly Disagree*, while the UPC sample's mode was *Disagree*. The p-value of the Mann-Whitney U test was 0.035.

The response mode data for the SUS questionnaire can be found in Table 8, while full results can be accessed in Appendix B and [55].

Statement	NLB Responses			KVP Responses		
	Total Mode	JPL Mode	UPC Mode	Total Mode	JPL Mode	UPC Mode
Section 5: I think the language above is readable by a scientist or engineer	Strongly Agree	Strongly Agree	Strongly Agree	Neutral	Neutral	Agree/ Neutral
Section 6 (SUS #1): I think that I would like to use the language above frequently	**Agree**	**Strongly Agree**	**Agree**	Strongly Agree	Strongly Agree	Strongly Agree
Section 7 (SUS #2): I found the language above unnecessarily complex	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree
Section 8 (SUS #3): I thought the language above was easy to use	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree
Section 9 (SUS #4): I think that I would need the support of a technical person to be able to use the language above	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree

[52] The Mann-Whitney U test is a nonparametric statistical test that compares two independent groups to assess whether they are drawn from the same population or not. This test is appropriate for ordinal or continuous data that do not meet the assumptions of parametric tests.

Statement	NLB Responses			KVP Responses		
	Total Mode	JPL Mode	UPC Mode	Total Mode	JPL Mode	UPC Mode
Section 10 (SUS #5): I found the various functions in the language above were very well integrated	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree
Section 11 (SUS #6): I thought there was too much inconsistency in the language above	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree
Section 12 (SUS #7): I would imagine that most people would learn to use the language above very quickly	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree
Section 13 (SUS #8): I found the language above very cumbersome (awkward) to use	**Strongly Disagree**	**Strongly Disagree**	**Disagree**	Strongly Disagree	Strongly Disagree	Strongly Disagree
Section 14 (SUS #9): I felt very confident using the language above	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree	Strongly Agree
Section 15 (SUS #10): I needed to learn a lot of things before I could get going with the language above	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree	Strongly Disagree

Table 8. Mode for each response in Sections 5-14. The *Statement* column in sections 5-14 includes the statement provided to respondents and the SUS question number (except for Section 5). The *Responses* columns show the mode for each sample (JPL and UPC) and the combined mode in the *Total Mode* column. Discrepancies between samples are highlighted in bold.

5 Discussion

In this section, we analyze various topics related to utilizing Tychonis in practical situations. We examine Tychonis' design characteristics and their applicability to real-world scenarios, the separation of modeling and resolution concerns, strategies to balance software design purity with real-world applicability and growth, the performance of Tychonis's `Solver` and `SolverStrategy` classes, integration of Tychonis with mission software, and the impact of mission resources and risk postures on the adoption of novel technologies like Tychonis. Lastly, we discuss our study's outcomes on the two textual languages and their implications for space missions.

5.1 Metamodel Software Design vs. Applicability

The previous sections of this document presented how Tychonis' design characteristics enable extension and reusability in a way that advances the present state of the art in the field of opportunity search software. Nonetheless, this achievement is not immune to examination by both its creators and potential users.

Future critical assessments of the framework's design may raise concerns about how it handles the separation of modeling and resolution concerns, as well as the abstraction it promotes. An illustrative example is the scenario where a user searches for an opportunity using a SPICE-based `Solver`, which would require loading into SPICE kernel files that contain body definitions, ephemeris data, and other relevant information. In this case, a Tychonis `Query` validation call might need to verify whether a specific body name (e.g., "*EARTH*") referenced in the `Query` instance actually exists in the kernels, which would mean that the `Query` object must be able to communicate with SPICE. However, this requirement

poses a challenge to maintaining a clean separation between the modeling and resolution steps for an opportunity. To address this, one feasible solution is to defer the validation of some `Query` components until `Solver` code executes, which in our example means that it is the `Solver` implementation that checks the existence of the body name in the kernels loaded into SPICE. This is the strategy adopted by Tychonis, as it preserves the separation between modeling and resolution steps, albeit at the cost of conducting a less thorough validation at the modeling stage. Nonetheless, this approach can balance the need for maintaining design purity with the practicality of building a usable and extensible framework that meets the diverse needs of its potential users.

Another example, but this one on the abstraction front, is that `Result` instances contain result tables that have been designed to be generic in that they can contain any element as long as it implements the `Resultable` interface. This is powerful in order to make `Solver` classes decide autonomously what data goes into a `Result`. However, users and other `Solver` classes might need to have a way to obtain more concrete result data beyond text-based output (i.e., `stringValue()` in `Resultable`). In this case, given the design need for abstraction needs to be relaxed, Tychonis provides `Query` capabilities to retrieve objects of specific class types from a `Result` instance as described in the previous section.

These exemplifying challenges and their solutions prove that a good balance between software design purity, real-life applicability, and capability growth can resolve current and future needs as long as judiciousness is maintained over time[53].

[53] This is embodied by the maxim: *The architecture of a software system tends to degrade over time, unless an effort is made to keep it clean.*

5.2 Solver Performance

Tychonis's default implementations of `Solver` and `SolverStrategy` classes implement algorithms that could be sped up with their parallelized analogs. For `Solver` classes, an idea is to parallelize the search of time windows by dispatching packaged sub-searches to different threads, then analyzing and consolidating time windows found in each of the threads. This is also applicable in `Query` instances in which parts of the input are unknown. For instance, if there is a distance `Query` object in which the target body is an unknown, one sub-search could search for target body b1, another sub-search could search for b2, and so on. One thing to note in this pattern is that given the flexibility afforded by Tychonis in the development of `Solver` classes, one could implement this parallelism within a `Solver` as operating system threads, as suggested previously, but also as computer cluster jobs, MapReduce [56] functions, or other avenues for implementation of the fork-join[54] [57] pattern.

At the composite opportunity level, `SolverStrategy` classes can also implement parallelism when they resolve the opportunity tree. One common occurrence is to have two non-Boolean `Query` objects connected by an AND or an OR. In this example, each one of the non-Boolean `Query` objects can be resolved independently in parallel. Once both those `Query` objects are resolved, then the Boolean operation can be executed. This mode of operation can be generalized to trees of any size. Also, in this depiction, `SolverStrategy` directs the resolution in a fork-join manner similar to that of parallelized `Solver` classes; with the implication that this algorithm can also be implemented through various parallel computing paradigms. Current SPICE versions, however, do not support function-level parallelism. If there is a need to implement parallelism within a single multi-core computer, it might follow that

[54] The fork-join pattern is a programming paradigm commonly used in parallel computing, in which a task is recursively split into smaller subtasks, which are executed concurrently. Once all subtasks are completed, their results are merged or joined to produce the final output. This pattern is particularly useful for optimizing the performance of applications that can be parallelized, such as those with many independent and identical tasks. The fork-join pattern is often implemented using libraries or frameworks that provide support for multithreading or distributed computing.

parallelism of SPICE at the application thread level is not possible. However, there are ways to mitigate the latter concern. The Modeling and Verification Group at JPL is currently building a Java library called ParSPICE that can create an arbitrary number of non-blocking SPICE engines within the same Java Virtual Machine (JVM). The library provides access to the typical SPICE calls, which are then dispatched in a non-blocking manner to the created SPICE sub-engines, thereby enabling SPICE parallelization. This library can be useful should a mission decide to implement new `Solver` code with SPICE.

5.3 Release and Adoption

We need to underscore the strategic relevance of integrating Tychonis with mission-specific software in the proposal, development, or operations phase. It is important to consider that the needs and resources of a mission in these three phases can significantly vary. Proposals, for instance, typically have limited resources and would benefit from adopting reusable software to limit development time. Missions in the development phase, on the other hand, have more resources but become increasingly risk-averse to new developments as launch date approaches. Finally, missions in operations typically have sustained but diminishing funding and may become more open to adopting new technologies, particularly after the prime mission objectives have been achieved. Such risk and resource profiles, coupled with a mission's ongoing need to search for geometric events, provide engineers with opportunities to adopt a framework like Tychonis. Given its reliance on good design practices and its ability to address user needs, incorporating Tychonis can be less expensive and less risky than adopting ad-hoc approaches.

Tactically, we propose the integration of Tychonis with the opportunity search capability of SOA as a first step. This step is particularly fitting given that the management of SOA's development was under the purview of one of the authors' teams at JPL. The NASA-managed Psyche mission, currently in the development phase for preliminary science planning, is also using SOA and will

continue to use the software during mission operations. Likewise, the Europa Clipper mission will use SOA to schedule the activities of some of its instruments during mission operations. We expect that integrating SOA with Tychonis will lead to cost savings, as opportunity definitions and their search algorithms could be developed once by either mission and shared by both. A successful SOA/Tychonis integration can also pave the way for Tychonis' integration with JPL software other than SOA.

Beyond JPL, we intend to release Tychonis as an open-source codebase, thereby increasing the software's availability and potentially facilitating its adoption by research centers and universities. Such adoption can lead to integrations with other third party tools and augmentations to Tychonis' open-source repository. It is worth noting that the potential integrations and augmentations of Tychonis' open-source repository are not limited to applications relating to geometric events for science planning. As an example, the tool described in [58] presents a proof-of-concept software that determines the availability and data rate at which different space data nodes can communicate with each other. This software, in a way, models parts of a protocol stack similar to TCP/IP[55] where these space data nodes act as routers and are transmitting information with each other. For this tool and others similar to this one, Tychonis could be extended to model and search for multi-hop routes that account for orbital mechanics, optimize network capacity, line of sight between routers, and even build a full routing table[56]. These new events and their search methods could make it into an open source repository available for anyone to use and be implemented in those space routers or ground planning systems.

[55] TCP/IP stands for Transmission Control Protocol/Internet Protocol. It is a set of communication protocols used to interconnect network devices on the internet. TCP/IP provides end-to-end connectivity by specifying how data should be packetized, addressed, transmitted, routed, and received by network devices. The protocol has been foundational in the development of the internet, and it continues to be widely used today.

[56] A routing table is a data table stored in a network device, such as a router or switch, that contains information used to determine the best path for forwarding data packets between different network segments. The routing table includes entries for destination addresses as well as information about the next device or hop to which the packet should be sent. This information is used by the device to make decisions about where to send packets of data based on the routing protocols it has learned and the network topology it is connected to.

Another advantage of Tychonis as an open source tool is that it has the potential to be used as a practical teaching tool in science and engineering courses on various topics such as planetary dynamics, space communications, and remote sensing. This means that students could use Tychonis to learn about space geometry without concerning themselves overly with specific algorithms, which can be a barrier for some students. In software-oriented courses, exercises could be proposed on the development of opportunities and Solvers using Tychonis, which could serve as a case study on how good software practices can be beneficial in the development of larger software applications. In short, the authors see the potential for Tychonis to be used as a valuable educational tool in various fields related to space science and engineering.

5.4 Textual Language

One of the main takeaways from the study presented in Section 4 is that respondents considered that NLB was, by a substantial margin (see Table 8, Section 5), a more readable implementation of a textual language for our purpose. An expectation, while designing the two language options, was that NLB would be considered more readable overall due to its similarity to the English language, but there was also doubt as to whether, given the software development experience of the samples and their potential more comprehensive exposure to data exchange formats like JSON or XML, that KVP could also be considered quite readable. One thought we are currently considering is the fact that the positive evaluation of NLB in terms of readability also makes it more communicable within a group, and only not at the level of just one individual. In other words, scientists and engineers, with no previous training, can look at events written in it and understand their meaning and debate whether an event makes sense or not within the context of a mission. This debate is a key point, as we have learnt that scientists and engineers, especially during the remote work spells of the COVID-19 pandemic, exchange textual information via email and chat facilities more frequently than when they are physically co-located. While

this is not a core point to our research, it makes an argument that the ability to understand or read an event might be more important than the ability to write an event. In essence, more readability might imply higher-quality group communication and decision-making via digital-textual means.

Another reflection is that the differences between NLB and KVP in relation to the SUS scores results are small and the SUS scores are overall very high, indicating that both options are highly usable according to the sample and the methods used. Delving into the details of the scores, the two samples combined gave a higher score to KVP, by a very small margin; however, the JPL sample gave a higher score to NLB, and the UPC sample gave a higher score to KVP. Because both samples are small and the differences in scores are small, this could be construed as part of survey *noise*, but we believe there is something indicated by this small divergence. Our thesis is that native English speakers favor NLB, and the totality of the JPL sample is indeed composed of native English speakers. In contrast, the UPC sample was composed, in its totality, by non-native English speakers, and while they all had very good command of the English language and never asked for clarification about the grammar or vocabulary that was part of the NLB option, a slight hesitation towards a language they are not fully comfortable with, when compared to their native languages, might be encoded in their responses. As we saw for the statements "I think that I would like to use the language above frequently" (Table 8, Section 6) and "I found the language above very cumbersome (awkward) to use" (Table 8, Section 13), the mode results indicate the UPC sample was slightly less enthusiastic about NLB than it was for KVP relative to the JPL sample. While we are not considering developing an NLB option in different languages, one way to validate this thesis would be to perform the same study in which both KVP and NLB use words and constructions in the native language of the respondents. In any case, and as a corollary for the SUS scores, we (1) do not consider usability for one language option to be substantially better than for the other, and (2) both language options are highly usable as evidenced by the high SUS numbers.

Given the interpretation of the data, we have concluded it is more sensible to proceed with research on the implementation of a textual language that adheres to the NLB philosophy. This research would include the completion of the grammar for a variety of frequently used geometric events, the design of a unit system for values such as distance, and a mature way to define dates and times. In order to implement the language, one route to take is to leverage the Xtext language development framework [59], as it provides a path to transform text into instances of a metamodel, which aligns with the design of the Tychonis framework. Another benefit of Xtext is that, depending on the environment in which it is used, it provides auto-complete and other guard rails for languages implemented with it. Notably, the need for such capability was a comment provided by respondents during our dry run of the survey.

Additional future research should also consider the automatic expansion of the language grammar based on independent augmentations of the Tychonis framework. This would imply that if a developer adds a new type of searchable geometric event to Tychonis, the textual language would automatically be augmented without having to involve a language developer to explicitly augment the grammar, as the grammar would be implicitly encoded in the Tychonis metamodel. As already covered, one of the main tenets of the Tychonis framework is separation of concerns. These concerns comprise (1) the description, or modeling, of families of geometric events, (2) the description of the algorithms to search for a family of geometric events, and (3) the capture and provisioning of the geometric event search results. In regards to a textual language, a fourth concern should consider how the parameters of a family of geometric events would manifest within the context of a given textual language such as the ones described in this manuscript.

6 Conclusion

The current state of the art in opportunity search software has been limited by the inadequacies of existing solutions. Both mission-developed scripts and multi-mission frameworks have been found wanting in terms of essential qualities such as extensibility, reusability, and a clear separation between the definition of events and their search. To overcome these limitations, we embarked on the development of a new framework named Tychonis. Tychonis is designed to be a solution that advances the state of the art in opportunity search software by integrating with both current and future mission software. Its core capabilities include enabling end-users to extend opportunities and search algorithms without modifying the tools that use the framework, and promoting the cross-mission reusability of the framework and developments implemented in it.

Our design goals for Tychonis were consciously guided by best practices in software design. The first goal was separation of concerns, which provides distinct constructs for the modeling of opportunities, search of opportunities, and the capturing of search results. This improves code readability and better maps to user needs, while enabling additional design goals. The second goal was user extensibility, which allows users to extend the constructs that model each concern. The third goal was mission reusability, which makes Tychonis an independent framework that is agnostic as to what tools it can integrate with and provides a complete set of capabilities in its default form. The fourth and final goal was verification and validation, which allows for the validation of opportunities modeled with Tychonis without the need to develop additional code.

To demonstrate the practical application of Tychonis, we presented a case study of its integration with SOA. Our intent was to highlight the value added by the design goals through more technical and tangible descriptions. The integration pattern described in this paper can serve as a useful reference for parties

interested in integrating Tychonis with their own tools. It also proves that the idea of Tychonis as a reusable and extensible framework that can evolve separately from the tools that integrate with it is doable and potentially a pattern to follow by other frameworks is a similar environment.

As a complement to Tychonis, we conducted an inquiry to examine two distinctive modalities for modeling geometric events through text, which obviate the requirement for extensive knowledge of programming languages. These modalities comprise a natural-language-based language and a key-value-based language. Our findings indicate that while both modalities are highly usable, the natural-language-based language appears to be more legible. A noteworthy observation is that it would be sensible to integrate the natural-language-based language with Tychonis, a task that could be accomplished via an automated mechanism that synchronously updates the language with any modifications made to Tychonis, thus ensuring its up-to-date status.

The Tychonis framework and the languages we designed represent a significant step forward in the field of opportunity search software. By addressing the limitations of existing solutions and meeting a set of well-defined design goals, our ideas have the potential to greatly enhance the capabilities of future mission software and promote reusability in a way that was not possible before in this domain. At this juncture, it brings us immense happiness to realize that our efforts will aid in the advancement of state-of-the-art space mission software, even beyond the opportunity search domain.

7 References

[1] R. Shishko and R. Aster, "NASA systems engineering handbook," NASA Special Publication, vol. 6105, 1995.

[2] M. Llopis, C. A. Polanskey, C. R. Lawler, and C. Ortega, "The Planning Software Behind The Bright Spots On Ceres: The Challenges And Successes Of Science Opportunity Analyzer," presented at the 2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT), IEEE, 2019, pp. 1–8.

[3] I. Deliz, A. Connell, C. Joswig, J. J. Marquez, and B. Kanefsky, "COCPIT: Collaborative Activity Planning Software for Mars Perseverance Rover," presented at the 2022 IEEE Aerospace Conference (AERO), IEEE, 2022, pp. 01–13.

[4] S. Chien et al., "ASPEN–Automated Planning and Scheduling for Space Mission Operations".

[5] A. Yelamanchili et al., "Automated science scheduling for the ECOSTRESS mission," 2019.

[6] P. F. Maldague, S. S. Wissler, M. D. Lenda, and D. F. Finnerty, "APGEN scheduling: 15 years of Experience in Planning Automation," presented at the SpaceOps 2014 Conference, 2014, p. 1809.

[7] C. R. Lawler, F. L. Ridenhour, S. A. Khan, N. M. Rossomando, and A. Rothstein-Dowden, "Blackbird: Object-Oriented Planning, Simulation, and Sequencing Framework Used by Multiple Missions," presented at the 2020 IEEE Aerospace Conference, IEEE, 2020, pp. 1–20.

[8] G. Rabideau et al., "Onboard automated scheduling for the Mars 2020 rover," 2020.

[9] J. Riedel et al., "AutoNav Mark3: Engineering the next generation of autonomous onboard navigation and guidance," presented at the AIAA Guidance, Navigation, and Control Conference and Exhibit, 2006, p. 6708.

[10] B. Streiffert and T. O'Reilly, "The evolution of seqgen-a spacecraft sequence simulator," presented at the SpaceOps 2008 Conference, 2008, p. 3523.

[11] P. Pashai, M. Hurst, E. Fosse, S. Myint, and N. Rossocheva, "Test as you Fly: Using Flight Telemetry in the Mars 2020 Uplink Simulation & Validation Process," presented at the 2022 IEEE Aerospace Conference (AERO), IEEE, 2022, pp. 1–9.

[12] M. D. Johnston et al., "Automated scheduling for NASA's deep space network," AI Magazine, vol. 35, no. 4, pp. 7–25, 2014.

[13] B. A. Streiftert, C. A. Polanskey, T. O'Reilly, and J. Colwell, "Science opportunity analyzer - a multi-mission approach to science planning," in 2003 IEEE Aerospace Conference Proceedings (Cat. No.03TH8652), 2003, p. 8_3615-8_3627.

[14] L.-K. Chan and M.-L. Wu, "A systematic approach to quality function deployment with a full illustrative example," Omega, vol. 33, no. 2, pp. 119–139, 2005.

[15] C. A. Polanskey, B. Streiffert, T. O'Reilly, and J. Colwell, "Advances in science planning tools with the Science Opportunity Analyzer," 2002.

[16] E. Vinterhav and T. Karlsson, "Script based software for ground station and mission support operations for the Swedish small satellite Odin," Acta Astronautica, vol. 61, no. 10, pp. 912–922, 2007.

[17] C. H. Acton Jr, "Ancillary data services of NASA's navigation and ancillary information facility," Planetary and Space Science, vol. 44, no. 1, pp. 65–70, 1996.

[18] J. Smith et al., "MONTE Python for deep space navigation," 2016.

[19] B. De Win, F. Piessens, W. Joosen, and T. Verhanneman, "On the importance of the separation-of-concerns principle in secure software engineering," presented at the Workshop on the Application of Engineering Principles to System Security Design, Citeseer, 2002, pp. 1–10.

[20] C. Acton, N. Bachman, B. Semenov, and E. Wright, "SPICE tools supporting planetary remote sensing," 2016.

[21] B. Semenov, "WebGeocalc and cosmographia: Modern tools to access OPS SPICE data," presented at the 2018 SpaceOps Conference, 2018, p. 2366.

[22] S. Stallcup, "Solar system geometry and ephemeris processing for the HST," presented at the Observatory Operations to Optimize Scientific Return, SPIE, 1998, pp. 402–409.

[23] M. Almeida, "Solar system operations lab for constructing optimized science observations," in SpaceOps 2012, 2012.

[24] P. Van Der Plas, "MAPPS: A science planning tool supporting the ESA solar system missions," presented at the 14th International Conference on Space Operations, 2016, p. 2512.

[25] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, EMF: Eclipse Modeling Framework. Pearson Education, 2008.

[26] W. Hürsch and C. Lopes, "Separation of Concerns. Northeastern University," TR NU-CCS-95-03, USA, 1995.

[27] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. E. Lorensen, Object-oriented modeling and design, vol. 199, no. 1. Prentice-hall Englewood Cliffs, NJ, 1991.

[28] G. Booch, I. Jacobson, and J. Rumbaugh, "The Unified Modeling Language," Unix Review, vol. 14, no. 13, p. 5, 1996.

[29] M. Voelter et al., "DSL engineering-designing, implementing and using domain-specific languages," 2013.

[30] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (XML)," World Wide Web Journal, vol. 2, no. 4, pp. 27–66, 1997.

[31] R. F. Paige, D. S. Kolovos, and F. A. Polack, "A tutorial on metamodelling for grammar researchers," Science of Computer Programming, vol. 96, pp. 396–416, 2014.

[32] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: A controlled experiment," presented at the Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 763–774.

[33] M. Fowler, "Inversion of control containers and dependency injection pattern," http://www. martinfowler. com/articles/injection. html, 2006.

[34] R. C. Martin, Agile software development: principles, patterns, and practices. Prentice Hall PTR, 2003.

[35] P. Maes, "Concepts and experiments in computational reflection," ACM Sigplan Notices, vol. 22, no. 12, pp. 147–155, 1987.

[36] D. E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Third. Reading, Mass.: Addison-Wesley, 1997.

[37] N. Bevan, "ISO 9241: Ergonomic requirements for office work with visual display terminals (VDTs)-Part 11: Guidance on usability," Tc, vol. 159, p. 61, 1998.

[38] J. W. Lloyd, "Practical advantages of declarative programming," presented at the Joint Conference on Declarative Programming, 1994, pp. 3–17.

[39] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A structured English query language," presented at the Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control, 1974, pp. 249–264.

[40] E. F. Codd, "Relational Database: A Practical Foundation for Productivity," in ACM Turing Award Lectures, New York, NY, USA: Association for Computing Machinery, 2007, p. 1981.

[41] T. Winograd, "Frame representations and the declarative/procedural controversy," in Representation and understanding, Elsevier, 1975, pp. 185–210.

[42] R. Kowalski, "Algorithm = logic + control," Communications of the ACM, vol. 22, no. 7, pp. 424–436, 1979.

[43] B. Shackel, "Usability–Context, framework, definition, design and evaluation," Interacting with computers, vol. 21, no. 5–6, pp. 339–346, 2009.

[44] J. Nielsen, Usability engineering. Morgan Kaufmann, 1994.

[45] M. W. Lansdale and T. C. Ormerod, Understanding interfaces: A handbook of human-computer dialogue. Academic Press Professional, Inc., 1994.

[46] H. X. Lin, Y.-Y. Choong, and G. Salvendy, "A proposed index of usability: a method for comparing the relative usability of different software systems," Behaviour & information technology, vol. 16, no. 4–5, pp. 267–277, 1997.

[47] J. Brooke, "SUS-A quick and dirty usability scale," Usability evaluation in industry, vol. 189, no. 194, pp. 4–7, 1996.

[48] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability scale," Intl. Journal of Human–Computer Interaction, vol. 24, no. 6, pp. 574–594, 2008.

[49] T. Bray, "The javascript object notation (json) data interchange format," 2070–1721, 2014.

[50] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON schema," presented at the Proceedings of the 25th international conference on World Wide Web, 2016, pp. 263–273.

[51] O. Ben-Kiki, C. Evans, and B. Ingerson, "YAML Ain't Markup Language (YAML) (tm) Version 1.2," YAML.org, Sep. 2009. [Online]. Available: http://www.yaml.org/spec/1.2/spec.html

[52] V. R. B. G. Caldiera and H. D. Rombach, "The goal question metric approach," Encyclopedia of software engineering, pp. 528–532, 1994.

[53] Llopis, Marcel, Franch, Xavier and Soria, Manel, "Questionnaire used to evaluate the usability of two declarative computer languages to model geometric events in space." Zenodo, May 22, 2022.

[54] I. E. Allen and C. A. Seaman, "Likert scales and data analyses," Quality progress, vol. 40, no. 7, pp. 64–65, 2007.

[55] Llopis, Marcel, Franch, Xavier and Soria, Manel, "Questionnaire used to evaluate the usability of two declarative computer languages to model geometric events in space. Responses from usability questions.". Zenodo, May 31, 2022.

[56] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[57] M. E. Conway, "A multiprocessor system design," presented at the Proceedings of the November 12-14, 1963, fall joint computer conference, 1963, pp. 139–146.

[58] P. Betriu, M. Soria, J. L. Gutiérrez, M. Llopis, and A. Barlabé, "An assessment of different relay network topologies to improve Earth–Mars communications," Acta Astronautica, vol. 206, pp. 72–88, May 2023, doi: 10.1016/j.actaastro.2023.01.040.

[59] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," presented at the Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, 2010, pp. 307–309.

Appendix A

Questionnaire used to evaluate the usability of two declarative computer languages to model geometric events in space. The questionnaire starts with Section 1, which includes demographics questions. It is followed by Sections 2 and 3, which have two exercises to be completed by the respondents. In each exercise, respondents are asked to model a geometric event with the two language options. Sections 4-14 include questions concerned with usability.

Section 1 - Demographics

Prompt	Possible Answers
1. What is/are your college major(s)?	*Free-form text*
2. What is your educational level? *If you are currently working towards a degree, and this degree is higher than previous degrees, please choose the level of the degree you are currently working towards.*	• *BS* • *MS* • *PhD*
3. How many years have passed since you graduated from your highest degree?	*Number entry*

Prompt	Possible Answers
4. How many years of experience do you have in the aerospace domain? *Besides work experience, please also count school projects and classes if they are in the aerospace domain*	*Number entry*
5. Do you think of yourself as a software developer?	*Yes/No*

Section 2 - Exercise 1

In this exercise you will be asked to model and search for a simple geometric event with the two different language options. You will see an example with both languages. You will use the example as a starting point to model the new event.

Example event: "Between the start of year 2000 and the end of year 2005, the distance between the Earth and the Moon is less than 400,000 km.". Potential solutions:

Language A	Language B

DEF event1 AS
DISTANCE FROM Moon
TO Earth
LESS THAN 40K
DURING
01/01/2000:12/31/2005
SEARCH FOR event1
	DistanceQuery event1{
	observer: "Moon"
	target: "Earth"
	test: <
	amount: 40K
	start_time: 01/01/2000
	end_time: 12/31/2005
	}
	SEARCH FOR event1

Event: "Earth and Mars are at a distance of more than 70M km in all of 2021".

Prompt	Possible Good Answers (free-form text)
Please model the event above with Language A	DEF event_name AS DISTANCE FROM Earth TO Mars MORE THAN 70M DURING 01/01/2021:12/31/2021 SEARCH FOR event_name

Please model the event above with Language B	DistanceQuery event2{
	observer: "Earth"
	target: "Mars"
	test: >
	amount: 70M
	start_time: 01/01/2021
	end_time: 12/31/2021
	}
	SEARCH FOR event2

Section 2 - Exercise 2

In this exercise you will be asked to model a composite geometric event with the two different language options. You will see a new type of event and its definition in both languages. You will use that as a starting point to model the new event.

Occultation Events in Language A	Occultation Events in Language B

OccultationEvent event1{
type: {"any", "full", "annular", "partial"}
back_body: "backBody"
front_body: "frontBody"
observer: "observerBody"
start_time: startTime
end_time: endTime
}
SEARCH FOR event1
	DEF event1 AS
	{ANY, FULL, ANNULAR, PARTIAL}OCCULTATION
	BETWEEN BACK BODY body1
	AND FRONT BODY body2
	OBSERVED BY BODY observer
	DURING startTime:endTime
	SEARCH FOR event1

Event: "Earth and Mars are at a distance of more than 70M km, and at the same time, Mars and Earth are in full solar conjunction (solar conjunction is when the Sun is between two bodies). Use the 2010-2020 interval".

Prompt	Possible Good Answers (free-form text)

Please model the event above with Language A	DistanceQuery event1{
	observer: "Earth"
	target: "Mars"
	test: >
	amount: 70M
	start_time: 01/01/2010
	end_time: 12/31/2020
	}
	
	OccultationEvent event2{
	type: "full"
	back_body: "Mars"
	front_body: "Sun"
	observer: "Earth"
	start_time: 01/01/2010
	end_time: 12/31/2020
	}
	
	SEARCH FOR event1 AND event2

Please model the event above with Language B	DEF event1 AS FULL OCCULTATION BETWEEN BACK BODY Mars AND FRONT BODY Sun OBSERVED BY BODY Earth DURING 01/01/2010:12/31/2020 DEF event2 AS DISTANCE FROM Moon TO Earth MORE THAN 70M DURING 01/01/2010:12/31/2020 SEARCH FOR event1 AND event2

Sections 4-14 - Usability Questions

These sections included an example in each language option, and below each example, the statements below. Respondents selected a response on a Likert 1-5 scale for each question and for each language option. In this Likert 1-5 scale, 1 indicated "Strongly Disagree" and 5 indicated "Strongly Agree". Section 4 was a question on readability, whereas Sections 5-14 were part of the System Usability Scale (SUS).

- Section 4: I think the language above is readable by a scientist or engineer
- Section 5: I think that I would like to use the language above frequently
- Section 6: I found the language above unnecessarily complex
- Section 7: I thought the language above was easy to use
- Section 8: I think that I would need the support of a technical person to be able to use the language above
- Section 9: I found the various functions in the language above were well integrated
- Section 10: I thought there was too much inconsistency in the language above
- Section 11: I would imagine that most people would learn to use the language above very quickly
- Section 12: I found the language above very cumbersome (awkward) to use
- Section 13: I felt very confident using the language above
- Section 14: I needed to learn a lot of things before I could get going with the language above

Appendix B

This Appendix presents the response count of the usability questions in Appendix A, "Sections 4-14 - Usability Questions," for each sample (JPL and UPC), as well as the aggregate for both. The response count is provided for each question and its Likert scale response.

JPL Sample

	JPL Sample					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I think the language above is readable by a scientist or engineer	NL-Based	**10**	8	0	0	0
	Key-Value Pair	4	4	**8**	2	0
I think that I would like to use the language above frequently	NL-Based	**10**	7	1	0	0
	Key-Value Pair	**10**	8	0	0	0
I found the language above unnecessarily complex	NL-Based	0	0	1	5	**12**
	Key-Value Pair	0	0	4	4	**10**

	JPL Sample					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I thought the language above was easy to use	NL-Based	**10**	8	0	0	0
	Key-Value Pair	**14**	4	0	0	0
I think that I would need the support of a technical person to be able to use the language above	NL-Based	0	2	0	4	**12**
	Key-Value Pair	0	0	2	4	**12**
I found the various functions in the language above were well integrated	NL-Based	**14**	4	0	0	0
	Key-Value Pair	**12**	4	2	0	0
I thought there was too much inconsistency in the language above	NL-Based	0	1	1	2	**14**
	Key-Value Pair	0	1	1	0	**16**

	JPL Sample					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I would imagine that most people would learn to use the language above very quickly	NL-Based	**14**	2	2	0	0
	Key-Value Pair	**12**	4	2	0	0
I found the language above very cumbersome (awkward) to use	NL-Based	0	0	0	6	**12**
	Key-Value Pair	0	0	0	5	**13**
I felt very confident using the language above	NL-Based	**12**	5	1	0	0
	Key-Value Pair	**12**	5	1	0	0
I needed to learn a lot of things before I could get going with the language above	NL-Based	0	1	0	1	**16**
	Key-Value Pair	0	1	0	6	**11**

UPC Sample

	UPC Sample					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I think the language above is readable by a scientist or engineer	NL-Based	**9**	0	0	0	0
	Key-Value Pair	0	**4**	**4**	1	0
I think that I would like to use the language above frequently	NL-Based	1	**7**	1	0	0
	Key-Value Pair	**8**	1	0	0	0
I found the language above unnecessarily complex	NL-Based	0	0	0	2	**7**
	Key-Value Pair	0	0	0	3	**6**
I thought the language above was easy to use	NL-Based	**8**	1	0	0	0
	Key-Value Pair	**6**	2	1	0	0

	UPC Sample					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I think that I would need the support of a technical person to be able to use the language above	NL-Based	0	0	0	0	**9**
	Key-Value Pair	0	1	2	1	**5**
I found the various functions in the language above were well integrated	NL-Based	**6**	0	2	1	0
	Key-Value Pair	**6**	3	0	0	0
I thought there was too much inconsistency in the language above	NL-Based	0	0	1	0	**8**
	Key-Value Pair	0	0	0	1	**8**
I would imagine that most people would learn to use the language above very quickly	NL-Based	**7**	0	1	1	0
	Key-Value Pair	**4**	3	2	0	0

	UPC Sample					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I found the language above very cumbersome (awkward) to use	NL-Based	0	0	0	**7**	2
	Key-Value Pair	0	0	0	2	**7**
I felt very confident using the language above	NL-Based	**6**	2	1	0	0
	Key-Value Pair	**8**	1	0	0	0
I needed to learn a lot of things before I could get going with the language above	NL-Based	0	0	0	1	**8**
	Key-Value Pair	0	0	1	2	**6**

Aggregate (JPL plus UPC)

	Aggregate					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I think the language above is readable by a scientist or engineer	NL-Based	**19**	8	0	0	0
	Key-Value Pair	4	8	**12**	3	0
I think that I would like to use the language above frequently	NL-Based	11	**14**	2	0	0
	Key-Value Pair	**18**	9	0	0	0
I found the language above unnecessarily complex	NL-Based	0	0	1	7	**19**
	Key-Value Pair	0	0	4	7	**16**
I thought the language above was easy to use	NL-Based	**18**	9	0	0	0
	Key-Value Pair	**20**	6	1	0	0

	Aggregate					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I think that I would need the support of a technical person to be able to use the language above	NL-Based	0	2	0	4	**21**
	Key-Value Pair	0	1	4	5	**17**
I found the various functions in the language above were well integrated	NL-Based	**20**	4	2	1	0
	Key-Value Pair	**18**	7	2	0	0
I thought there was too much inconsistency in the language above	NL-Based	0	1	2	2	**22**
	Key-Value Pair	0	1	1	1	**24**
I would imagine that most people would learn to use the language above very quickly	NL-Based	**21**	0	3	3	0
	Key-Value Pair	**16**	7	4	0	0

	Aggregate					
Question	**Language**	**Strongly Agree**	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I found the language above very cumbersome (awkward) to use	NL-Based	0	0	0	13	**14**
	Key-Value Pair	0	0	0	7	**20**
I felt very confident using the language above	NL-Based	**18**	7	2	0	0
	Key-Value Pair	**20**	6	1	0	0
I needed to learn a lot of things before I could get going with the language above	NL-Based	0	1	0	2	**24**
	Key-Value Pair	0	1	1	8	**17**