Acta Astronautica 183 (2021) 319–329

Contents lists available at ScienceDirect

Acta Astronautica

journal homepage: www.elsevier.com/locate/actaastro

Tychonis: A model-based approach to define and search for geometric events in space

M. Llopis [a,*], M. Soria [b], X. Franch [c]

[a] Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA, 91109, USA
[b] Universitat Politècnica de Catalunya - ESEIAAT, C/ Colom, 1-11, 08222, Terrassa, Spain
[c] Universitat Politècnica de Catalunya, Campus Nord, C/Jordi Girona Salgado 1-3, 08034, Barcelona, Spain

ARTICLE INFO

Keywords:
Science planning
Opportunity search
Geometric events
Software architecture
Metamodeling
Reusability
Extensibility

ABSTRACT

Space missions need software in order to be aware of the occurrence of geometric events in different phases of their project lifecycle. However, current software packages that provide capabilities to search for geometric events might lack desirable software qualities such as extensibility and reusability, and some conflate the definition of events with the search for the same. In order to improve the state of the art, we propose a framework named Tychonis that, via metamodeling techniques and conscious use of object-oriented design best practices, (i) can integrate with current and future mission software, (ii) enables end-users to extend opportunities and search algorithms without modifying the tools that use the framework, and (iii) promotes the cross-mission reusability of the framework and its extensions. These attributes make it less costly for projects to develop software to search for geometric events. Tychonis is provided as a software library that allows missions to add their own data structures and constructs to the framework, including geometric event types and search algorithms. This is accomplished with Tychonis' use of the Eclipse Modeling Framework (EMF) to capture its metamodel, which enables the generation of Java classes that represent built-in or user-developed Tychonis constructs. In the present paper, we elaborate on the impetus for the development of Tychonis, show its proposed design and use, and discuss growth opportunities for future consideration.

1. Introduction

Space missions use the terms *opportunity*, or *geometric opportunity*, to define a geometric constraint of interest. Examples of an opportunity could be "*the spacecraft is within 300.*000 km *of Dione*", or "*the Sun is occulted by Jupiter or Europa as seen from the spacecraft*". This interest in knowing when an opportunity occurs can be driven by a number of engineering or science needs; for instance, to develop a science observation [1], to inform mission design [2], or to archive data that can assist in future efforts [3]. Today, the need for specialized opportunity search software becomes evident given the difficulty involved in defining and searching for most opportunities, either via analytical or numerical means, as they involve complex celestial mechanics methods.

Current opportunity search software typically comprises two families of tools:

(1) Mission-developed scripts. These scripts can be created with toolkits like Matlab [4] or languages like Python with support from libraries like SPICE [5] or MONTE [6]. Whenever there is a need for a mission to search for an opportunity whose search algorithm has not yet been implemented, an engineer or scientist will clone and modify an existing script or create a new script. In general, the lifecycle of these scripts is tied to that of a mission as their development begins when a mission is conceived, and their maintenance and use stop when a mission ends. Given that these scripts might be designed within the context of only one mission, there could be obstacles to reuse them in other missions. These obstacles include but are not limited to: not being generic enough in the definition of an opportunity, the use of mission-specific frameworks or tools, or a develop-as-you-need mentality that might preclude future extensibility. More importantly, this approach can induce programming errors as it does not separate between the modeling of an opportunity and its resolution algorithm. In other words, this method breaks the *separation of concerns* design principle [7], which states that "*software should be decomposed in such a way that different concerns or aspects of the*

* Corresponding author.
E-mail addresses: marcel.llopis@jpl.nasa.gov (M. Llopis), manel.soria@upc.edu (M. Soria), franch@essi.upc.edu (X. Franch).

https://doi.org/10.1016/j.actaastro.2021.01.057
Received 13 June 2020; Received in revised form 3 December 2020; Accepted 29 January 2021
Available online 24 February 2021

problem at hand are solved in well-separated modules or parts of the software".

(2) Multi-mission frameworks. This comprises software designed to be used across missions and provides a set of frequently used capabilities that can be extended by mission staff. Frameworks that operate in this fashion can be provided as either capabilities in a software library like SPICE's Geometry Finder [8]; or capabilities in an end-user software package like the Science Opportunity Analyzer (SOA) [9], WebGeoCalc [10], Percy [11], the Solar System Operations Lab (SOLab) [12], or the Mission Analysis and Payload Planning System (MAPPS) [13]. While some of these approaches 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 methods with different parameters. Should those 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 above. On the other hand, while a tool like SOA provides a separation between the definition of an opportunity and the search of the same, it doesn't 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 that a development team augments the software with additional opportunities. The latter approach could entail a wait longer than the project can afford due to more protracted development, release, and deployment cycles linked to larger development efforts.

From the reasoning above, the authors understand the main technical shortcomings of the current approaches to be: (i) minimal separation between the definition of an opportunity and the search for the same; (ii) less-than-optimal reusability of opportunity definitions and search algorithms across missions; and (iii) a complex, and potentially ad hoc extensibility mechanism for opportunities and their search algorithms. These challenges result in increased development costs for missions and a reduced ability to capitalize on knowledge across them. In response to the identified shortcomings, the aim of the present document is to describe a software framework named Tychonis[1] that, by virtue of its design, (i) can integrate with current and future mission software, (ii) enables end-users to extend opportunities and search algorithms without modifying the tools that use the framework, and (iii) promotes the cross-mission reusability of developments implemented in it.

The rest of this paper describes the framework at different levels and provides examples of how users would extend and utilize it. The paper's main sections are "2. Approach", which discusses the technology choices, the use cases the software implements, and a high-level view of the constructs that support the use cases; "3. Structure and extensibility", which provides a more in-depth explanation of the constructs and their relationships, along with a discussion as to how end-users would augment the framework and incorporate their own constructs; "4. Searching for opportunities", which provides code-based and textual descriptions of how end-users model and search for opportunities; and "5. Discussion", which analyzes certain design decisions and details known future steps.

[1] Named after the book "Tychonis Brahe Astronomiae Instauratae Mechanica" where the XVI century Danish astronomer Tycho Brahe describes the instruments he used to measure star and planetary positions accurate to within 1 min, or 1/60th of a degree.

2. Approach

To address the comments related to the state of the art of opportunity search software, Tychonis provides facilities to extend its own capabilities in a structured manner. This is analogous to how other multi-mission tools in the mission planning and execution domain separate the core functionality of the software from the specifics about how missions use the software itself. APGen [14], for instance, is a tool to simulate spacecraft activity plans, obtain the effects on the spacecraft's subsystems, and schedule activities for the spacecraft to execute. In this case, the simulator, the language to define spacecraft behavior models, the user interface, etc. are part of the multi-mission "core" capabilities of the software, whereas user-developed mission-specific capabilities such as specific spacecraft behavior models, spacecraft states, mission schedulers, etc. belong to the "adaptation" layer. Multi-mission developers maintain the "core" and own the release of new versions of APGen, while missions modify the adaptation layer and execute the simulator and the schedulers.

In a similar manner, Tychonis provides extensible and reusable capabilities via an object-oriented metamodel [15]. The choice of the word "metamodel" is not accidental since Tychonis' metamodel is used to create ontological types of opportunities and search logic. It is relevant to note that while Tychonis contains a metamodel, it also contains models implemented in it. Unless extending the framework, end-users operate in model space, where they (or software under their command) create models that adhere to generalized constructs in the metamodel. In this document, the authors will use the term "metamodel" when referring to both the metamodel and model parts of Tychonis.

Tychonis uses the Eclipse Modeling Framework (EMF) [16] to define its metamodel. EMF, in turn, provides the Ecore modeling capability, which is similar to the Unified Modeling Language (UML) [17] 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 Tychonis, these generated Java classes are a reproduction of the metamodel and can be used to create, for example, geometric opportunities types that abide by templatized representations within the metamodel. Other advantages of using EMF are its capabilities to (i) validate user objects, (ii) package a model and its generated classes as a Java library, (iii) modify a model during run-time, and (iv) introspect the model definition and its relationships. These capabilities are necessary 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 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 what `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 creation of these constructs is in response to the need to separate the modeling and resolution concerns within the software. `Query` classes model types of opportunities and their relations, `Solver` classes contain the code that searches for an atomic opportunity modeled in one `Query` instance, `SolverStrategy` classes associate `Solver` instances to `Query` instances and resolve composite opportunities, and the `Result` class describes the data returned after an opportunity has been searched. This separation delineates the boundary between modeling and resolution and makes it easier to support the different use cases the authors have identified.

2.1. Use cases

Tychonis supports three types of use cases: Integration use cases, End-user use cases, and Maintenance use cases. 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.

Integration use cases are developer-oriented and contain the retrieval of the Tychonis framework from a repository and the writing of computer code to connect mission software with Tychonis. Mission software can support multiple ways to input and represent opportunities, including form-based or textual descriptions, but also projectional editors (also known as structured editors or syntax-directed editors) [18], which provide a visual representation of an opportunity's structure. Integration entails the development of code that uses Tychonis `Query` class-based representations directly, or the implementation of logic that translates mission software constructs into Tychonis' representations. Once this integration is completed, mission software will not need to be updated and recompiled every time a user modifies the metamodel and generates new Java classes. Software that integrates with the framework can poll the framework to understand what `Query` or `Solver` classes are available and, as a result, update its own understanding of what opportunities it can search for, and present those to the user.

End-user use cases (Fig. 1, top half) are interaction-oriented and describe how Tychonis is called by other software as a consequence of user actions. An engineer, with mission software that has already been integrated with Tychonis, will model opportunities on a user interface (UI). This will result in mission software instantiating `Query` classes.

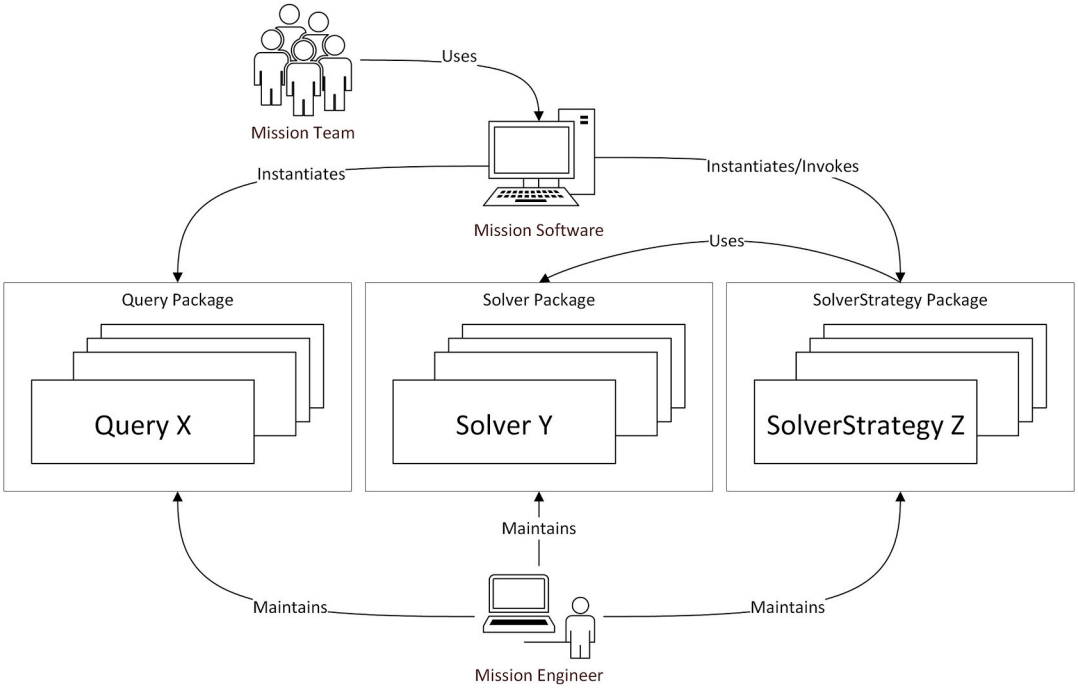

Fig. 1. End-User Use Cases (top) A mission team operates mission software that integrates with Tychonis. Mission software instantiates `Query` classes that represent an opportunity. The same or other software invokes a `SolverStrategy` to resolve the opportunity. Maintenance-Driven Use Cases (bottom): 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.

After this, the user can direct mission software to search for an opportunity, which will cause mission software to instantiate a `SolverStrategy` class and trigger the resolution of the opportunity. Tychonis will then return an instance of a `Result` class that will provide all the relevant data about the found opportunity. The data within the `Result` instance can then be used directly by the mission team or displayed by the calling software on the UI.

Maintenance use cases (Fig. 1, bottom half) are developer-oriented and comprise the addition, modification, or deletion of `Query`, `Solver`, and `SolverStrategy` classes in order to augment Tychonis' capabilities. The framework provides a number of built-in `Query` classes that define frequently used opportunities, a number of `Solver` classes, and a built-in `SolverStrategy`. If users are not satisfied with the built-in classes, then they can add, delete, or modify classes of each type by first modifying the Ecore metamodel, then generating new Java versions of the modified classes or entirely new classes via EMF, and finally adding code to the generated `Solver` or `SolverStrategy` Java classes. It is worth noting that `Query` classes only represent a data model with no algorithms, hence the need to modify the generated Java classes will be limited.

2.2. Opportunities

Semenov details a number of opportunities that can be searched with the WebGeoCalc software [10]. Those are Position, Angular Separation, Distance, Sub-Point, Occultation, Surface Intercept, Target in Field of View, and Ray in Field of View. In order to aid in the development of requirements and testing, current versions of Tychonis model and resolve the aforementioned opportunities by default. The opportunities within WebGeoCalc, however, are atomic in that users can only search for one opportunity at a time. In contrast, the authors of [9] describe how SOA can combine atomic opportunities into a larger opportunity via Boolean logic. Tychonis has adopted this idea and defines a set of `Query` classes in the metamodel that can reference and relate `Query` instances hierarchically in a tree. At this point, only Boolean `Query` classes contain such references since the built-in geometric `Query` classes are considered terminal within the tree structure. Fig. 2 shows an example of a composite opportunity named Qx that contains several Boolean relationships.

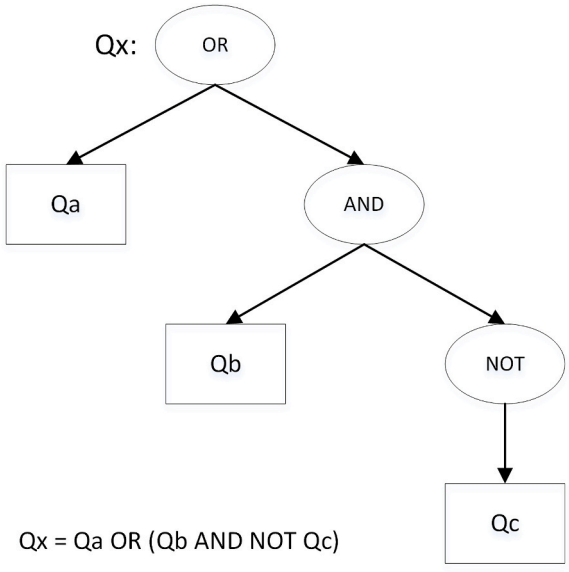

Fig. 2. A composite opportunity Qx: Ellipses represent Boolean `Query` class instances whereas rectangles represent terminal geometric `Query` class instances.

3. Structure and extensibility

This section details the most relevant classes involved in Tychonis' use and extension from an object-oriented modeling perspective [19]. In most cases, the process for extending Tychonis' components follows a repeatable pattern: The user augments the Ecore metamodel with additional classes that implement an interface or extend another existing class, generates Java classes with EMF, implements algorithms within the generated classes, and finally compiles and packages the new classes into a Java ARchive (JAR) file. In this workflow, mission software will not need to be recompiled in order to benefit from the additions.

The ensuing discussion will be kept at a high level and only include the most important relationships and core classes. Note that the Ecore modeling framework provides its own concepts to define metamodels; for instance, an `EClass` is an Ecore class while an `EReference` is a reference from an `EClass` to another `EClass`. These Ecore-specific names will not be addressed directly, but the reader shall assume that the use of object-oriented constructs alludes to the Ecore form of the same. Also, in order to improve reading fluidity, when the text references a class, the "class" suffix might be dropped (e.g., the `Query` class being referred to as `Query`) unless needed. References to instance names will be followed by the word "instance" (e.g., `Query` instance).

3.1. Query

In its definition of the `Query` metamodel (Fig. 3), Tychonis provides a base class named `Query` that contains a coordinate system used by

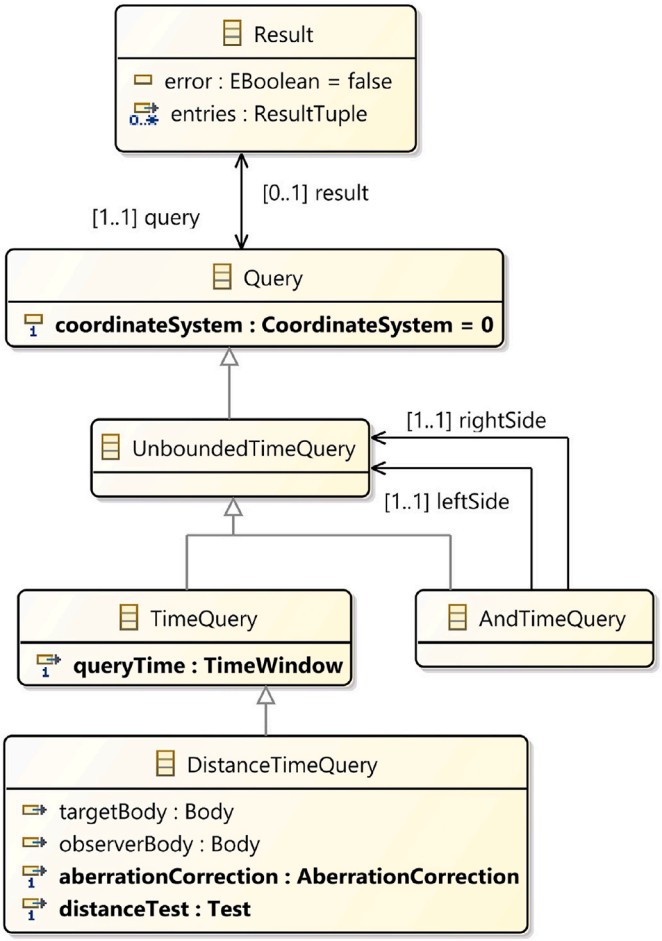

Fig. 3. The `Query` class hierarchy with two example extensions, `DistanceTimeQuery` and `AndTimeQuery`.

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 once 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 difference between `UnboundedQuery` and `TimeQuery` is that `UnboundedTimeQuery` does not set a time interval in which to define an opportunity whereas `TimeQuery` does bound an opportunity with a start time and an end time. This is needed in order to avoid searching for an opportunity in an indefinitely large search space. Also, Boolean `Query` classes don't need to define a time boundary because it is implicit in the query's reference to other queries and in the execution of the Boolean operation. In consequence, `TimeQuery` becomes the parent class for the built-in atomic geometric `Query` classes while `UnboundedTimeQuery` becomes the parent class for the built-in Boolean `Query` classes.

If a user decides to add a new type of `Query`, the first step would be to identify whether it would be preferable to extend the `UnboundedTimeQuery` or `TimeQuery` classes, built-in geometric or Boolean `Query` classes, or mission-developed `Query` classes. The choice would depend on the level of code reusability the user is seeking.

Concrete extensibility examples include:

(1) The definition of `DistanceTimeQuery`, a built-in geometric query that models opportunities where a body is less than or more than a specified distance from another body. This query is modeled by extending `TimeQuery` and referencing two `Body` instances (a target and an observer), a `DistanceTest` instance that includes the distance inequality, and an `AberrationCorrection` instance for light speed and relative velocity magnitude corrections.

(2) `AndTimeQuery`, a built-in Boolean query that models an AND Boolean relationship between two other `Query` instances. This type of query extends `UnboundedTimeQuery`, as it has no need to explicitly define a search time window, and references two other `UnboundedTimeQuery` instances that capture the two sides of the AND operation.

3.2. Solver

`Solver` contains the code that resolves a `Query`. In general, there should be at least one `Solver` for every `Query` users would need to resolve. One particular case in which there could be more than one `Solver` class for a specific `Query` class is when users are experimenting with a new `Solver` with improved code or new libraries that might be more performant than the current implementation. In such a scenario, the previous `Solver` can be phased out once users certify the new `Solver` is an improvement over the previous one and that the new one is as accurate as expected. The onus is on `SolverStrategy` to choose what `Solver` to use for a specific `Query`.

`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 will implement both the `solve()` and `validate()` methods of the `Solver` interface. The `solve()` method implements the search algorithm for the `Query` instance passed as a parameter, and `validate()` checks that the `Query` instance passed as a parameter can be resolved by the `Solver` class. The `validate()` method returns a list of `ValidationResult` objects which contain information about the results of the validation; i.e., validation passed/not passed, source of errors, human-readable message, etc. Calling the `solve()` method implies an internal call to the `validate()` method to ensure a viable resolution.

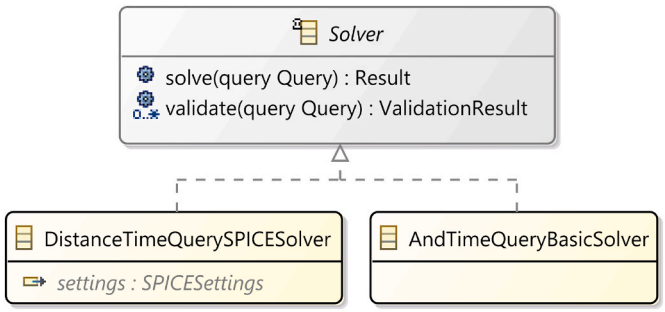

Fig. 4. The `Solver` interface with its two methods and two built-in `Solver` classes: a `DistanceTimeQuerySolver` implemented with SPICE and an `AndTimeQuerySolver`.

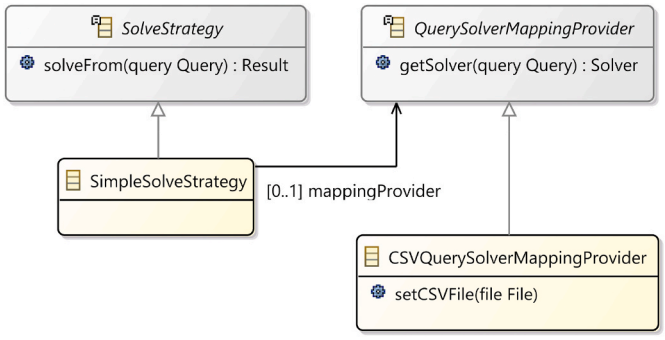

Fig. 5. The `SolveStrategy` interface with `SimpleSolveStrategy` as its default implementation. Tychonis also provides extensibility for `SimpleSolveStrategy` by letting users implement their own dependency injection provider via `QuerySolverMappingProvider`.

Fig. 4 shows the described structure and also depicts two extensions, `DistanceTimeQuerySPICESolver` and `AndTimeQueryBasicSolver`, which are both built-in `Solver` classes. The latter implements an intersection of time windows from the two sides of an AND relationship defined in an `AndTimeQuery` object, while the former finds time windows for `DistanceTimeQuery` objects using SPICE. Note that a `Solver` for `DistanceTimeQuery` objects could have been implemented through the use of libraries or means other than SPICE. However, in this case, as in other `Solver` classes bundled by default within Tychonis, SPICE was chosen for implementation. Also worth discussing is that `DistanceTimeQuerySPICESolver` can search for distance opportunities even when either one or both `Body` parameters in a `DistanceTimeQuery` instance are unknown; as represented by a `null` Java value. If a `Body` instance is set to null, the resolution algorithm within `DistanceTimeQuerySPICESolver`'s `solve()` method will then search for all bodies loaded by `Solver` that satisfy the `distanceTest` constraint within the `DistanceTimeQuery` instance.

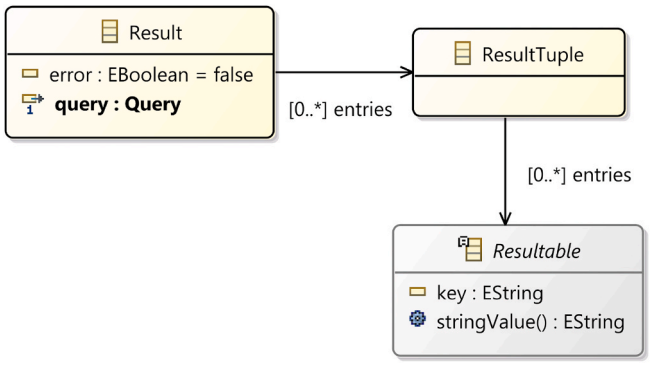

Fig. 6. The `Result` class hierarchy. `Result` contains a tabular structure that includes multiple `ResultTuple` classes as rows. Each `ResultTuple` object, in turn, contains `Resultable` objects that represent the table's cells.

3.3. SolverStrategy

The `SolverStrategy` interface is an extension point used to develop the logic behind `Query`-to-`Solver` assignments, the algorithm to traverse a tree composed of multiple `Query` instances, and the execution of `Solver` code for each `Query` node. Classes that implement this interface will incorporate a method called `solveFrom`() that receives the root `Query` instance in the tree for which the user would like to receive a `Result` instance. This description and what follows is captured graphically in Fig. 5.

Tychonis provides a built-in implementation of SolverStrategy called `SimpleSolverStrategy` (see Fig. 6). This implementation and its helper classes are provided as an example of `SolverStrategy` extensibility:

- `SimpleSolverStrategy` implements `SolverStrategy`. It resolves a `Query` tree by first resolving the leaf nodes, then by propagating results upwards and successively resolving `Query` nodes until the `Query` instance passed as a parameter is resolved. Note that the initial implementation of this resolution algorithm is not parallelized, i.e., it operates in a serial manner even though it could be parallelized due to data independence across nodes.
- `QuerySolverMappingProvider` is an interface used by `SimpleSolverStrategy`. Its purpose is to provide a `Solver` instance for a specific `Query` instance through a method known as dependency injection [20]. Dependency injection proposes that the implementation of at least one of two or more dependent components should be realized at runtime via another component. 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 user extensibility, as missions might want to provide `Query`-to-`Solver` dependencies through various means.
- `CSVQuerySolverMappingProvider` implements the `QuerySolverMappingProvider` interface. It retrieves the mappings from a Comma-Separated Values (CSV) file and provides a `Solver` through the `getSolver`() method defined in the `QuerySolverMappingProvider` interface.

3.3. 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. This is a bidirectional relationship as a `Query` object also contains a reference to its `Result` instance. The definition of the bidirectional relationship between a `Query` instance and a `Result` instance is the responsibility of `SolverStrategy` as `Solver` is only responsible for the creation of the `Result` object.

`Result` is not designed to be modified or extended by users, but it provisions support to reference different data types or classes. This is due 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`. Data type agnosticism is achieved by supporting the description of results in a tabular format in which each cell of a table can be an instance of a class that implements the `Resultable` interface. The end result is that `Solver` developers can decide freely what data a `Solver` will return.

In order to structurally describe the result table, `Result` contains a list of references to `ResultTuple` objects that act as the table rows. Each `ResultTuple` object within a `Result` instance, in turn, references instances of classes that implement the `Resultable` interface. `Resultable` is implemented as a <key, value> pair via a `String` key within each `Resultable`, and a `String` value via the `stringValue`() method that would need to be implemented by the user. A typical use of <key,value> is to employ the key to explain what type of data this `Resultable` contains, and the value to provide a serialized text version of the object that can be used by external software. The `Body` class used to denote space bodies is a prime example of this fact as it implements the `Resultable` interface. Particularly, `Body` instances can be used to define "observer", "target", and other relationships between bodies in `Result` instance rows (e.g., <"target","Venus"> and <"observer", "Cassini">). Tychonis also provides means for searching data types within `Result` instances. Please refer to Section 4.3 for an example.

4. Searching for opportunities

Having covered the prerequisite background on the most relevant classes involved in Tychonis' use and extension, the authors can provide a pattern for the use of the framework to search for opportunities. This pattern is composed of three steps: (i) modeling of opportunities, (ii) resolution, and (iii) parsing of results. From the user perspective, software that uses Tychonis typically provides support for the three steps at a more interaction-oriented level: users manipulate a UI to define opportunities (modeling), users choose an option on the UI to trigger a search (resolution), and finally, the UI shows the search results (parsing). All these user interactions would result in internal instantiations of EMF-generated Java classes provided by Tychonis. The separation of the different software concerns (i.e. modeling, resolution, and parsing) is one of the key improvements over other opportunity search software, as it provides a framework for conceptualizing needs as components that change at different times for different reasons [21]. From a practical perspective, it is hard for scientists and engineers to read code that mixes both modeling and resolution, as they need to have expertise on both the scientific or engineering definition of an opportunity and in how a software library (e.g., SPICE, MONTE) searches for it. Tychonis proposes a solution to this matter.

In order to illustrate the previous comments within the context of the forthcoming discussion, this paper will provide examples of these three phases. These examples, while useful, will be static, meaning that the code presented will not be parameterized as it would if it was integrated with real-life mission software. An ideal integration between Tychonis

and mission software would programmatically model opportunities from the user's input and would not involve writing code such as the one provided. As relevant additional background, integration with real-life mission software could entail direct use of the Tychonis' EMF-generated Java classes; however, an alternative path for integration would entail using an Ecore XML description file (tychonis.ecore) that describes the full Tychonis metamodel. Software that relies on the description file could then use reflection [22] to instantiate classes from the Tychonis metamodel. A benefit of this approach is the fact that no modification (and no recompilation) would be needed on the software that uses Tychonis if the metamodel changes and its Java classes do need to be regenerated.

4.1. Modeling

The goal of this step is to model or define an opportunity in unambiguous terms. This lack of ambiguity is provided by built-in `Query` classes or by `Query` classes that users define within the metamodel. Ambiguity in the definition of an opportunity is an important topic related to the ambiguity inherent in natural languages that are used in fields in which precision is crucial. When a natural (and typically ambiguous) language like English is used to define opportunities, interpretation of what an opportunity means can vary depending on the receiver's understanding of the sentence that communicates a specific opportunity. Take for instance 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 someone else might think the front-back relationship is with the Sun in front of Venus. There is also the discussion of time; what is the time window in which such an opportunity should be searched? The role of a metamodel in these circumstances is to provide clarity as to what it means to capture domain-specific knowledge. It sets rules about such a domain.

Continuing with the previous opportunity example, the definition of the `OccultationTimeQuery` class bundled within Tychonis defines an atomic occultation opportunity as primarily needing (i) a front body, (ii) a back body, (iii) an observer, (iv) a type (full, annular, or partial), (v) an aberration correction, and (vi) a time window via its inheritance of `TimeQuery`. Given the strictness of `OccultationTimeQuery`, engineers can define a more concise version of the previous opportunity as *O1:* "*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*". The Java code to model O1 is found in Listing 1 and it assumes the use of the default aberration correction and the default occultation type (i.e., any). Note that while the code shown models an opportunity successfully, it does not exemplify an integration with mission software that considers parameterized and variable user input. Also, Listing 1 shows a number of ancillary EMF idioms, capabilities, and autogenerated constructs. Among the latter is `Tychonis Factory`, an autogenerated class used to create Tychonis class instances which are then populated with data via getters and setters. These getters and setters respect the metamodel and are also autogenerated by EMF. The last comment on Listing 1 is that, if the user chooses not to set one of the `Body` parameters in an `OccultationTimeQuery` instance, then that parameter will become an unknown that also needs to be searched for. For instance, if the developer omitted line 16, the opportunity defined in `anOccultationQuery` would then be defined as "*there is any type of occultation between any back body and the Sun, as seen from Earth anytime in 2019*".

Listing 1
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*"

```
1.  Body frontBody = TychonisFactory.eINSTANCE.createBody();
2.  frontBody.setName("SUN");
3.  Body backBody = TychonisFactory.eINSTANCE.createBody();
4.  backBody.setName("VENUS");
5.  Body observerBody = TychonisFactory.eINSTANCE.createBody();
6.  observerBody.setName("EARTH");
7.  OccultationTimeQuery anOccultationQuery = TychonisFactory.eINSTANCE.createOccultationTimeQuery();
8.  TimeWindow searchWindow = TychonisFactory.eINSTANCE.createTimeWindow();
9.  TimeInstant startTime = TychonisFactory.eINSTANCE.createTimeInstant();
10. startTime.setTimeFromString("01/01/2019");
11. TimeInstant endTime = TychonisFactory.eINSTANCE.createTimeInstant();
12. endTime.setTimeFromString("12/31/2019");
13. searchWindow.setStartTime(startTime);
14. searchWindow.setEndTime(endTime);
15. anOccultationQuery.setFrontBody(frontBody);
16. anOccultationQuery.setBackBody(backBody);
17. anOccultationQuery.setObserverBody(observerBody);
18. anOccultationQuery.setQueryTime(searchWindow);
```

While O1 is a good use example of the Tychonis meta-model, it might be a structurally simpler view of what can be accomplished with the framework. One could have, for instance, another opportunity that is defined this way *O2:* "*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 turned into a composite opportunity such as "*the distance between Earth and the moon is more than* 370 km *in the last three months of 2019, AND the distance between Earth and the Moon is less than* 390 km *in the last three months of 2019*". This opportunity is modeled in Listing 2 as a composite opportunity with two `DistanceTimeQuery` objects and one `AndTimeQuery` object. This example is not accidental, as it opens the door for additional comments on the extensibility of the metamodel beyond `Query`, `Solver`, and `SolverStrategy`. To illustrate, let's observe that Fig. 3 showed that `DistanceTimeQuery` contains a reference to a `Test` class instance. If `Test` can, at the moment, only be used to contain a less-than or greater-than inequality, a user could extend that class in the metamodel, potentially name it `BetweenTest`, and include within it an upper bound value and a lower bound value. This would consider an "is between" relationship which would negate the need for the composite distance opportunity with an AND.

Listing 2
Java code that models the O2 opportunity with two DistanceTimeQuery objects and one AndTimeQuery.

```
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);
```

4.2. Search

Searching for an opportunity implies choosing one of the available `SolverStrategy` classes and calling the `solveFrom()` method within the chosen `SolverStrategy`. In this scenario, the `solveFrom()` method receives the root `Query` node in the opportunity tree a user would want to start searching from. An example can be found in Listing 3, which shows code that solves opportunity O2 from Listing 2. The onus is on the chosen `SolverStrategy` to find a way to resolve the composite opportunity tree and, along the way, connect each `Query` node with a `Result` instance that contains the data created by the `Solver` classes used. As one can imagine, there are multiple ways to implement this logic; that's why Tychonis enables users to implement their own `SolverStrategy` beyond the built-in implementation.

Tychonis's built-in implementation is provided by the `SimpleSolverStrategy` class. It contains a post-order tree traversal algorithm [23] that visits and resolves all `Query` nodes. Every time the algorithm finds a `Query` node that is not solved but is solvable (i.e., the child `Query` nodes it depends on have already been solved or the node is a leaf node) it will use a lookup within a `QuerySolverMappingProvider` object that binds `Query` classes and `Solver` classes via the `getSolver()` method. This method will inspect the passed `Query` instance and return a `Solver` object to `SimpleSolverStrategy`. The received `Solver` object will then be used to solve the `Query` instance. This will be done for each `Query` node successively going left to right and upwards in the tree structure until the root `Query` node - the one passed to `solveFrom()` - is also resolved.

Also relevant is a discussion on `QuerySolverMappingProvider`'s built-in implementation. The main driver behind the creation of `QuerySolverMappingProvider` is to let users choose whether the mapping between a `Query` class and a `Solver` class should be statically defined in compiled code vs. dynamically defined via dependency injection, and in case the mapping should be dynamic, where this mapping should reside. A default implementation of `QuerySolverMappingProvider` is provided as `CSVQuerySolverMappingProvider`. This implementation reads a CSV file in which each row contains the name of a `Query` class name, followed by the name of the EMF factory class responsible for the creation of a `Solver` instance, and the name of the actual method in that factory that returns the `Solver` instance. This makes it possible for `CSVQuerySolverMappingProvider` to create a `Solver` instance via reflection when the framework requests a `Solver` instance with the `getSolver()` method.

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

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

4.3. Parsing results

At this point, each `Query` object in the opportunity tree has a reference to a `Result` object that contains result data generated by the `Solver` classes. The most typical scenario is for mission software to call the `get Result()` method on the root `Query` node to obtain the `Result` instance for the whole opportunity tree. Mission software would then

parse the `Result` object and inspect its component `ResultTuple` instances, which in turn are decomposable into `Resultable` objects. This provides a table-like structure that can be parsed hierarchically with ease. Listing 4 contains an example of how opportunity O2's result can be parsed and printed on a computer terminal.

SPICE-implemented `Solver` will check the body name contained in a `Query` exists in the kernels passed to the `Solver`. This is the approach taken by Tychonis, as it keeps the modeling and resolution steps separate at the expense of performing a less profound validation at modeling time.

Listing 4
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).

```
1.   Result result = andQuery.getResult();
2.   for (ResultTuple tuple : result.getEntries()) {
3.       for (Resultable resultable : tuple.getEntries()) {
4.           printKeyValue(resultable.getKey(), resultable.stringValue());
5.       }
6.       System.out.println();
7.   }
```

```
[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}]
```

In terms of extensibility, the fact that each cell in the `Result` table contains references to objects that implement the `Resultable` interface provides flexibility for engineers to decide what type of data goes into the table. Should a mission require the addition of a new data type into `Result` objects, an engineer would define this type as a class that implements the `Resultable` interface, then he/she would create or modify an existing `Solver` to place the new data type into the table. This flexibility, however, could make it complicated to retrieve the contents of the table by data type; it is for this reason that `Result` and `ResultTuple` provide methods to query their own contents. For instance, method `public <T> EList<T> getAll(Class<T> aClass)` can be used to obtain all `TimeWindow` instances within a `Result` instance if it's invoked this way: `result.getAll(TimeWindow.class)`.

5. Discussion

While Tychonis' design qualities provide for extension and reusability in a way that advances the current state of the art of the opportunity search software discipline, it is not free from examination by its creators and potential users. In the following paragraphs, the authors discuss strategic actions and opportunities for growth.

5.1. Software design vs. applicability

Future critical comments on the design of the framework might allude to the enforcement of separation of modeling and resolution concerns and the abstraction the framework promotes. For instance, searching for an opportunity with SPICE alone implies loading kernel files that contain body definitions, ephemeris data, etc. An implication is that a potential Tychonis `Query` validation call might want to check whether a body name (e.g., "EARTH") referenced in a `Query` instance exists in the system (i.e., in the SPICE kernels). However, this means the separation between the modeling and resolution steps of an opportunity is breached, as the `Query` object needs to know whether SPICE (or other libraries with their own idiosyncrasies) will be used to search for the opportunity. An alternative solution is for `Query` classes to delay the validation of some of their component parts via `Solver` code, e.g., a

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.

5.2. Textual language

The current implementation of Tychonis puts an emphasis on an initial investment in the infrastructural integration between mission software and the framework itself. There are user needs that can be served by Tychonis as long as a mission has an aptitude for software integration tasks, enough funding, and time; however, having those is not always possible. There are also certain cases in which a user needs to sporadically search for opportunities away from other mission software that might be too large to learn just to occasionally search for an opportunity. In order to appeal to such cases, the authors will be developing a less programming-oriented option to use the framework that involves a textual query language. This solution is similar to what occurs when a database user models a Structured query Language (SQL) [24] query and executes it on a Database Management System (DBMS). Tychonis would implement a grammar [25] that maps text input to query metamodel constructs, which in turn would be directly searchable by the system. This is akin to Percy's capability to express opportunities in textual form and search for the time windows in which they occur; with the caveat that Percy is not user-extensible in the way Tychonis is. The authors' research will involve the development of a user-friendly language, the integration of the language with the rest of Tychonis,

and explore avenues for manual and automated extension of the grammar if there are changes to the metamodel.

5.3. Performance enhancements

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 b_1, another sub-search could search for b_2, 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 [26] functions, or other avenues for implementation of the fork-join [27] 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 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 the Jet Propulsion Laboratory (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 [28] (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.

5.4. Release and adoption

The authors also place their focus on the importance of the integration with software in use by a mission in the proposal, development, or operations phase. The needs and available resources of a mission in these three phases vary: proposals have limited resources and thus would be willing to limit development time by adopting reusable software, missions in the development phase should have enough resources but become more risk-averse to new developments as launch date approaches, and missions in operations typically have sustained but diminishing funding and their risk position to adopt new technologies might become more lenient over time, especially after the prime mission objectives are completed. The aforementioned risk and resource availability profiles, together with a mission's continued need to search for geometric events, afford engineers windows of opportunity to adopt a framework like Tychonis. After all, it should be less risky and less expensive to incorporate a framework built on good design practices that addresses user needs than it is to adopt more ad-hoc approaches.

Tactically speaking, the authors' first step is to integrate Tychonis with SOA's capability for opportunity search [29]. This seems a natural step given the management of SOA's development is under the purview of one of the authors' teams at JPL. NASA's JPL-managed Psyche mission is currently using SOA in the development phase for preliminary science planning and it will also use the software during mission operations. In addition, NASA's JPL-managed Europa Clipper mission will use SOA during mission operations to schedule the activities of some of its instruments. The expectation is that the SOA/Tychonis integration will provide cost savings as opportunities will be developed once by either mission and enjoyed by both. It seems sensible to think that a successful integration with SOA and certifiable cost savings in Psyche and Europa Clipper can result in the integration of Tychonis with JPL software other than SOA. At the extra-JPL level, the authors will release Tychonis as an open-source codebase. This will result in widespread availability of the software and in potential adoption by research centers and universities. This adoption can, in turn, evolve into augmentations to Tychonis' open-source repository. Given the framework's abstraction from low-level programming, it is foreseeable how Tychonis could be used as a practical teaching tool in science and engineering courses on planetary dynamics, space communications, remote sensing, etc. Tychonis could, when needed, enable students to think about space geometry without overly concerning themselves with specific algorithms. More software-oriented courses can propose exercises on the development of opportunities and `Solver`s, and even use Tychonis as a case study on how good software practices can be beneficial in the development of larger software applications.

6. Conclusion

Current opportunity search software can be categorized as belonging to one of two categories: mission-developed scripts or multi-mission frameworks. Software in both categories might lack desirable qualities such as extensibility, reusability, and a separation between the definition of events and the search for the same. The existence of these shortcomings presents an opportunity to develop a software package that can advance the state of the art. In response to the challenge, the authors devised Tychonis; a framework that (i) can integrate with current and future mission software, (ii) enables end-users to extend opportunities and search algorithms without modifying the tools that use the framework, and (iii) promotes the cross-mission reusability of the framework and developments implemented in it. These capabilities are facilitated through conscious application of software design's best practices and the creation of an object-oriented metamodel. Tychonis' metamodel is developed with EMF and Ecore, which are able to autogenerate Java classes that reproduce the metamodel and can be integrated with mission software. These Java classes, and their representation in the metamodel, were designed to bound the modeling of an opportunity and the resolution of the same; with each concern assigned to a different class. In addition, while all these classes are user-extensible, Tychonis provides built-in implementations that can be used by missions from day one.

Looking towards the future, Tychonis presents some areas for growth: (i) the development of a textual language for non-programmers or users who are choosing not to integrate with other software, (ii) performance enhancements through the implementation of parallelized Solvers, (iii) the introduction of Tychonis in space missions to provide for final validation, and (iv) subsequent open-sourcing and adoption of the framework. The aspiration is for Tychonis to become a one-stop-shop for a mission's opportunity search needs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the SOA and SPICE development

teams for the inspiration that led to this effort. The authors would also like to thank the Planning and Execution Systems Section within JPL for its dedication to improving the state of the art of mission planning software. The work described in this paper was carried out at the Jet Propulsion Laboratory (JPL), California Institute of Technology (Caltech), under a contract with the National Aeronautics and Space Administration (NASA).

References

[1] T.D. Robinson, L. Maltagliati, M.S. Marley, J.J. Fortney, Titan solar occultation observations reveal transit spectra of a hazy world, Proc. Natl. Acad. Sci. Unit. States Am. 111 (25) (2014) 9042–9047.
[2] T. Lam, B. Buffington, S. Campagnola, A robust mission Tour for NASA's planned Europa Clipper mission, in: 2018 Space Flight Mechanics Meeting (P. 0202, 2018.
[3] L. Spilker, Cassini-Huygens: Recent Science Highlights and Cassini Mission Archive, 2019.
[4] E. Vinterhav, T. Karlsson, Script based software for ground station and mission support operations for the Swedish small satellite Odin, Acta Astronaut. 61 (10) (2007) 912–922.
[5] C.H. Acton Jr., Ancillary data services of NASA's navigation and ancillary information facility, Planet. Space Sci. 44 (1) (1996) 65–70.
[6] J. Smith, W. Taber, T. Drain, S. Evans, J. Evans, M. Guervara, W. Schulze, R. Sunseri, H.C. Wu, MONTE python for Deep Space Navigation, 2016.
[7] B. De Win, F. Piessens, W. Joosen, T. Verhanneman, November. On the importance of the separation-of-concerns principle in secure software engineering, in: Workshop on the Application of Engineering Principles to System Security Design, 2002, pp. 1–10.
[8] C. Acton, N. Bachman, B. Semenov, E. Wright, SPICE Tools Supporting Planetary Remote Sensing, 2016.
[9] M. Llopis, C.A. Polanskey, C.R. Lawler, C. Ortega, July. 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), IEEE, 2019, pp. 1–8.
[10] B. Semenov, WebGeocalc and cosmographia: modern tools to access OPS SPICE data, in: 2018 SpaceOps Conference, 2018, p. 2366.
[11] S. Stallcup, July. Solar system geometry and ephemeris processing for the HST, in: Observatory Operations to Optimize Scientific Return, vol. 3349, International Society for Optics and Photonics, 1998, pp. 402–409.
[12] M. Almeida, Solar system operations lab for constructing optimized science observations, in: SpaceOps 2012 Conference, 2012.
[13] P. Van Der Plas, MAPPS: a science planning tool supporting the ESA solar system missions, in: SpaceOps 2016 Conference, 2016.
[14] P.F. Maldague, S.S. Wissler, M.D. Lenda, D.F. Finnerty, APGEN scheduling: 15 years of experience in planning automation, in: SpaceOps 2014 Conference, 2014, p. 1809.
[15] R.F. Paige, D.S. Kolovos, F.A. Polack, A tutorial on metamodelling for grammar researchers, Sci. Comput. Program. 96 (2014) 396–416.
[16] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.
[17] J. Rumbaugh, G. Booch, I. Jacobson, The Unified Modeling Language Reference Manual, second ed., Addison-Wesley, Boston, 2010.
[18] T. Berger, M. Völter, H.P. Jensen, T. Dangprasert, J. Siegmund, November. Efficiency of projectional editing: a controlled experiment, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 763–774.
[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W.E. Lorensen, Object-oriented Modeling and Design, vol. 199, Prentice-hall, Englewood Cliffs, NJ, 1991. No. 1.
[20] M. Fowler, Inversion of Control Containers and the Dependency Injection Pattern, online, martinfowler.com, 2004. Available at: https://martinfowler.com/articles/injection.html. (Accessed 26 May 2020).
[21] R.C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2002.
[22] P. Maes, Concepts and experiments in computational reflection, ACM Sigplan Not. 22 (12) (1987) 147–155.
[23] D.E. Knuth, The Art of Computer Programming, vol. 1, Pearson Education, 1997.
[24] D.D. Chamberlin, R.F. Boyce, May. SEQUEL: a structured English query language, in: Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, 1974, pp. 249–264.
[25] A.V. Aho, J.D. Ullman, R. Sethi, Compilers: Principles, Techniques and Tools, 1986.
[26] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, 2004.
[27] M.E. Conway, November. A multiprocessor system design, in: Proceedings of the November 12-14, 1963, Fall Joint Computer Conference, 1963, pp. 139–146.
[28] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, The Java Virtual Machine Specification, Pearson Education, 2014.
[29] M. Llopis, X. Franch, M. Soria, Integrating the science opportunity analyzer with a reusable opportunity search framework, in: ASCEND 2020, 2020, p. 4221.