Sailing Towards an Expressive Scheduling Language for Europa Clipper

Adrien Maillard and **Marijke Jorritsma** and **Steve Schaffer**
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
Pasadena, California 91109

Abstract

The mission planners for NASA's Europa Clipper deep-space mission use automated scheduling software to generate activity plans and command sequences for multiple instruments and subsystems before sending the sequences for execution on board. Both science and engineering planners must translate their intents into expressions of mission constraints, goals, and preferences that the scheduling engine understands. This paper describes the ongoing development of a dedicated domain-specific java-embedded language that can efficiently and accurately capture such concerns for the Europa Clipper mission, along with a user-interface companion for the language.

Introduction

Automated scheduling for space missions present some unique challenges as it involves controlling high-risk assets far from possible human intervention over extended periods of time. A spacecraft has numerous systems that may be running concurrently each with different operating constraints. Each plan comprises hundreds of activities and must ensure appropriate use of critical resources such as memory or energy. These plans, which are developed pre-launch, are iteratively updated with new model information and are intended for use in ground operations throughout the entire mission (Chien et al. 2021).

The Advanced Multi-Mission Operations System (AMMOS) is a NASA program managed by the Multi-mission Ground Systems and Services Program Office. Its objective is to provide on-the-shelf software for Ground Data Systems (GDS). Table 1 (Basilio and Di Pasquale 2017) shows the part of AMMOS GDS software and hardware in 4 NASA missions. It can be seen that AMMOS is a sizeable part of the ground data systems of these missions.

Among AMMOS current developments, AERIE will be the next-generation multi-mission activity planning and simulation software framework. Its Merlin and Falcon components are customizable for mission-specific needs and will replace multiple legacy tools currently in use. Among these, Merlin is the *activity planning* component which will replace APGEN (Maldague et al. 2014; 1998) in AMMOS. The objective of the present work is to develop a scheduling prototype for Merlin. Falcon is the sequencing tool.

Project	Cost in GDS (M$)	from AMMOS
Mars Exploration Rover	11.3	68%
Mars Reconnaissance Orbiter	9.7	97%
Spitzer Space Telescope	12.0	71%
Dawn	9.2	95%

Table 1: Part of AMMOS GDS software and hardware in several NASA missions. From (Basilio and Di Pasquale 2017) .

The other context of this research is the Europa Clipper mission,a NASA interplanetary mission whose goal is to study Jupiter's moon, Europa, through a series of 44 close flybys. It is scheduled to launch in October 2024 and arrive in the Jovian system in April 2030. The spacecraft will carry 9 instruments to study Europa's interior, ocean, geology, chemistry, and habitability (Bayer et al. 2019), each conducting multiple coordinated science campaigns. Science planning for such a complex mission presents numerous operational constraints on the spacecraft in order to achieve measurement requirements on observing geometry, target illumination, spacecraft pointing, instrument mode, and observation timing/cadence (Pinover et al. 2020). In order to ensure coverage of the concrete details of realistic mission scheduling use cases, the scheduling prototype was undertaken within a cycle of direct user feedback sessions with engineering and science planners from the Europa Clipper team.

This paper focuses on the modeling language and user interface that has been developed as elements of this multi-mission scheduler. In the remainder of this paper, we (1) present a new domain specific language for expressing science scheduling goals embedded within the Java programming language, (2) describe the development of a user interface companion to express the same concepts in a graphical manner, and (3) describe the scheduling approach that is currently in development. For each of these subjects, we also describe the directions of ongoing development.

Expressions of Goals

Mission and science planners must be able to express the goals, constraints, and preferences surrounding the activities to be scheduled. Example 1.1 is a realistic example of an activity scheduling goal for a plasma instrument that will help us define some important terms. The activity being scheduled here is a *Plasma Instrument Calibration Roll*. We will use the term **activity type** to describe its unique description concept and **activity instance** to describe its multiple instantiations within a plan. Item (1) of the example goal states that one instance of this activity must be scheduled for each *encounter*, which refers to each interval over which the spacecraft approaches, flies over, and departs from one of the moons of Jupiter (Callisto, Ganymede, or Europa). The nominal trajectory for Europa Clipper takes the spacecraft past each moon multiple times. Item (2) and (3) add **constraints on state variables**. State variables are time-varying values of different types (such as boolean, float, integer, or enumerated) that may represent such system facts as the spacecraft's altitude, the battery energy level, a data volume, or being eclipsed by Jupiter. Constraints on state variables restrict the times at which each activity can be scheduled. Item (4) adds a **mutual exclusion** constraint with other activities using pointing capabilities, forbidding concurrent execution. Items (5)-(8) define **subgoals** needed to achieve this goal. Item (8)(a) expresses a **preference** on the choice of the roll attitude, defining a small minimization problem. Item (9) expresses the possible **mutualization** of activities for satisfying several goals, stating that magnetometer calibration rolls with a certain parameterization can count for plasma instrument calibration rolls. If the scheduler is able to take advantage of this, it can reduce the total number of calibration activities in the plan and allow more time for science activities.

Example 1.1: Plasma Instrument Calibration Roll

Schedule *Plasma Instrument Calibration Roll* activity

(1) once for each encounter
  (a) except when the target is Ganymede
(2) when distance to Jupiter $< 8\times$ radius of Jupiter
(3) when target altitude $> 1.5 \times 10^8$m
(4) when not doing another activity that uses pointing
(5) roll about X-axis $\geq 100°$ within $<$ 2hr
(6) put Plasma Instrument in calibration mode
  (a) and back to survey after
(7) with parking of solar arrays at 25° during roll
  (a) and back to Sun-tracking after
(8) with initial slew to roll attitude
  (a) picked to minimize cryocooler exposure
  (b) and slew back to Earth-point after
(9) *Magnetometer Calibration Rolls* count for *Plasma Instrument Calibration Rolls* but only if
  (a) rolling about X-axis
  (b) aligned to Plasma Instrument exposure
  (c) solar arrays are parked
  (d) plasma instrument is in Calibrate mode
  (e) meets other geometric criteria above

(Numeric quantities have been sanitized.)

Note that this mixed declarative/imperative goal expression has been extracted from and mirrors a purely procedural APGEN scheduler specification that expanded explicitly to all necessary sub-activities (but with more arduous modularization and opportunity analysis).

A Java-embedded textual language has been designed to efficiently express such scheduling goals and constraints encountered frequently in the Europa Clipper mission domain. In the following subsections, we will describe the building blocks of this language and how they can be combined to form complex expressions.

Activity types

There is a distinction between an *activity type* and an *activity instance*. An activity type consists of a name and a list of named parameters. An activity instance has a type and concrete values for all of the parameters of its type. Most of the time, the objective of the language described here is to provide a compact way to specify how multiple activity instances of the same activity type should be scheduled under some constraints.

Temporal expressions

One of the most basic constructs used in the language is the *temporal expression*. Temporal expressions are sets of non-overlapping time intervals. In a temporal expression, any two intervals can share a bound (upper bound for one, lower bound for the other). There are several ways for expressing temporal expressions, whether tied to activities already present in the schedule, conditioned on state values, or a combination of both. Providing sufficient expressivity and modularity for temporal expressions is central to enabling users to communicate when activities must be scheduled.

Activity expressions An activity expression allows users to reference the time intervals during which already-present activities are scheduled. Each interval of the resulting temporal expression is made of start and end times of activities. A simple activity expression is shown in Listing 1. At the point it is evaluated, the expression will return the time intervals when any scheduled activity exists with type equal to `RollActivityType` and a parameter named `angle` equal to 270.

```
ActivityExpression rollActExpr = new
     ActivityExpression.Builder()
    .ofType("RollActivityType")
    .withParameter("angle", 270)
    .build();
```

Listing 1: Example activity expression.

It is also possible to combine simple statements in conjunctions or disjunctions such as in Listing 2.

```
ActivityExpression actExprDisjunction = new
     ActivityExpression.OrBuilder()
    .or( new ActivityExpression.Builder()
        .ofType( "RollActivityType" )
        .build())
    .or( new ActivityExpression.Builder()
        .ofType( "SlewActivityType" )
        .withParameter( "axis", BodyAxis.X )
        .build())
    .build();
```

Listing 2: Example disjunctive activity expression.

Note that the *disjunction* of two expressions whose co-domain is a temporal expression is the *union* of the two temporal expressions. Similarly, the *conjunction* of these two expressions is the *intersection* of the co-domains. These expressions cannot be statically evaluated because they depend on the current state of the schedule, and are thus evaluated dynamically during scheduling.

State constraint expressions A state constraint expression allows users to specify the time intervals during which a state variable satisfies a condition on its value. For example, the conjunction of 3 elementary state expressions shown in Listing 3 models all the intervals during which the encounter phase state is equal to `Encounter`, the altitude state is between 20 and 50km, and the altitude derivative state is greater than 0.0km/s. Operators include, but are not limited to: equal, between, above, and below. State constraint expressions can be used in scheduling goals, but can also be attached directly to an activity type to confer that all their instantiations are constrained by it.

```
StateConstraintExpression approachStateConstraint =
     new StateConstraintExpression.Builder()
    .andBuilder()
    .equal(encounterPhase, OrbitPhasesEnum.ENCOUNTER)
    .between(altitude, 20, 50)
    .above(altitudeDerivative, 0.0)
    .build();
```

Listing 3: Example conjunctive state constraint expression.

Composite temporal expressions Composite temporal expressions are a way of combining activity, state constraint, and other composite temporal expressions together through disjunction or conjunction. Listing 4 shows the conjunction of the activity expression defined in Listing 2 and of the state constraint defined in Listing 3.

```
TimeRangeExpression combined = new TimeRangeExpression
    .Builder().andBuilder()
    .from(approachStateConstraint)
    .from(actExprDisjunction)
    .build();
```

Listing 4: Example composite temporal expression.

A variant of composite temporal expression allows conditioning on the transitions (or lack thereof) in a state's value, as shown in Listing 5. If `downlinkPassWindows` represents all the available communication intervals and the `dlRate` state represents the available bandwidth that changes within each such pass, the combined expression would model all intervals of each constant downlink rate within the downlink windows. A rate-switching activity could then be scheduled for each derived interval.

```
TimeRangeExpression dlRates = new TimeRangeExpression
    .Builder()
    .from(downlinkPassWindows)
    .ofEachValue(dlRate)
    .build();
```

Listing 5: Example constant-value composite expression.

Composite temporal expressions also allow users to apply filter and transform operations on time windows from other expressions. Filters selectively retain only time windows meeting some criteria while transforms modify each window by expansion, contraction, or shifting manipulations. Listing 6 shows how to apply a minimum duration filter of 20 seconds to exclude tiny windows, followed by a transform to extend each remaining interval by 1 second at both ends as a margin strategy.

```
TimeRangeExpression filterExpressions = new
     TimeRangeExpression.Builder()
    .from( downlinkPassWindows )
    .thenFilter(Filters.minDuration( ofSeconds(20) ))
    .thenTransform(Transforms.extend(
      ofSeconds(1), ofSeconds(1) ))
    .build();
```

Listing 6: Example filter / transform expression.

Possible operations may be stateless as seen in Listing 6, but may also be stateful and context-dependent as in Listing 7 and Figure 1, which shows a variation of the previous filter. In this case, an encounter is a long period in which there are many downlink windows. For each encounter, we want to apply an initially more aggressive filter until the first matching downlink window is found, and then loosen the filter for all subsequent downlinks in the same encounter. This construction forms a *latching filter* object that can be then used the same as a regular filter.

```
TimeWindowsFilter dlDurationFilter = new Filters.
     LatchingBuilder()
    .withinEach( encounter )
    .filterFirstBy( Filters.minDuration( ofSeconds(20)
      ) )
    .thenFilterBy( Filters.minDuration( ofSeconds(10)
     ) )
    .build();
```

Listing 7: Example latching filter.

Time expressions Individual time points can be absolute fixed temporal values, but may also be defined relatively. The basic `START` and `END` *time anchors* specify

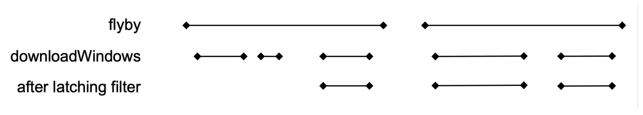

Figure 1: Example showing the application of a latching duration filter on download windows with a flyby as a reference period.

a single time point relative to the start or end of an interval, and may be extended with simple arithmetic. Listing 8 shows how a time point can be computed from one anchor and two durations. Note that this expression construct is not attached to any particular interval yet; it is functional and can be used to define relative start and end times of activities during goal definition.

```
TimeExpression actStart = new TimeExpression.Builder()
    .from( TimeAnchor.START )
    .minus( warmupDuration )
    .minus( startingDuration )
    .build();
```

Listing 8: Example relative time expression

Goals

Goals are expressions focusing on scheduling a particular consistent set of activity instances. They may be only partially achievable. In our setting, many users create their goals separately with a limited view of the overall set of goals and they are all aggregated to be solved together. A notional goal is made of

- a name,
- a scheduling priority which reflects its scheduling order relative to other goals,
- a temporal scope during which the goal is applicable, which may be different from the overall scheduling horizon. In other words, there will be no attempt to satisfy the goal outside of its temporal scope.
- an activity expression, describing the type of activity whose instantiates may satisfy the goal, along with any specific parameter values for these instances
- an optional state constraint expression constraining the periods during which it is possible to schedule the activity instances

In the next paragraphs we will see that several flavors of goals exist.

Coexistence goal A coexistence goal is a type of goal in which activity instances are scheduled with respect to an existing set of time periods, be those instances of another activity type or a conjunction of state values. As this goal is dependent/relative on other conditions or activities, it is important to mind the order in which it will be processed during the scheduler operation.

Listing 9 gives an example of the syntax of a coexistence goal that requests one roll activity (previously shown) for each interval during which the approach state constraint is valid. In this example, start and end times are specified with regard to the start and end of each state constraint interval, as seen before with time expressions. The approach state constraint can be replaced with any temporal expression.

```
CoexistenceGoal goal = new CoexistenceGoal
    .Builder()
    .thereExists(rollActExpr)
    .forEach(approachStateConstraint)
    .startAt(TimeAnchor.Start)
    .endsAt(TimeAnchor.End)
    .named("calibrationactitivitygoal")
    .withPriority( 7.0 )
    .build();
```

Listing 9: Example coexistence goal

While it is possible to define start and end times as well as durations in the goals, some activities have their own duration model which may depend on internal physical constraints. In this case, the user would not specify all of the temporal parameters.

Frequency goal A frequency goal expresses the need for scheduling activities at a regular cadence, i.e. a set of activities separated by a specific range of durations. Listing 10 shows a frequency goal requesting one calibration activity every two hours for all the time defined by the absolute `timeperiod` interval. This example also includes states constraints restricting the periods during which it is possible to schedule activities. Such constraints might make the strict frequency goal unachievable if there are no permissible windows aligned to the proper frequency.

```
FrequencyGoal goal = new FrequencyGoal
    .Builder()
    .thereExists(calibrationActType)
    .every(Duration.ofHours(2))
    .forAllTimeIn(timeperiod)
    .attachStateConstraint(approachStateConstraint)
    .named("FrequencyGoalExample")
    .withPriority( 8.0 )
    .build();
```

Listing 10: Example frequency goal

Duration goal A Duration goal expresses the need for scheduling a number of activities whose total duration lies in a range of durations. Listing 11 shows an example of such a goal in which 3 to 4 calibration activities must be scheduled for a total duration of 15 to 17 hours. Note that the duration of activity instances may not be directly controllable and instead defined within the activity type model, which may make the goal unachievable.

```
DurationGoal goal = new DurationGoal
    .Builder()
    .thereExists(calibrationActType)
    .totalDuration(Range.of(.ofHours(15), .ofHours(17)
    )
    .cardinal(Range.of(3,4))
    .forAllTimeIn(timePeriod)
    .named("DurationGoalExample")
    .withPriority( 9.0 )
    .build();
```

Listing 11: Example duration goal

Procedural goals Some specific activities may have their own ineffable scheduling logic and either (1) provide the necessary activity instances directly as an input to our scheduler after generation within some external software or process or (2) design their own domain scheduler in an imperative manner to be integrated in the mission model. For integration with these possibilities, a procedural goal may be defined that accepts some user-specified black-box function that takes the current schedule and returns any requested new activity instances. The function is then executed at the appropriate priority-ordered step in the scheduling process, and the scheduler will attempt to insert any returned instances into the growing plan. Listing 12 shows a procedural goal used to provide raw activity instances to the scheduler. The function is reduced to the minimum, providing a list of hard-coded activity instances and ignores the current schedule argument.

```
var activities = java.util.List.of( act1, act2 );
Function<Plan, Collection<ActivityInstance>> generator
    = (schedule) -> activities;

var proceduralGoal = new ProceduralGoal.Builder()
    .forAllTimeIn( timePeriod )
    .generateWith( generator )
    .named( "ProceduralGoalExample" )
    .withPriority( 7.0 )
    .build();
```

Listing 12: Example procedural goal

Composite goals A composite goal is a conjunction or disjunction of other goals. The conjunctive composite requires satisfaction of each subgoal while the disjunctive only requires one of its subgoals be satisfied. Listing 13 shows how to form a goal from the conjunction of three subgoals. While the composite goal has its own priority to order it within the wider scheduling process, the subgoals are ordered according to their relative priority within the composite. This construct opens up the specification of task decomposition trees, similar to Hierarchical Task Networks, with conjunctions of subgoals being analogous to *subtasks* and disjunctions being analogous to *methods*.

```
var downlinkGoal = new CompositeAndGoal.Builder()
    .forAllTimeIn(planningHorizon)
    .and( beforeDownlink )
    .and( duringDownlinkGoal )
    .and( afterDownlinkGoal )
    .named( "CompositeGoalExample" )
    .withPriority( 7.0 )
    .build();
```

Listing 13: Example composite goal

Global constraints

A constraint is said to be global when it is not attached to a specific goal or its activity instances, but rather to the whole scheduling problem, meaning it must be enforced every time a change is made to the schedule in service of any goal. A global constraint may be set on resources to prevent oversubscription (e.g. energy or memory) or on activity types such as to prevent parallel execution of certain activity types (a mutual exclusion constraint). Our language only models these two types of global constraints for now. Listing 14 shows an example of mutex constraint disallowing the parallel execution of three activity types.

```
 var exampleMutexConstraint = GlobalConstraint.
    atMostOneOf(List.of(
        ActivityExpression.ofType( actType1 ),
        ActivityExpression.ofType( actType2 ),
        ActivityExpression.ofType( actType3 )
    ));
```

Listing 14: Example n-ary mutex constraint

Preferences

Preferences (or soft constraints) allow the user to define what should happen when a goal can be satisfied in more than one way; in other words, how the scheduler should behave when there is flexibility. For example, there may be several ways of satisfying the duration goal defined in Listing 11: scheduling 3 activities lasting 5 hours each or scheduling 4 activities lasting 4 hours each, etc. One preference in this case may be be to minimize the total number of activities, trying to schedule 3 activities rather than 4. It can also consist in trying to schedule activities with similar durations.

The only preference that can be specified by the user in the current version of the language concerns the satisfaction of composite goals. Listing 15 shows the definition of a conjunction composite goal in which the second goal is made of a disjunction goal with a preference stating that the preferred subgoal is the one in which the scheduled activity instances start at the latest possible time.

```
var downlinkGoal = new CompositeAndGoal.Builder()
    .forAllTimeIn(timePeriod)
    .and( downlinkGoal )
    .and( new OptionGoal.Builder()
        .exactlyOneOf()
        .or( slowSetupDownlinkGoal )
        .or( fastSetupDownlinkGoal )
        .optimizingFor( Optimizers.latestStartTime() )
        .build())
    .build()
```

Listing 15: Example preference

It might be difficult to express scheduling preferences as they may depend on internalized unconscious domain-dependant knowledge. In practice, it is only when the user is confronted with a concrete schedule that they can point to what is *not* right, and then modify the handles at their disposal to steer the results in the desired direction. As a result, defining what kind of preferences will be included in the language is an ongoing user-focused research topic.

Designing a User Interface

To extend access of the scheduling software to users who are less familiar with programming in Java, a companion user-interface (UI) was designed in parallel with the domain-specific language. Through visual styling and patterns of interaction, the UI aims to provide users with guardrails for authoring expressions and goals that effectively achieve their intended scheduling outcome. Our design proposes that the companion UI and the code editor be presented side-by-side in the application and simultaneously updated in real-time, regardless of which interface is used, so that users can see how the UI translates into code and vice versa.

Constraint authoring in a user-interface has been implemented in other planning software such as Science Opportunity Analyzer (SOA) (Llopis et al. 2019), a science planning tool used by the DAWN and PSYCHE missions. In SOA's opportunity search query builder, users work in a UI to connect boxes together to specify the geometric conditions for which they would like to schedule an observation. To specify the details of a geometric constraint such as a distance constraint, users select the constraint box to pull up a contextual panel to input the details of their constraint, such as distance from the target body. The benefit of SOA's constraint authoring UI is that it leverages visual design to scaffold the authoring experience by separating the specification of the constraint type in one window and the contextual details panel in another. This design simplifies the editing process as it allows users to quickly find and make changes to the parameters of a selected constraint.

To create a UI for Europa Clipper's scheduling syntax we designed an interface that would allow users to specify an opportunity for an observation and how they would like one or more activities to occur within that opportunity window. To do this we categorized the expressions into four different categories and created a design system that uses patterns of interaction and visual styling to help the user author achieve their scheduling goals. Expression types are represented as boxes that can be arranged in relation to other boxes and connected through lines.

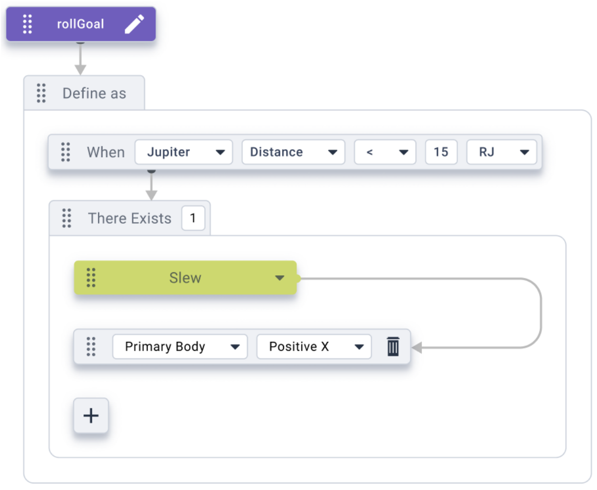

Figure 2: Example scheduling goal defined in UI

UI Components and patterns of interaction

The following section describes the different components of the UI syntax, broken into four categories; (1) Variable blocks, (2) Reference blocks, (3) Scheduling Opportunities, and (4) Scheduling Specifications.

Variable Blocks

Goals, windows, and constraint expression composites are designed as Variable blocks. Variable blocks allow users to create a new goal or window by clicking directly in the box to give it a new name and connecting it to a Definition block.

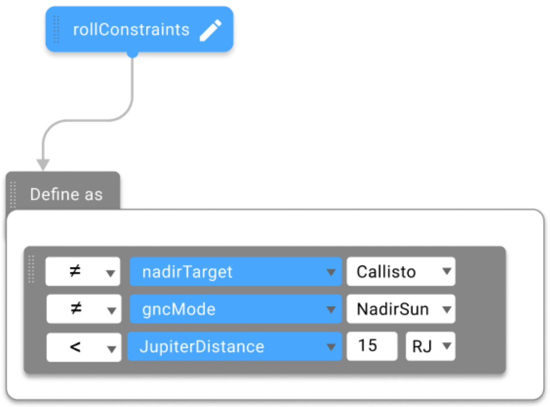

Figure 3: Constraint Composite block including of three different constraints

Reference Blocks

Activity types, activity parameters, and states are designed as Reference Blocks. Each reference box allows the user to pick from predefined lists that represent the activity types, activity parameters, and state detailed in the adaptation.

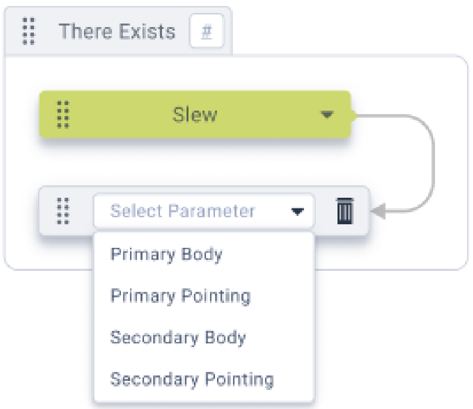

Figure 4: Activity parameters are available through a drop-down menu in an Activity Parameter block connected to an Activity block

Scheduling Opportunities

Definition blocks are connected to a Variable block to allow users to define the conditions for which they would like to schedule an activity. Multiple Definition blocks can be used in one scheduling goal if it contains multiple Variable blocks. The following blocks can be placed within a Definition block to define a scheduling opportunity.

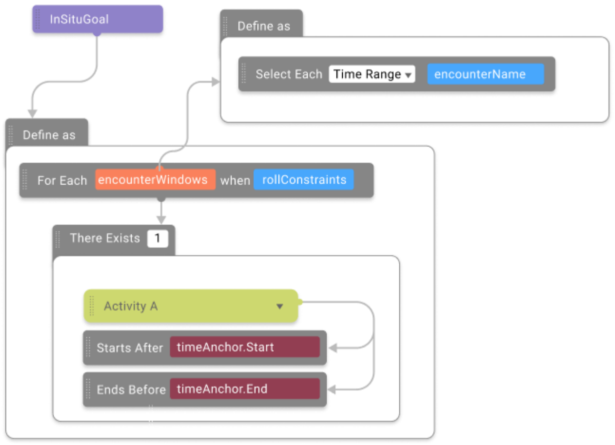

Figure 5: Activity parameters are available through a drop-down menu in an Activity Parameter block connected to an Activity block

Constraint Expression blocks including Geometric, Temporal, State, Activity Relational, Epoch, and Duration can be placed into a definition block to define a scheduling opportunity goal. Each constraint type is designed such that the inputs for the specified constraint are exposed to the user.

Composite blocks allow users to combine two or more state and or constraint values, or goals.

Transform blocks allow users to transform Time Range Expressions (referred in the UI as ”windows”) by filtering, contracting, or subtracting times. This can be applied to an entire window or unique values within that window.

Scheduling Specifications

Frequency Constraint blocks allow users to specify the number of activity instances that must occur during a defined scheduling window. Activity parameters blocks and Activity Attribute blocks may be connected to activity type blocks to further specify how an activity should be scheduled into the timeline.

Future work in UI

Future development of the companion UI will focus on developing a UI such that it can translate and produce the code syntax in real time. Additional design work includes developing blocks for Mutual Exclusion Constraints and Preferences. During user testing, test subjects requested a ”toolbox” to help locate block types easily, so an information architecture for organizing block types will also be designed and tested. Additional design work beyond the above mentioned, will be informed by user testing and development of the syntax.

Scheduling

The scheduling algorithm is a constructive algorithm in which goals are scheduled in priority-first order. To schedule a given goal, valid constraint intervals are computed and the current schedule is examined to look for activities already present and satisfying the goal. If this is not the case, the goal will generate conflicts. Conflicts can be resolved with different possible resolutions, based on heuristically chosen start times for activities when several are available. This way of allocating responsibilities between scheduler and goal allows for more flexibility in terms of scheduling algorithms. Such a simple approach has its drawbacks in terms of resulting schedule quality but it has advantages in our setting. Priorities are an easy way of expressing preferences but in practice, it is usually difficult to discriminate between scientific activities. Scheduling priority are rather an easily understandable handle for tweaking the results when mission/-science planners are not satisfied with the resulting schedule.

Future work in scheduling

The scheduling aspects of this work are still in their early stage as the focus has been given to developing the language for expressing goals. There are several paths are envisioned with regard to scheduling.

Other search strategies Because of the current greedy scheduling strategy and the fact that goals are totally ordered, the explored search space is reduced to a minimum, decision-making being necessary only when temporal flexibility exists for activities. This might prevent the achievements of low-priority goals because of early commitment for goals higher in the priority list. An approach worth exploring is reasoning over *bins of priorities* that would contain several goals, thus allowing for more exploration while keeping dimensionality low.

Explainability Interpreting the results of a scheduler processing hundreds of states on long scheduling horizons is a challenge for the mission planners. Our goal is to provide explainability features to mission planners and operators. One good recent example is in the context of the task scheduling for Perseverance, the Mars 2020 rover. Crosscheck (Agrawal, Yelamanchili, and Chien 2020) has been developed to provide explanations as to why some activities failed to be scheduled by analyzing constraint intervals and resource consumption. We expect that this development will be user-focused, as to provide explanations to most frequently asked questions about the produced schedules. For example, these will include explanations about why a goal has failed to be satisfied but also about the choices leading to the current start time of scheduled activities or the usage history of a resource.

Mixed-initiative scheduling The current scheduling prototype is intended to be used in a one-shot process. After running the scheduler the first time, if the user wants to look at a different schedule resulting from minor changes to the input data, they must re-run the scheduler. This process could be described as the following sequence (1) Run scheduler, (2) Visualize schedule, (3) Change input data, (4) Rerun scheduler, (5) Visualize results. As the prototype cannot process previous plans, it has to recompute a schedule from scratch which in the average case takes more time than making changes to the current schedule and might completely change the resulting schedule.

One objective is to give the user the ability to interact with the scheduler (ideally through a graphical user interface) in a way that would minimize the (a) running time and (b) instability from one iteration to another. The goal is to arrive at the following sequence: (1) Run scheduler, (2) Visualize schedule (3) Change data and visualize the minimally local impact immediately, (4) Accept or reject the schedule modification.

To achieve this objective requires designing an graphical user interface, considering local search algorithms, and implementing efficient dependency analysis in the model to avoid costly recomputation.

Related Work

There exists several languages built for specifying planning problems such as the Planning Domain Definition Language (Gerevini and Long 2005), the New Domain Definition Language (NDDL) associated with the EUROPA planner (Barreiro et al. 2012), the Action Notation Modeling Language ANML (Smith, Frank, and Cushing 2008), or Linear Temporal Logic (LTL) and its more recent scheduling extension (Luo et al. 2016). While their expressive power might make it possible to express scheduling problems (most handle resources and some activity decomposition) if needed, they are not designed for this purpose. Their planning-centric approach builds the modeling around the concepts of actions with their preconditions and postconditions and revolves around finding sequences of actions leading to a desired state. They do not explicitly provide operators for inserting activities at a specific frequency, with applicability periods, or coexistence conditions. In our scheduling setting, the goal is to provide a practical way for the mission and science planners to express mostly *activity insertion* goals in a condensed manner, possibly with preferences.

Past work in a setting closer to pure scheduling such as ours can be found in APGEN (Maldague et al. 2014; 1998), in the adaptation of ASPEN for the ROSETTA mission (Chien et al. 2021), in the scheduling languages designed for staff scheduling problem (Gärtner et al. 2011), for scheduling space operations at the Beijing Aerospace Control Center (Xing et al. 2016), or in coverage scheduling with CLASP (Doubleday 2016; Yelamanchili et al. 2019; Maillard, Chien, and Wells 2021) in which polygons associated with coverage constraints constitutes a specific form of campaign. With regard to the modeling language, this previous work usually provide an imperative language to the user. It results in goals being a collections of algorithms. While a purely declarative approach is not possible in our setting for operational aspects, we try to tend in this direction to increase modularity and readability of goals. We believe that Java-embedding, functional aspects of the syntax, and a pleasant user interface are also a way of improving the user experience during goal authoring. With regard to scheduling aspects, while it would probably be possible to translate the goals into a linear programming or constraint satisfaction formalism, one of the main objective here is to provide explainability features to the user which makes any black-box solvers unsuited as decision-making needs to be traced. Also, the particular aspects mixing discrete and continuous aspects, that can only be simulated, because noninversible, sometimes (e.g. attitude planning), and the large size of problem instances, makes us lean towards scheduling approaches that favor early instantiation/grounding and fast computation such as greedy approaches rather than least commitment approaches.

Conclusion

In this paper, we have presented a Java-embedded domain-specific language designed to be used by mission and science planners to author scheduling goals for the Europa Clipper mission. This language has features inspired both by imperative languages and propositional logic in an effort towards the definition of more declarative scheduling goals. A companion user-interface has been developed and several user sessions with mission and science planners have demonstrated that having both best accommodates user preferences and knowledge. A primary scheduling strategy has been presented and several future directions for development have been outlined, including mixed-initiative scheduling and explainability.

References

Agrawal, J.; Yelamanchili, A.; and Chien, S. 2020. Using explainable scheduling for the mars 2020 rover mission. In *ICAPS 2020 Workshop of Explainable AI Planning (XAIP)*.

Barreiro, J.; Boyce, M.; Do, M.; Frank, J.; Iatauro, M.; Kichkaylo, T.; Morris, P.; Ong, J.; Remolina, E.; Smith, T.; et al. 2012. Europa: A platform for ai planning, scheduling, constraint programming, and optimization. *4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*.

Basilio, E., and Di Pasquale, P. 2017. The advance multi-mission operations system (ammos)-an introduction to the multi-mission products and services used by 50+ nasa missions. In *Interplanetary Small Satellite Conference*.

Bayer, T.; Bittner, M.; Buffington, B.; Dubos, G.; Ferguson, E.; Harris, I.; Jackson, M.; Lee, G.; Lewis, K.; Kastner, J.; Morillo, R.; Perez, R.; Salami, M.; Signorelli, J.; Sindiy, O.; Smith, B.; Soriano, M.; Kirby, K.; and Laslo, N. 2019. Europa clipper mission: Preliminary design report. In *2019 IEEE Aerospace Conference*, 1–24.

Chien, S. A.; Rabideau, G.; Tran, D.; Troesch, M.; Doubleday, J.; Nespoli, F.; Ayucar, M. P.; Sitja, M. C.; Vallat, C.; Geiger, B.; Vallejo, F.; Andres, R.; Altobelly, N.; and Kueppers, M. 2021. Activity-based scheduling of science campaigns for the rosetta orbiter. *Journal of Aerospace Information Systems* In Press.

Doubleday, J. R. 2016. Three petabytes or bust: Planning science observations for nisar. In *SPIE 9881*.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in pddl3. Technical report, Technical Report 2005-08-07, Department of Electronics for Automation . . . .

Gärtner, J.; Musliu, N.; Schafhauser, W.; and Slany, W. 2011. Temple - a domain specific language for modeling and solving staff scheduling problems. In *2011 IEEE Symposium on Computational Intelligence in Scheduling (SCIS)*, 58–64.

Llopis, M.; Polanskey, C. A.; Lawler, C. R.; and Ortega, C. 2019. 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)*, 1–8. IEEE.

Luo, R.; Valenzano, R.; Li, Y.; Beck, J. C.; and McIlraith, S. A. 2016. Using metric temporal logic to specify scheduling problems. In *Proceedings of the Fifteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'16, 581–584. AAAI Press.

Maillard, A.; Chien, S. A.; and Wells, C. 2021. Planning the coverage of planets under geometrical constraints. *Journal of Aerospace Information Systems* 18:5:289–306.

Maldague, P.; Ko, A.; Page, D.; and Starbird, T. 1998. Apgen: A multi-mission semi-automated planning tool. In *First international NASA workshop on planning and scheduling*, 363–365.

Maldague, P. F.; Wissler, S. S.; Lenda, M. D.; and Finnerty, D. F. 2014. Apgen scheduling: 15 years of experience in planning automation. In *SpaceOps 2014 Conference*, 1809.

Pinover, K.; Ferguson, E.; Bindschadler, D.; and Schimmels, K. 2020. The reference activity plan: Collaborative, agile planning for nasa's europa clipper mission. In *2020 IEEE Aerospace Conference*, 1–13.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The anml language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 31.

Xing, J.; Zhu, H.; Li, L.; and Zou, X. 2016. The unified scheduling language designed for the space mission scheduling platform. In *2016 IEEE Aerospace Conference*, 1–7.

Yelamanchili, A.; Chien, S.; Moy, A.; Shao, E.; Trowbridge, M.; Cawse-Nicholson, K.; Padams, J.; and Freeborn, D. 2019. Automated science scheduling for the ecostress mission. In *International Workshop for Planning and Scheduling for Space (IWPSS 2019)*, 204–211. Also appears at ICAPS SPARK 2019.

Acknowledgments

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004). The cost information contained in this document is of a budgetary and planning nature and is intended for informational purposes only. It does not constitute a commitment on the part of JPL and/or Caltech.