

The Behavior, Constraint, and Scenario (BeCoS) Tool: A Web-Based Software Application for Modeling Behaviors and Scenarios

Justin D. Kaderka,¹ Matthew L. Rozek,² John K. Arballo,³ David A. Wagner,⁴ and Michel D. Ingham⁵
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 91109, USA

The Behavior, Constraint, and Scenario (BeCoS) tool has been developed to allow engineers to specify system and component behaviors. The tool is a web application that is developed in JavaScript and uses the React framework for the user interface and Redux for maintaining application state. The foundation of the tool is its underlying ontology, which expands upon a previously-defined behavior ontology with a scenario ontology. The behavior ontology includes elements like behaving elements, state variables, parameters, and constraints, while the scenario ontology includes core constructs like activities, temporal constraints, and timepoints. BeCoS allows users to easily create behaving elements and to specify their state variables, parameters, state machines, and constraints. BeCoS also allows users to develop temporal constraint networks that specify constraints on component states over time. BeCoS is a prototype tool that has been deployed and tested by systems engineers on the Europa Clipper project, which generated several use cases and helped steer its current developmental effort. By enabling systems engineers to specify behavior in a semantically-rigorous manner, BeCoS is an enabling technology for analyses that previously could not be performed, and when exporting its model to other tools, allows for consistent behavior models to be used.

I. Nomenclature

<i>APGEN</i>	=	Activity Plan Generator
<i>BeCoS</i>	=	Behavior, Constraint, and Scenario tool
<i>COTS</i>	=	Commercial-off-the-shelf
<i>FDD</i>	=	Functional Description Documents
<i>IMCE</i>	=	Integrated Model-Centric Engineering
<i>MBSE</i>	=	Model-Based Systems Engineering

II. Introduction

Behavior is a critical aspect of robotic spacecraft that engineers must characterize. Behaviors of individual components aggregate into the system behaviors needed to complete the mission. These characterizations of behavior, captured in the form of a behavioral system model, can be used for many different purposes, including the many analyses of the spacecraft system that are needed across the engineering lifecycle (e.g. power/energy or data analyses to name a few). Moreover, accurate and complete descriptions of behavior in the development phase of the lifecycle are needed so that engineers can develop and implement the flight software and hardware according to a specification.

In its simplest description, behavior is defined by state variable properties of a particular system component and the values that they have over time. Behavior can be classified into either intrinsic or scenario behavior. Intrinsic

¹ Systems Engineer, System Architectures and Behaviors Group, M/S 321-425P, AIAA Member.

² Systems Engineer, Flight System Systems Engineering Group, M/S 321-525E, AIAA Member.

³ Scientific Applications Software Engineer, Structure of the Universe Group, M/S 169-327.

⁴ Group Supervisor, System Architectures and Behaviors Group, M/S 301-445P, AIAA Senior Member.

⁵ Europa Clipper Project Software Systems Engineer, Project Software and Information Systems Engineering Group, M/S 321-541, AIAA Associate Fellow.

behavior describes the inherent dynamics/physics of a component, that is, all the ways that its state can change under the influence of other component states and inputs. Prior work has represented intrinsic behavior in the form of “state effects models” [1]. One example of intrinsic behavior is power consumed by a lightbulb, which is calculated from the current and a voltage drop through the element. Scenario behavior, on the other hand, is a type of asserted, or control, behavior that captures a particular evolution of the component’s or system’s state over time, and can be represented either declaratively (goal-based) or imperatively (procedure-based). A declarative approach focuses on *what* needs to be accomplished whereas an imperative approach focuses on *how* it is accomplished. Traditionally in spacecraft design, scenario behavior has been imperatively defined, and indeed many simulation tools and programming languages support the expression of imperative logic to describe asserted behavior. However, this approach has become cumbersome with increasingly complex systems and it has the limitation that operational intent has to be inferred. A declarative, goal-based, expression of scenario behavior is preferable as it explicitly describes intent (through goals) without necessarily describing how it is accomplished, and ultimately produces a specification of behavior that is completely verifiable [2, 3].

To date, behaviors on NASA/JPL’s robotic spacecraft have been described by systems engineers primarily through Functional Description Documents (FDDs), in which behaviors are described textually or through informal visual representations without further semantic meaning. This has led to discrepancies in their descriptions and, as an example, has led to differences in interpretations between the systems engineers and flight software developers. However, the adoption of Model-Based Systems Engineering (MBSE) presents an opportunity to model behaviors and scenarios in a semantically-rigorous manner. Doing so early in the project lifecycle enables the systems engineer to perform analyses and to link behavior models directly to other systems engineering tools. For example, behavior captured in state machines could be exported to a third party execution tool to perform state reachability analyses, and behavior information including scenarios could be exported to a mission planning tool that elaborates and schedules activities and produces component state timelines that are ready to be executed.

However, modeling behaviors and scenarios in existing standard languages like SysML using commercial off-the-shelf (COTS) modeling tools is a tedious process, largely because these tools have so many features (most of which are not relevant to behavior modeling) that they are cumbersome for non-expert users. Developing behavior models using COTS SysML modeling tools results in many user mouse-clicks even for a simple behavior model. In addition, the semantics of SysML are not readily amenable to describing temporal constraint networks as one may want to do for scenarios. To address such limitations in standard languages and tooling, a web-based application – the Behavior, Constraint, and Scenario (BeCoS) tool – has been developed as a front-end for behavior modeling and specification. BeCoS is a currently a functioning prototype that has been tested by select systems engineers on NASA/JPL’s planned Europa Clipper mission.

III. Ontology

An ontology has been developed to sufficiently express the concepts within behavior modeling, and to provide an underlying schema for the BeCoS tool. This ontology expands upon a previously-defined behavior ontology [4], which describes the intrinsic behavior of components, and is described in Figures 1-3. Briefly, in its simplest form, behavior is described as a system of equations (referred to as behavior constraints) that relate state variables (variables whose value is time-varying, such as power consumed by the behaving element) and parameters (variables whose value is time invariant, such as resistance), and is shown in Figure 1. Each state variable and parameter is associated with a codomain, or value type (e.g. a state variable representing current would have an “electric current” value type). Components that have behavior are called behaving elements, and are associated with their own state variables, parameters, and element behaviors, which is a specialization of behavior constraint (Figure 2). In addition, a state variable can be linked to a state machine, which is a specialization of codomain, as might be the case for a Switch behaving element that can have an Open and Closed state (Figure 3). A state machine has one or more states (discrete state variable values), which are connected with transitions that can have triggers (a command or event that must happen for the transition to execute) and/or guards (constraints that must be true for the transition to execute).

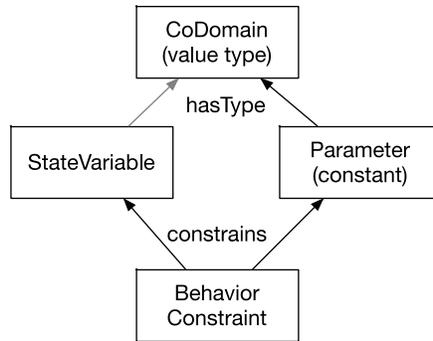


Fig. 1. Behavior is described as a system of equations, referred to as behavior constraints, that relate state variables (time-varying) and parameters (time invariant). State variables and parameters each are associated with the codomain, or value type.

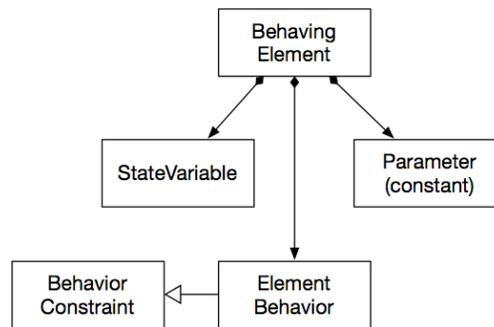


Fig. 2. Behaving elements are components that have behavior, and are associated with state variables, parameters, and element behavior (a type of behavior constraint).

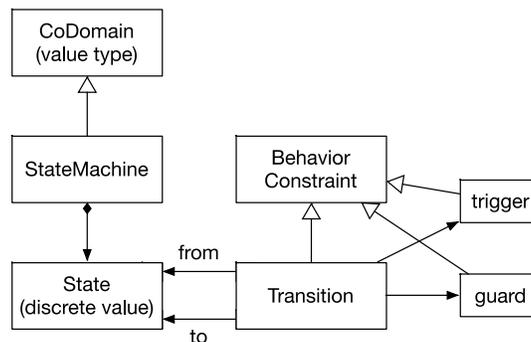


Fig. 3. A state machine is a specialization of codomain, and has discrete-value states. Transitions relate how states progress from one to another, and can be associated with triggers (commands or events that must happen for transition to occur) and guards (constraints that must be true for transition to occur).

A scenario ontology was developed to describe and specify coordination of behaviors over time and is grounded in a timeline representation [5]. The ontology that has been developed enables the user to express intent in a purely declarative form rather than in an imperative fashion (i.e. a step-by-step, procedural approach). That is, the goal of a declaratively-specified scenario is to constrain what and when something needs to happen without having to prescribe a strict procedure. A primary construct of this ontology is the schedulable behavior constraint (as shown in Figure 4), which is a behavior constraint applied to a specific interval of time. To accomplish this, the schedulable behavior constraint is also a specialization of temporal constraint, which is a constraint between two timepoints, and can specify either that the two timepoints occur at the same time (referred to as an “equals” temporal constraint) or that one timepoint precedes the other by an exact time or time range (referred to as a “precedes” temporal constraint). One example of a schedulable behavior constraint is “SwitchPosition = Closed”, where “SwitchPosition” is the state

variable associated with a state machine and “Closed” is one of its discrete-value states, and this constraint is asserted over a time interval specified by its associated temporal constraint. Finally, scenarios (Figure 5) are composed of schedulable behavior constraints and temporal constraints. Temporal constraints can describe the temporal relationship either between schedulable behavior constraints, or between an individual timepoint and a timepoint associated with a schedulable behavior constraint. Additionally, a scenario is defined for an instance of the system, which contains a set of behaving elements; thus, the schedulable behavior constraints relate instances of behaving elements’ state variables and parameters.

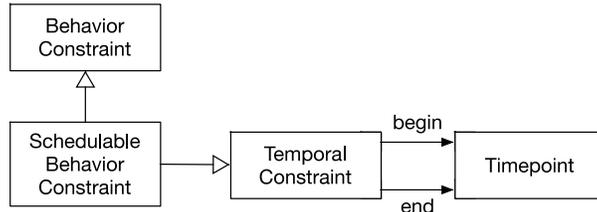


Fig. 4. A temporal constraint constrains the time between two timepoints, while a schedulable behavior constraint is a behavior constraint that applies over an interval of time.

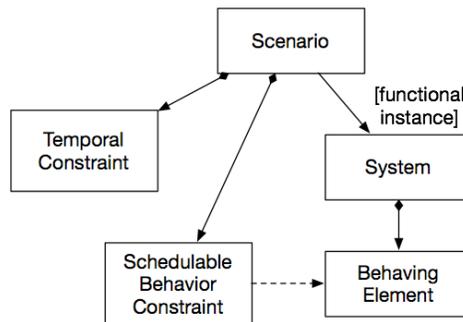


Fig. 5. Scenarios are composed of temporal constraints and schedulable behavior constraints, which constrain instances of behaving elements’ state variables and parameters.

This scenario ontology expresses assertions about behavior over time that can represent different semantics including engineering/control intent, observed history, or asserted conditions (e.g. fault conditions). Although BeCoS, which implements a behavior model adhering to the scenario ontology, does not directly perform any particular analysis, the specification is intended to support design/plan analyses that need to reason about the consistency of asserted intent in the context of intrinsic behavior (constraints that can’t be relaxed), plan objectives (that can be relaxed or changed), and other asserted facts such as fault conditions.

This ontology, taken together, contains all the constructs necessary for modeling behavior and scenarios. It is the meta-model used in BeCoS and in data exchange into and out of the tool. While the behavior ontology has been approved and is institutionally-supported, the scenario ontology described here has yet to be formalized, and it may be subject to modification during the approval process.

IV. Implementation

BeCoS is a web application that has been developed for behavior modeling. A guiding principle used during the app’s development has been to ensure models are “correct by construction”, which is enforced through validation checks throughout the app. A second guiding principle has been to present the user with only the relevant information for the scope of the current modeling task. The web app is a Node.js app⁶ that uses the React framework⁷, which is a popular framework for user interfaces and was chosen due to the developer’s familiarity with it. The Redux library⁸ is used in conjunction with React for application state management. A single, immutable, read-only state-tree is

⁶ <https://nodejs.org/>

⁷ <https://facebook.github.io/react/>

⁸ <http://redux.js.org/>

maintained; when users interact with the app, actions are dispatched to pure functions (i.e. reducers) that update the state-tree.

A JSON file⁹ is the data exchange format for model information, which uses a schema adhering to the ontology. The JSON file can be retrieved either from the local file system or a remote server database. Once a model is loaded into BeCoS, it persists and is manipulated entirely in the browser's memory until the model is again saved or exported as a JSON file.

Figure 6 shows the splash page for BeCoS tool. Four buttons (tabs) at the top of the app, named "Elements", "Interactions", "State Machines", and "Scenario", define the four sections of the app, which are used to build behavior models. The app is described in the following sections, and a simple example of a circuit is used to illustrate the app's capability. This example has four behaving elements: a battery, switch, lamp, and controller. In order to illuminate the lamp, the controller must be on and the switch must be closed.

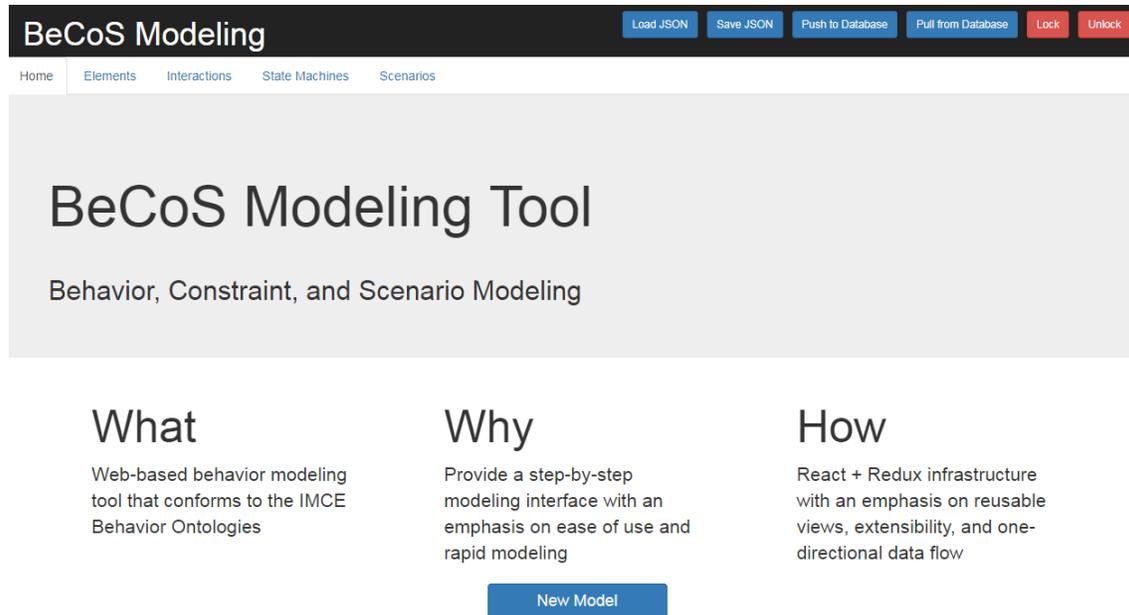


Fig. 6. Splash page of the BeCoS tool.

In the Elements tab, the user creates and/or edits behaving elements as well as the parameters and state variables associated with each. The Elements tab is shown in Figure 7 and displays a Lamp behaving element. As previously discussed, parameters, such as "Luminous Efficacy", have values that are time invariant, whereas state variables, such as "VoltageAcrossLamp", have values that vary with time. Attributes for each of these, such as descriptions, values, units, and symbols, are easily editable through a table that is created with the Bootstrap library¹⁰.

Behavior constraints, such as "Lumen Output", are also created in the Elements tab through a constraint editor (Figure 8). This editor is presented when the user selects the "Edit" button in the constraint table of the Elements tab. The constraint editor is scoped appropriately and presents only the state variables and parameters associated with the currently-selected behaving element, which is the Lamp in Figure 8. The user can easily create a symbolic expression by selecting the "+" icon, which adds the variable's symbol to the text box, or by typing the equation directly into the text box. The syntax of such If the user types a variable that is non-existent or is not associated with the selected behaving element, a validation error is issued and the user will not be able to save the constraint.

⁹ <http://www.json.org/>

¹⁰ <https://getbootstrap.com/>

BeCoS Modeling

Load JSON Save JSON Push to Database Pull from Database Lock Unlock

Home Elements Interactions State Machines Scenarios

New Behaving Element Delete Element

Behaving Elements

- ▼ Battery
- ▼ **Lamp**
- ▼ Switch
- ▼ Controller

Behaving Element: Lamp

Type: Component

name --	description --
Lamp	

Parameters

name --	description --	quantity kind --	value --	units --	symbol --	
Ohmic Resistance		electrical resistance	50	ohm	R	x
Luminous efficacy		one	4	one	eta	x

New

State Variables

name --	description --	quantity kind --	units --	symbol --	
Lumen Output		one		L	x
VoltageAcrossLamp		voltage		deltaV	x
CurrentThroughLamp		electric power		i	x

New

Global Element Constraints

name --	description --	expression	
Unnamed		$L = \text{delta}V \cdot i \cdot \text{eta}$	Edit x

Fig. 7. Behaving elements, as well as their state variables, parameters, and constraints, are defined in the BeCoS Elements tab. A Lamp behaving element is shown in this example model of a circuit.

BeCoS Modeling

Load JSON Save JSON Push to Database Pull from Database Lock Unlock

Behaving Element: Lamp

Edit Equation

Parameters

name --	description --	quantity kind --	value --	units --	symbol --	
Ohmic Resistance		electrical resistance	50	ohm	R	+
Luminous efficacy		one	4	one	eta	+

State Variables

name --	description --	quantity kind --	units --	symbol --	
Lumen Output		one		L	+
VoltageAcrossLamp		voltage		deltaV	+
CurrentThroughLamp		electric power		i	+

$L = \text{delta}V \cdot i \cdot \text{eta}$

Save
Close

Fig. 8. The constraint editor allows the user to create constraints with state variables and parameters scoped to the selected Behaving Element. A constraint relevant for the Lamp component is shown in this example model, relating the Lumen Output of the Lamp to the Voltage across the Lamp and Current through the Lamp.

The Interactions tab allows the user to specify the behavior of interacting components (Figure 9). Interactions are defined as behavior constraints that involve state variables and/or parameters from more than one behaving element. In the circuit example, Kirchoff’s voltage law would be one such interaction, which relates the voltage drops across the Battery, Switch, and Lamp. In the Interactions tab, the user selects all behaving elements that are involved with a specific interaction (called interacting elements in this context), and then creates one or more constraints using the constraint editor from Figure 8. However, in the context of Interactions, the editor is now scoped to include state variables and parameters associated with all interacting elements.

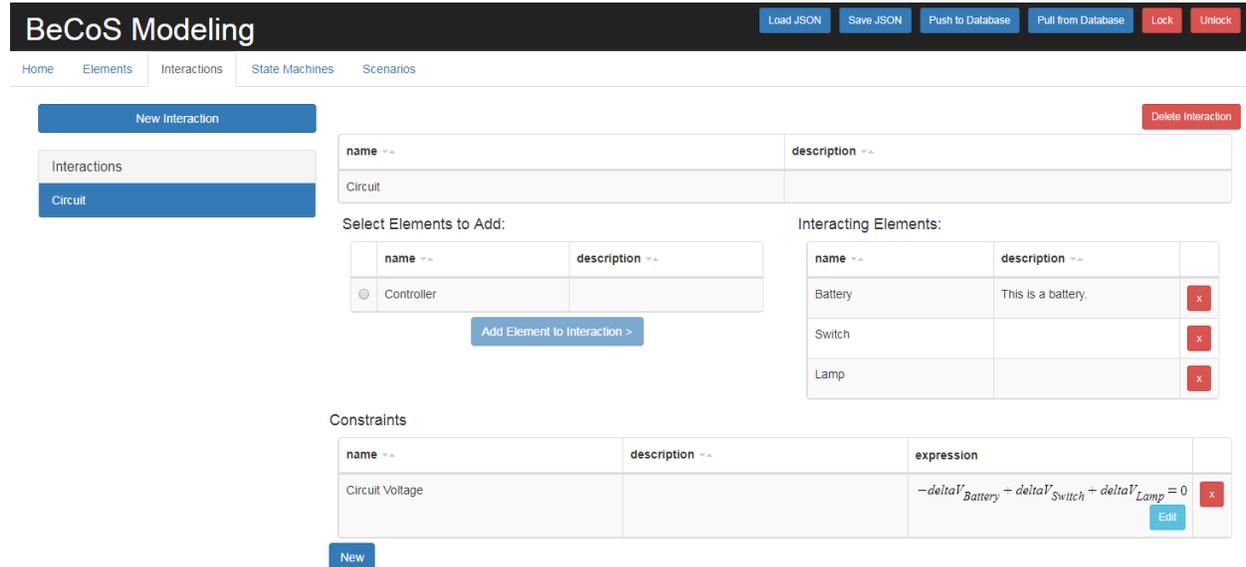


Fig. 9. The interaction tab allows creation of constraints with state variables and parameters scoped to the selected interacting elements. In this example a simple constraint has been created that relates the voltage drops across the Lamp, Switch, and Battery elements.

The State Machines tab (Figure 10) allows the user to define state machines for discreet-valued state variables, including states, transitions, constraints (element behaviors associated with states), triggers, and guards. In the circuit example, the Switch behaving element has a state variable called “SwitchPosition” that is associated with a state machine. It has the following behavior: a switch in its Closed state is characterized by the constraint “ $\delta V=0$ ”, and can transition to its Open state upon receipt of an “OPEN” command; a switch in its Open state is characterized by the constraint “ $i=0$ ”, and can transition to its Closed state upon receipt of a “CLOSE” command. The visual representation of state machines uses the D3 library¹¹ [11], so the user can graphically add, remove, and rename states, as well as specify transitions between states. The user can add other information to the state machine, such as element behavior, guards and triggers, by selecting the appropriate entity and using the Bootstrap table editor below the D3 visualization. Once defined, constraints associated with a state will appear in the state icon, while triggers and guards will appear on their respective transitions using the “Trigger [{Guard}]” syntax. Multiple constraints on a given state and guards on a given transition can be defined and they are cumulatively interpreted with the logical “AND”.

¹¹ <https://d3js.org/>

BeCoS Modeling

Load JSON Save JSON Push to Database Pull from Database Lock Unlock

Home Elements Interactions State Machines Scenarios

Behaving Elements

SwitchPosition

State Machines

name --	description --	
Switch		x

```

stateDiagram-v2
    state Closed as Closed {
        deltaV = 0
    }
    state Open as Open {
        i = 0
    }
    Open --> Closed : CLOSE {{ deltaV > 0 },]
    Closed --> Open : OPEN {{ i > 0 },]
  
```

Help

State Closed

Description:

Switch in the closed state

Associated Constraints

Behaving Element: Switch

name --	description --	expression	
Unnamed		$\delta V = 0$	Edit x

Now

Fig. 10. Users specify state machines through a graphical editor in the State Machines tab. Additional information, such as state constraints and transition guards and triggers, can be added by selecting an entity and using the table editor. In the example shown, two states of a Switch component are shown, with appropriate transitions (with specified guards and trigger events) between them.

Finally, in the Scenarios tab (Figure 11), the user declaratively specifies the evolution of a component’s behavior (i.e. its state) over time using a temporal constraint network [6] representation. In the example shown in Figure 11, a scenario is developed that turns the lamp on. Initially, the switch is Open and the controller is turned Off. After five seconds, the controller is turned to Standby, and 10 to 15 seconds later, the controller is turned On and simultaneously the switch is Closed. After 30 seconds, the Controller is turned back to Standby, and the switch is simultaneously Opened.

Like the State Machine tab, the primary workspace of the Scenario tab is created using the D3 library. In this tab, users create scenarios, also referred to as “rules” in BeCoS, and add execution contexts, which representing a behaving element, to them by selecting the “+” icon next to the behaving elements listed in the containment tree. The elements that compose a scenario (i.e. timepoints, temporal constraints, and schedulable behavior constraints, which are also called activities) are added to the execution contexts by selecting the appropriate icon for a given element, and then clicking inside the scenario workspace. Activities and “precedes” temporal constraints are editable within the Inspector pane (Figure 12), where a name and schedulable behavior constraints can be added to an activity or a minimum and maximum duration can be added to a “precedes” temporal constraint. For the latter, these durations are displayed on the temporal constraints in the “[min, max]” syntax (e.g. “[10, 15]”), which indicates the subsequent timepoint occurs between 10 and 15 seconds after the preceding timepoint. Such a temporal range is acceptable, and expected, when specifying scenarios declaratively. The timepoints may be further constrained or grounded by a scheduling tool outside the BeCoS tool.

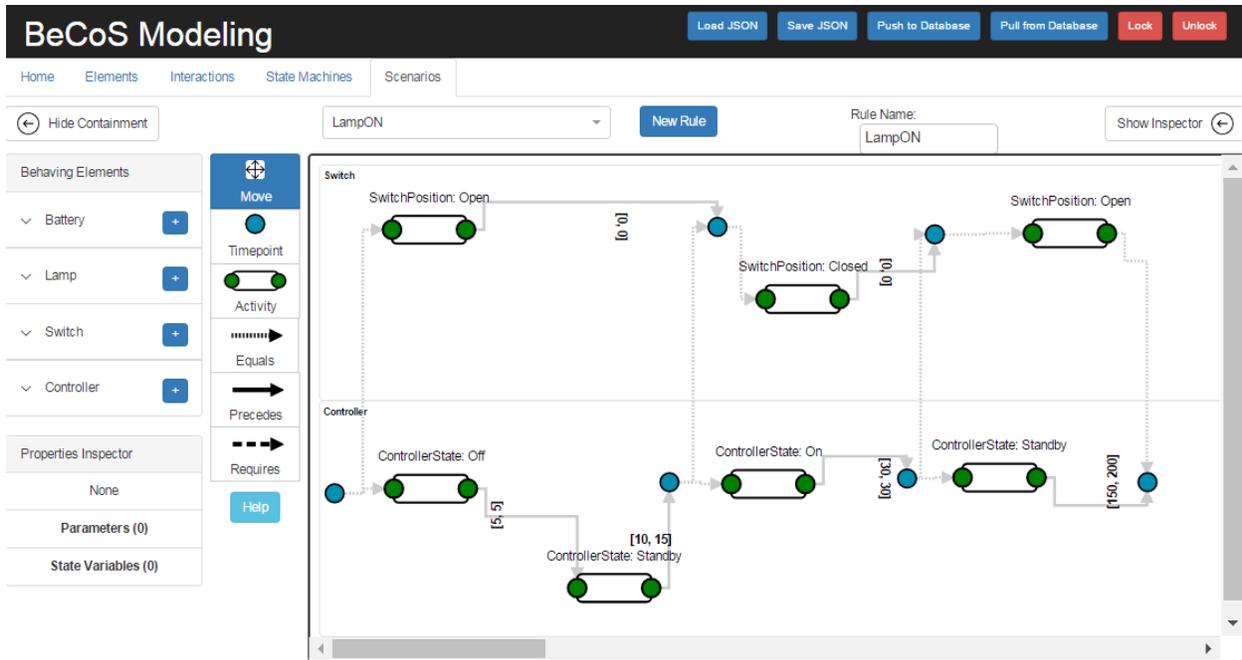


Fig. 11. The Scenario tab is where users can build a temporal constraint network and declaratively assert how component states evolve over time. The example scenario specifies a Switch state changing from Open to Closed, then back to Open, in response to a Controller component in its On state.

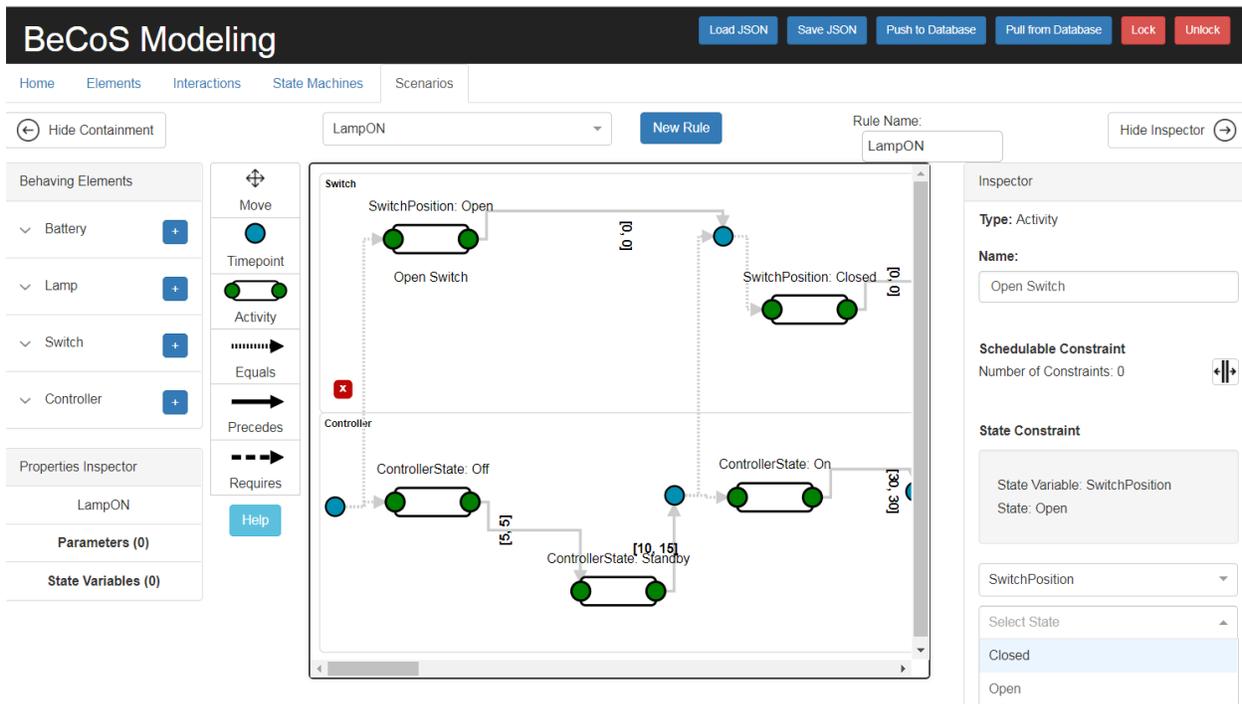


Fig. 12. In constructing a rule, users can add a name, state constraint, or schedulable constraint to an activity through the inspector pane, which is displayed on the right side of the Scenario tab.

V. Status and Future Work

While the BeCoS tool is currently a prototype, it has been the focus of multiple years of modest development and maturation. Select systems engineers on the planned Europa Clipper mission have tested the deployed BeCoS tool

and have provided feedback. This feedback has informed use cases that are important to consider for a production tool, and software bugs and tool improvements identified from this feedback are the focus of the current developmental effort.

In addition to these improvements, substantial effort will be applied this fiscal year to connect BeCoS with a JPL-developed activity scheduler and plan generator (APGEN [7]), and Europa Clipper data will be used as a test case. APGEN is currently being used on several flight projects, and on Europa Clipper, it is the primary tool for producing activity timelines for the Mission Plan. In the current paradigm, behaviors and scenarios are specified in the APGEN project adaptation layer, which serves as the “input” to APGEN; an APGEN modeler manually programs this adaptation layer by interpreting behaviors from a variety of sources including Excel spreadsheets, PowerPoint slides, emails, and verbal conversations. Upon execution, APGEN schedules activities for the mission resulting in activity timelines for each modeled component. One persistent issue with this approach is the unverifiability of the modeled system behavior. Verifiable behaviors are particularly important since the manual interpretation of behavior in creating the adaptation layer can lead to an erroneous implementation. Currently, the only way a systems engineer verifies the APGEN model is by either analyzing its outputs, which by definition is not verifying the modeled behavior (the inputs), or by inspecting the adaptation layer, which is rarely undertaken as very few systems engineers can understand APGEN code (and even if they could, it is in the form of distributed imperative logic, which is inherently hard to review). A more ideal and robust approach involves the systems engineer directly specifying behavior in a declarative behavior model using a tool like BeCoS, which can be more easily inspected and verified. This model would be exported to the APGEN adaptation layer, and would be automatically transformed to populate behaviors in the required syntax. This new workflow requires agreement on the type and form of information being exchanged, and requires modifications to APGEN so that it accepts declaratively-specified scenarios instead of its usual imperatively-specified ones.

Another implementation concern that needs to be addressed in BeCoS is that of types versus instances. A type can have many instances. Each instance inherits behavior that is defined on the type, while each instance can be in a state that is different from another instance. For example, a spacecraft uses a particular reaction wheel model that has a defined behavior – this behavior would be defined for the reaction wheel type. Typically a spacecraft will have three or four instances of this reaction wheel. Each instance inherits behavior defined on the reaction wheel type (so the behavior is only defined once), while each instance can be in a different state (e.g. three of the instances can be in a high-speed state while the fourth instance is off). Thus in BeCoS, the intrinsic behavior specified in the Element and State Machine tabs should be specified on the type, while scenarios should specify behavior on instances. However, the current implementation of BeCoS does not distinguish between type and instance. As a workaround, it is assumed that everything in BeCoS is modeled as an instance, which is acceptable in the short-term, and implies that all behavior must be duplicated for multiple instances that would have otherwise originated from a single type.

VI. Conclusion

The BeCoS tool is a web application developed to enable systems engineers to easily and rigorously specify behaviors and scenarios in a semantically-meaningful manner. The aim is to enable behavioral modeling on flight projects earlier in their lifecycle; doing so will enable new model-checking analyses, such as state reachability analyses or analyses that ensure scenarios are acyclic graphs. Additionally, this tool allows behaviors and scenarios to be viewable and verifiable, and ultimately exported to scheduling or simulation tools, which ensures the behaviors being analyzed are always consistent across the various analyses and execution tools.

Acknowledgments

The authors would like to thank JPL’s Integrated Model-Centric Engineering (IMCE) initiative, which has funded this work over the past several years. In addition, the authors would like to thank the Europa Clipper project, which has sponsored several summer interns that have worked on BeCoS. The authors would also like to thank colleagues that have contributed or provided guidance and direction to this developmental effort, namely: Jean-Francois Castet, Erika Hill, Deanna Heer, Zachery Miranda, David Tsui, Thomas Kwak, Tyler Ryan, Brandon Wang, Steven Jenkins, and Nicolas Rouquette.

The work described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. © 2017. All rights reserved.

References

- [1] Ingham, M., Rasmussen, R., Bennett, M., and Moncada, A., "Engineering Complex Embedded Systems with State Analysis and the Mission Data System," *AIAA Journal of Aerospace Computing, Information and Communication*, Vol. 2, No. 12, Dec. 2005, pp. 507-536.
- [2] Dvorak, D., et al. "Goal-Based Operations: An Overview," *AIAA Infotech@Aerospace 2007 Conference*, Rohnert Park, CA, May 7-10, 2007.
- [3] Jones, G., et al. "Human-Rated Automation and Robotics," *Jet Propulsion Laboratory, JPL D-66871*, Pasadena, CA, 2010.
- [4] Castet, J.F., et al. "Ontology and modeling patterns for state-based behavior representation," *AIAA Infotech@Aerospace Conference*, Kissimmee, FL, January 5-9, 2015.
- [5] Chung, S. H., and Bindschadler, D. L., "Timeline-Based Mission Operations Architecture: An Overview," *Proceedings of the 12th International Conference on Space Operations*, Stockholm, Sweden, June 11–15 2012.
- [6] Dechter, R., Meiri, I., and Pearl, J., "Temporal Constraint Networks," *Artificial Intelligence* 49, 1991, pp. 61-95.
- [7] Maldague, P., et al. "APGEN Scheduling: 15 Years of Experience in Planning Automation," *AIAA SpaceOps Conference*, Pasadena, CA, May 5-9, 2014.