ExecSpec: Visually Designing and Operating a Finite State Machine-based Spacecraft Autonomy System

Russell Turner, Sharjeel Hooda, John Gersh, George Cancro
The Johns Hopkins University Applied Physics Laboratory
{russell.turner | sharjeel.hooda | john.gersh | george.cancro}@jhuapl.edu

Abstract

The increasing complexity of modern spacecraft autonomy systems makes it difficult for engineers, mission operators, and domain experts to completely understand them during design and development and to analyze autonomy behavior during testing and flight operations. To address these problems, The Johns Hopkins University Applied Physics Lab is exploring a new visual programming approach to autonomy development based on a combination of finite state machines (FSM) and data flow diagrams. This allows autonomy developers to construct a system of reusable FSMs interactively and link them to form autonomy components and subsystems. This paper describes our current work creating such a visual autonomy development environment, called ExecSpec. It provides an overview of the system, discusses issues for handling the visual complexity caused by multiple state machine systems, and describes how mission operators can use visual monitoring tools during flight to analyze on-board autonomy status.

1. Introduction

Spacecraft autonomy software development is particularly challenging because of the need to respond reliably to a variety of potential faults in complex hardware and software systems. Furthermore, autonomy systems are implemented in software developed by software engineers, while knowledge of the spacecraft hardware and subsystems lies in the system engineering, mission operations, and domain expert communities. As a result, problems frequently arise during autonomy software development because of poor communication and coordination between the domain experts and the autonomy software developers, as well conflicting domain expert goals.

This problem has been identified by Ingham [1], who has proposed a solution using an intermediate language common to systems engineers, experts, and developers. Languages alone are not sufficient to solve these problems if they must be translated by hand into underlying software code. Rather, a seamless autonomy development process must be put in place to translate these higher-level languages into autonomy implementations.

Ideally, an autonomy development system would be able to keep software developers, system engineers and domain experts involved in the autonomy design throughout the entire development lifecycle so that all experts can understand the ramifications of every decision made during that process.

At The Johns Hopkins University Applied Physics Laboratory, we have been working on a prototype autonomy development system to provide system engineering and domain experts with understandable, formalized, visual models that can directly translate into operational functionality [2]. Our approach, called ExecSpec was guided by the following principles:

Understandable – any domain expert can understand and create the design. Domain experts who are not software engineers must be able to review and design the autonomy system.

Design is the Code – the design should be directly convertible into a form that can be executed on a target system.

Pre- and Post-Launch Modifiability – The autonomy system should be able to be modified, disabled, or enabled in parts or in whole after launch by Mission Operations.

Quick Test Cycle for Rapid Deployment – The autonomy development system should enable interactive and automated verification techniques that can be used prior to upload.

User-Driven Interactive Visual Simulation – The autonomy system should enable a user to visually inspect the design as the user drives the autonomy
system through a scripted simulation or through interactive injection of events.

Automated Verification – The autonomy system design should promote the use of automated testing techniques, such as formal verification, to provide mechanisms for identifying flaws that may be difficult to find through manual inspection.

Commanding and Telemetry at any level – The autonomy system shall allow low-level commanding to be used simultaneously with full autonomous functionality.

Support Situation Awareness – The autonomy system should provide readily understandable descriptions and visualizations of the overall state of the spacecraft and its subsystems.

In this paper we will give an overview of our prototype system, called ExecSpec. The system is based on the direct execution of finite state machine diagrams. Although finite state machine state-transition diagrams have been used in the design and development of spacecraft autonomy systems for some time, they have always been translated into autonomy flight code either by hand or through automatic code generation. Hand-coding increases the possibility of errors and reduces flexibility for future change, while auto code generation makes it difficult to seamlessly upload, monitor and modify the autonomy systems during flight.

To address these limitations, we are developing a prototype of a new visual autonomy development environment that outputs the specification of a finite state machine system. This executable specification can be uploaded directly to the simulator or spacecraft at any time during testing or flight operations. Once on-board, a generic FSM interpreter executes the specification and reports the status of its execution through telemetry to the ground, where a visual presentation displays the spacecraft behavior by dynamically highlighting portions of the state-transition diagram. In this way, ExecSpec will be able to serve as a single integrated visual tool allowing team members to design, test, debug, and deploy autonomous behaviors, as well as to monitor and modify them during mission operations.

In addition, ExecSpec is designed to support the rapid assembly of autonomy systems through a prototype-instance method: individual finite state machines with well-defined input and output variables are represented visually and can be interconnected to form networks of interacting finite state machines. These visual system components can be saved in libraries of component prototypes which can be used to instantiate and assemble new autonomy system diagrams. This increases the reusability of autonomy system components among spacecraft and decreases total development time.

1.1. Finite State Machines

A finite state machine (FSM) is a model of computation that allows for transitions within a fixed number of defined states. At any particular time, the state machine may only be in one of the states, and transitions specify which states may be reached from the current state. Conditions on the transitions indicate the input event that will cause the machine to follow that transition to the next state. A finite state machine can be represented as a directed graph, where states and transitions are represented by nodes and edges, respectively.

Two major categories of FSMs are acceptors and transducers. An acceptor (or recognizer) FSM generates a binary output indicating that it either accepts or rejects the input given. Transducer FSMs differ in that outputs are generated by actions performed by movement through the FSM states. Actions can be associated with a particular input condition or transition, known as a Mealy machine, or upon entering/exiting a state, known as a Moore machine.

An FSM interpreter implements and executes a finite state machine. The interpreter parses a string of inputs to the FSM, changing state when transition conditions have been met and outputs a string of actions as it traverses the states.

1.2. Visual Programming

The simplicity and finite nature of FSMs makes them well-suited for visual programming. A visual representation of states and transitions based on a directed graph node-link diagram can easily be created and modified during development. Dataflow languages are also amenable to visual representation and have grown in popularity over the last twenty years [3]. They allow software components to be represented as "black boxes" which may be interconnected through well-defined input and output data interfaces.

By treating each FSM as a module in a dataflow programming language, diagrams representing systems of interacting state machines can be depicted graphically, providing a visual development environment that allows a user to design all the components of an FSM system interactively.

Once a designer has created a full or partial FSM system, a model checking tool can be used to validate the design against a requirements specification and
highlight counter-examples. One such verification tool is NuSMV, a symbolic model checker designed specifically to describe finite state machines [12]. The designer or test engineer can utilize the visual programming environment to test and debug the FSM against a simulator. This key benefit of the visual programming environment allows for users to determine the flow within the FSM and pinpoint potential problems quickly.

1.3. Spacecraft Autonomy Applications

Finite state machines are particularly appropriate for designing embedded, real-time systems that must react to input events, and therefore are a good candidate for use in spacecraft autonomy design. We believe that FSM state-transition diagrams are easy for autonomy system designers to understand, and will improve design quality and the efficiency of the design process. In operation, FSM specifications can be interpreted directly by an on-board FSM interpreter, simplifying operations and enhancing reliability.

A dynamic FSM visualization driven by telemetry could also be used by mission operators to continuously monitor behavior during flight. Such a visualization can provide real-time status of the state of the autonomy FSM and help localize a problem when it occurs.

2. Previous work

2.1. Visual Programming Tools

There are a number of commercial and open-source visual programming tools. Dataflow visual programming tools such as LabView, AVS and Matlab Simulink allow users to link modules together in circuit-like diagrams to implement applications in the domains of signal processing, scientific visualization and simulation. Two commercial visual finite state machine programming tools which allow the user to draw state-transition diagrams from which code can be generated are Matlab's Stateflow and Stateworks. Stateflow uses Harel's statecharts [4] as its diagramming notation, while Stateworks is based on the virtual finite state machine (VFSM) concept [5].

Harel statecharts extend the finite state machine concept in a number of ways, including the use of hierarchy, and are the basis for the statechart diagrams of the Unified Modeling Language (UML) standard. UML statecharts have been incorporated into a number of UML software engineering tools [6], such as MagicDraw and the Rhapsody model-driven development environment.

The VFSM methodology separates the control behavior of a software module specified as a finite state machine from the input and output interfaces to the underlying software platform. The resulting diagrams are simpler than the Harel statecharts, having only Boolean condition variables on the transitions and no state hierarchy.

2.2. FSM-Based Spacecraft Autonomy

State-transition diagrams have been used in the design process of several spacecraft autonomy systems, including those in APL's Solar-Terrestrial Relations Observatory mission (STEREO) [7]. The Jet Propulsion Laboratory (JPL) introduced the additional step of auto-coding statecharts for spacecraft fault protection programming in the Deep Space 1 mission [8]. The statecharts were drawn using Matlab's Stateflow tool. Stateflow automatically generated C code from the diagrams, which was then post-processed using JPL's own tools in order to integrate the code into the spacecraft's flight software. JPL's Deep Impact mission [9] also used Stateflow to draw and simulate fault protection flight code. Since the flight software was written in C++, additional post-processing tools were required, including the use of supplemental information in spreadsheets to convert the generated code into flight code. These tools are part of a software development and verification process at JPL called STAARS (STate-based Architecture and Auto-coding for Real-time Systems) [10].

While the JPL flight software designers reported a positive experience with the technique, we believe that the use of commercial off-the-shelf drawing tools, together with specialized code generation and post-processing does not provide the end-to-end flexibility and operations monitoring capability necessary for next generation autonomy development systems.

3. ExecSpec overview

ExecSpec is a prototype system we are developing to demonstrate our new approach to spacecraft autonomy development. It consists of two main components: an on-board FSM interpreter embedded within the spacecraft flight system, and a visual ground support tool for designing, testing, simulating, debugging, uploading and monitoring executable specification diagrams. The executable specifications
consist of multiple communicating finite state machines, each of which can be separately modified, uploaded, enabled and disabled in real time without on-board flight-code changes.

Although there exist several commercial systems for doing visual programming using FSM diagrams, we chose to implement our own tools for several reasons. First, existing systems do not have the right level of visual formalism necessary for autonomy programming. Tools such as Stateflow are based on UML statecharts, which we believe are too complex for non-programmers to understand. VFSM-based systems were not, in our opinion, expressive enough to give a readily understandable model of the autonomy system, due to their lack of state hierarchy. Finally, we wanted to build an integrated system with the ability to seamlessly upload specifications to the spacecraft and monitor the results visually, and we weren’t confident that the existing commercial systems could be adapted easily for such purposes.

3.1. System Architecture

The ExecSpec system consists of a ground system component used by designers and operators, and a flight system component that runs in the spacecraft or a test system. The ground system component consists of a visual tool for editing state machine diagrams, a model checker for verification, and a tool for converting the diagrams into a compact binary form for upload to the flight system. The flight system executes the FSM model, receives commands from the ground system and telemeters transition and state information back to the ground system.

3.2. Visual formalism

The state machine diagramming notation we adopted is based on VFSMs, with a few extensions similar to UML/Harel Statecharts (see Figure 1). An ExecSpec diagram takes the form of a hierarchical node-link diagram consisting of nodes connected by links. Various types of nodes are represented by different shapes, referred to as glyphs, while links between nodes are represented by lines.

3.3. Finite State Machines.

Rounded rectangle glyphs are used to represent states, connected by arrowed lines representing transitions. States drawn within other states express a hierarchal relationship. This is similar to Statecharts except that parallelism and orthogonality are not allowed and only Boolean conditions are allowed on transitions.

Each state can have entry and exit actions, which may be present at any level in the state hierarchy. Transitioning across lines of hierarchy executes the actions of the starting and final state as well as any intermediate states in the hierarchy. Every state machine has an initial state, indicated by a line arrow beginning with a dot which is entered upon system start-up. Every parent state within a state machine has a default child state indicated by a line arrow from the parent state’s border.

Figure 1: Example TWTA controller state machine modeled using the ExecSpec diagramming notation.

Transition links from state to state can originate and terminate at any level within the hierarchy of a state diagram. Non-default transitions can be of two types: passive (green) and active (blue). Active transitions, which represent autonomous control, execute entry and exit actions when traversed. Passive transitions, which represent actions performed by entities outside the autonomy software such as ground commands or hardware, will not cause any actions to be executed. By modeling non-autonomy entities within the same diagram as the autonomous control, a domain expert can view everything that would potentially affect the desired concept of operations. This feature also accomplishes the principle of enabling commanding at any level.

Transitions are labeled with the name of a specific input condition which, when it is set to true, will cause the state machine to change state along the transition.
3.4. Modeling Systems of State Machines

Practical executable specifications require multiple finite state machines interacting with each other. Also, to support engineering of large systems, there must be mechanisms to support modular components that can be saved, reused and composed in different ways. To represent this visually, we have used a hardware circuit diagram metaphor similar to LabView [11] and other data flow visual programming tools (see Figure 2).

State machines are considered to be self-contained components with well-defined input and output variables and are represented by rectangular glyphs containing the individual FSM diagrams. The input variables come in standard types (e.g. Boolean, real, integer) and can be attached along the left side (for inputs) or the right side (for outputs) of the state machine glyph. Variable types are indicated by glyph shape, e.g. triangles for Booleans, squares for integers. Inside the state machine, input variables are connected by link arrows to the condition labels they represent on transition arrows, while output variables are connected by link arrows to a state's entry and exit action glyphs. Outside the state machine, output variables of state machines may be connected to the input variables of another state machine to form dataflow diagrams visually similar to electronic integrated circuit diagrams.

Two predefined types of state machines with Boolean outputs have special glyphs to represent them: latches and timers. Latches set their output value to true when set, and reset their value to false when cleared. Timers are like Latches except that they set their value to true a specified time after being set. These are used to connect the entry and exit action outputs of States to persistent Boolean output values.

Finite state machine systems are connected to the real world or to each other via external data inputs and output actions. Input data are transformed into Boolean values by Condition nodes, which contain Boolean-valued expressions of their input variables. Conditions are represented by arrowed rectangle glyphs, and can be located either inside State Machines, where they are linked to transition conditions, or outside of State Machines, where they can be linked to Boolean State Machine inputs. Output actions are executed by Script nodes, which contain spacecraft system commands to be executed. Scripts are represented by rectangular icon glyphs with an input action variable that, when triggered, causes the commands to be executed (see Figure 3). State entry and exit actions, shown with narrow triangle glyphs, can be connected with links to Scripts directly, or via state machine outputs.

3.5. External Inputs and Outputs

Automatic verification of an ExecSpec model is done by converting it into a form readable by the NuSMV program in order to discover counter-examples derived from system design requirements. The ExecSpec project has developed a prototype model checking component that takes the ExecSpec finite state machine model and converts it to a NuSMV-readable form. An initial feasibility study of this system was made by encoding the entire STEREO fault protection system using the ExecSpec finite state machine formalism, resulting in a model consisting of 46 charts covering the full functionality of STEREO’s rule/macro based system. Details of this are explained in a companion paper [13].

4. ExecSpec Designer

The ExecSpec Designer (ESD) component of the ExecSpec system is a complete interactive development environment (IDE) for spacecraft autonomy systems. As such, it performs many of the functions of a typical IDE for general-purpose programming languages such as Eclipse or Visual C++, such as coding, debugging, testing, and managing components.
Since the "code" in this case is a visual specification, the central component of the ESD is the diagram editor, which allows interactive drawing of systems of finite state machine diagrams. The diagrams can be created from scratch, using a selection of drawing tools, or by stamping out prefabricated components selected from a palette of standard FSM state machine diagrams. The diagrams can be viewed and interacted with at different levels of detail depending on the task. At a low level of detail, only the top-level state machines and their interconnections are visible. At the highest level of detail, all the details of each individual state machine are displayed schematically, including input conditions and output actions.

To support interactive testing and debugging, the ESD diagrams can be manipulated by the user to change the state of various input conditions. These changes are communicated to the FSM interpreter, located in the ExecSpec flight component, and the resulting changes to the state of the FSMs are presented by animating the diagram. A historical view of the diagram's dynamic behavior can be seen in the ESD's timeline view, which displays time histories of autonomy state variables together with spacecraft telemetry points, and which has a time slider for interactive time selection (see figure 4).

4.1. Diagram View

The ESD diagram view is the area in which ExecSpec diagrams are drawn, edited and viewed by the user. The node and link glyphs of the diagrams are rendered as scalable drawings so they can be zoomed in and out to any size. The diagram is manipulated by selecting model tool icons from a toolbar panel. New diagram nodes are created by selecting the appropriate creation tool and clicking on the drawing surface at the desired location. Nodes can be selected, dragged and resized using the selection tool. Nodes can be connected together using the link tool to drag either a link arrow between two Variables, or a Transition arrow between two States. The path of link and transition arrows can be adjusted via the vertex control points.

When a diagram is uploaded to the test flight system and executed, the view highlights the current state and transition within a State Machine by using color. The user can toggle the Boolean state of input Conditions between true and false by using the stimulate tool to click directly on its glyph. The input event is sent to the test flight system FSM interpreter and the resulting changes in state are sent back and animate the ESD diagram view.

4.2. Levels of Detail

Because executable specifications can become quite large and visually overwhelming, it is important for the user to be able to vary the degree of visual complexity. We accomplish this through the use of levels-of-detail. Each type of node has multiple visual representations representing greater or lesser amounts of visual information. These are referred to as levels-of-detail. As the user zooms the diagram in and out, individual nodes of the diagram switch their LOD depending on their physical size on the screen.

At the lowest level of detail, with the diagram zoomed all the way out, state machines appear simply as empty boxes with connections to unspecified inputs and outputs (see figure 5). As the user zooms the diagram in on a specific state machine, the LODs increase to display increasing detail. At the highest LOD, all information about the state machine is visible including states, transitions, input and output variables, and links to transition conditions and entry and exit actions.
4.3. Palettes

One of the most powerful techniques of software engineering is the use of reusable software components that can be assembled in various ways to form larger systems. Visually this can be accomplished through a prototype-instance method where a set of prototypical components can be viewed, copied and connected.

In the ESD, this is done using the palette view together with the clone tool. The clone tool acts in a manner similar to a node creation tool except that it makes a complete copy of whatever node is currently selected in the palette. The palette view is a separate view that can display one or more palette files containing prototypical state machine instances. By selecting instances from the palette view, cloning them onto the diagram view via the clone tool, and then connecting the cloned state machines to each other via the link tool, the user can rapidly build a new executable specification of significant complexity.

4.4. Animation

A benefit of visual programming is that program execution can be easily monitored by animating the diagrams. The ESD does this by highlighting the current state of state machines, the Boolean state of input conditions, and the transition that led to the current state as the state of the machine changes. The input values driving the state machine execution can be loaded from a scenario file, or the input conditions can be triggered in real-time by the user via the stimulate tool. During in-flight execution, the animation monitors the real-time state of the spacecraft autonomy system.

4.5. Timeline

The time history of FSM execution can be visualized directly using the timeline view. This view displays the evolving state values of selected finite state machine elements and input conditions as a function of time in a bar-graph representation. Media-player style playback controls allow the simulation to be played, stopped or rewound, while a time slider enables the user to view the state of the program execution at any desired time.

5. Mission Operations

During a spacecraft mission, the ExecSpec flight component would execute the autonomy specification on board the spacecraft, controlling the spacecraft’s autonomous actions. An operational version of the ESD, driven by real-time telemetry, could be used to monitor the execution of the on-board FSM, helping mission operators understand why particular autonomous actions were undertaken. Normally, such a tool would not be used to edit the diagrams, but, but it could be used to animate them in real time or to review historical data, showing how spacecraft conditions caused state transitions and how those transitions commanded on-board actions.

The effectiveness of interactive timeline playback for understanding causal relationships leading to autonomous spacecraft actions has been demonstrated for rule-based autonomy in the STEREO mission [14] [15]. We expect that it will prove equally, if not more, useful in complex FSM autonomy.

Should modifications need to be made to the autonomy system during flight, these could be developed and tested using the development version of the ESD prior to uploading to the spacecraft.

6. Conclusions

We have presented a prototype of a next generation spacecraft autonomy development environment based on visual programming of an executable finite state machine specification. Through a visual programming paradigm, we have also been able to introduce powerful software engineering techniques into the development of spacecraft autonomy such as interactive testing, debugging, validation, and monitoring of software execution, as well as rapid assembly of software components through a prototype-instance methodology.

We believe this is a step forward for spacecraft autonomy development and represents an effort to move away from rule-based systems and auto-code generation tools used on previous spacecraft missions. We also view it as a stepping stone to more powerful autonomy technologies, and we believe it would be a significant asset for Operationally Responsive Space (ORS) applications because developers can take advantage of a pre-existing library of subsystem components to rapidly design a new spacecraft autonomy system in a shortened timescale.

7. Acknowledgments

We would like to thank the many contributors to the ExecSpec project, including Bill Innanen, Eliezer Kahn, Chris Monaco, Lillian Nguyen, Chris Olson, Mike Pekala, Mike Trela, and Dan Wilson. This work
was supported in part by the NASA RBSP and ALHAT programs, as well as internal APL IR&D Funding.

8. References


