SSC14-II-2

Describing and Deploying Satellite Behaviors Using Rules-based Statecharts

Kenneth B. Center
PnP Innovations
3921 Academy Pkwy. North NE, Albuquerque, NM 87109; (505) 503-1563
Ken.center@pnpinnovations.com

ABSTRACT

PnP Innovations has developed the ability to graphically author Universal Modeling Language (UML) statechart diagrams that can be translated directly to rules-based flight code. The generated code module integrates with an Autonomous Mission Manager (AMM) Architecture developed by a working group at the Air Force Research Laboratory. The role of the statechart-described module behaviors can be many and varied – from subsystem mode transition management to fault tolerance techniques to courses of action taken in response to non-deterministic events. The capability provides a unique opportunity for system designers or operators (not programmers) to capture behaviors in a form that is natural and deploy them on a satellite asset without passing through an implementation cycle that could result in code errors. The resulting process ensures that the design always remains consistent with the deployed implementation.

INTRODUCTION

In the current climate of shrinking government budgets, space system developers need technologies and tools that allow them to do more for less. Meeting this need necessitates that these systems are more:

- Capable of operating without ground operator intervention (autonomous)
- Responsive to non-deterministic events
- Robust in overall utility to their user community
- Resilient in the face of anomalies

Achieving these objectives requires that satellites are capable of being tasked at a high level (in terms of overall goals) and that they are then capable of carrying out those directives using their own decision logic through sensible planning, mindful at all times of limitations and constraints imposed by installed resources and consumables.

Investing in innovation that moves toward the widespread use of satellite autonomy has the potential to yield significant benefits to the space situational awareness (SSA) community and Defensive Space Control (DSC). Most notably, if satellite platforms have the ability to detect and respond to events without exchanging information over the space-to-ground link (which is subject to latency and availability limits), their response to those events can be significantly more timely, leading to higher overall system utility.

This paper describes a recent addition to the AMM capabilities – the addition of the ability to express desired system behaviors as graphical statecharts and deploy them directly to an active system without any software coding.

AMM ACTIVITY

PnP Innovations is actively involved in supporting the development of an autonomy architecture targeting space domain Decision Support Systems (DSS). This Autonomous Mission Manager (AMM) architecture provides a powerful data framework by which modular elements can be interconnected (both locally on the satellite and between spatially distributed elements).[1]

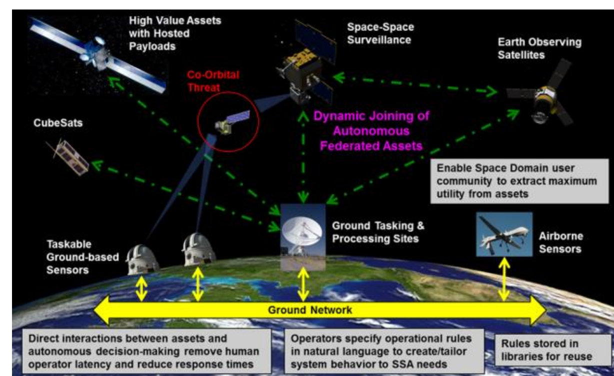

Figure 1: Federated Assets interacting in a distributed "Space WAN"

The standards being developed in support of this activity allow modules from different developers to be contributed and interoperate seamlessly. The AMM framework is already being used by a working group (comprised mostly of collaborating SBIR contractors)

to prototype an SSA system consisting of sensors models, spacecraft subsystem support software, data fusion algorithms, decision logic applications, planners, and ground stations. Referencing Figure 1, the intent is that elements exchanging data in an AMM architecture configuration can be seamlessly located across a Space Wide Area Network "Space WAN". Within nodes on the network, individual support components (physical sensors exposing compliant data interfaces and software applications performing specific SSA functions) may interact locally with each other but may also reach out to the wider network to interoperate with remote members of the architecture acting as "AMM Federated Assets".

The AMM working group has addressed the challenges associated with creating collaborative interactions between federated assets to achieve greater system utility though several use cases related to the monitoring of space-resident objects and responses to detected on-orbit events. Several components of the existing architecture utilize rules to define platform and mission behaviors in support of these system use cases. Our graphical statechart capture capabilities extend the use of rules-based programming by integrating a Jet Propulsion Laboratory (JPL) developed process named STAARS to better manage those rules by bridging the gap between UML-described behaviors and the resulting software. JPL has witnessed significant gains in software reliability and maintainability using STAARS on major NASA programs. Our extensions to AMM have yielded similar results.

AMM ARCHITECTURE

AMM fits within a larger service-oriented architectural framework that includes (from lower- to higher-functional utility layers as shown in Figure 2) device and application support, subsystem support, and finally autonomy. All layers are supported by a middleware framework called ASPIRE, which provides both a platform abstraction layer for common operating system services and a messaging framework supporting data interactions between applications and modular hardware deployed in a compliant space system.

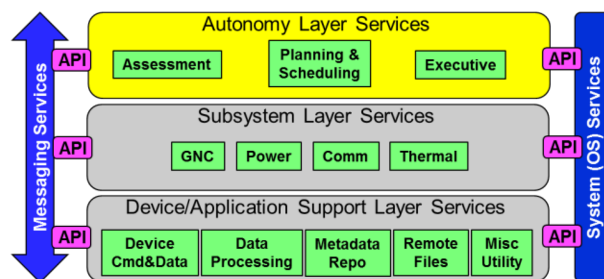

Figure 2: Larger Architecture Service Layers

Our AMM Working Group undertook a comprehensive trade study to determine the best course to take for supporting inter-module messaging. This trade led to the selection of the ASPIRE framework (Adaptive, Scalable, Portable Infrastructure for Responsive Engineering). ASPIRE was chosen because its native features were well-aligned with the CCSDS's Spacecraft Onboard Interface Service (SOIS)-defined capabilities and because ASPIRE's requirements, features, and source code could be controlled by the working group. Fundamentally, ASPIRE includes a transport agnostic, cross-subnet messaging service that can run on a variety of processing platforms (operating systems and processor architectures). ASPIRE uses a common message wire protocol to pass data and configuration information between components. A set of classes have been written to abstract message delivery, in both the local and non-local subnets. ASPIRE does not require specific hardware to execute. It is simply an application "wrapper" that can be attached at the lower level to specific hardware to abstract data interfaces.

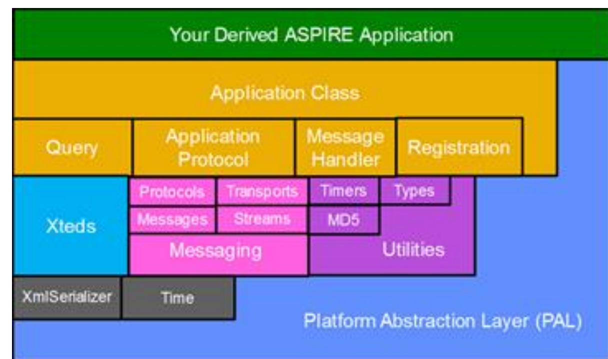

Figure 3: ASPIRE Layers indicating API Access Points

As shown in Figure 3, ASPIRE defines various layers which provide access to a range of functionality associated with registering a component, describing component interfaces with an XML data sheet, publishing messages as a provider, and locating other data interfaces using query services. The benefit of this layering is that the user of the ASPIRE infrastructure can take advantage of any layer to support integration of capabilities (for example, an application writer can use the Platform Abstraction Layer (PAL) to ensure that their code is executable on any of the supported platforms).

This layering approach helps prevent being locked in to any one underlying implementation and allows implementation specifics, such as OS and hardware, to be switched without affecting the above layers. Another similarly beneficial feature of ASPIRE is that APIs (Application Programming Interfaces) have been

developed for several high-level programming languages including C++, C#, and Java.

The modules in the autonomy layer can employ a variety of approaches toward facilitating the realization of operational goals. Procedural logic is one option – simply creating a software module that finds and attaches the appropriate abstract data sources to perform its role in the planning or execution process. This allows the software component to be highly reusable as long as the ontology remains static, but new requirements or features are not possible without modifying code and re-qualifying the module. Any time that the "seal is broken" on flight software in a major commercial program, the consequences can amount to millions of dollars. That said, many of the core services and functions in a satellite platform remain largely static. In these cases, it makes ultimate sense to take this approach of developing software modules with "long shelf lives" that can be configured/combined in modular building-block fashion to create typical satellite bus capability atop a potentially diverse set of supporting support hardware and processors.

USING RULES FOR DECISION LOGIC

Building on this paradigm and taking a step further, the AMM group moved to develop a capability to specify the operational behavior of a satellite (or the aggregate behavior of a collection of collaborating assets) using a rules-based approach. Rules are much closer to the "natural" language of engineering and better capture the mission-specific variations that typically discriminate one set of operations from another. Mission objectives and overall goals are also highly susceptible to change, particularly if there is an interest in the ability to adapt to dynamic global situations or in repurposing components to enhance or rebalance overall system utility. The AMM project was the initial proving ground for the application of rules-based decision logic. Initial efforts validated the rule engine capability with simple initial use cases – the management of dedicated solar array energy collection based upon battery telemetry. Later cases utilized rules for asset management, determining the conditions under which event messages should be published to other assets (using operator-prescribed rule logic) and on the tasked asset side determining responses based upon locally sensed state (such as an assigned latitude/longitude bound).

DROOLS AS THE RULE ENGINE

The Drools rules engine, its capability accessible via Java API calls, was adapted for compliance with the ASPIRE middleware being utilized for message exchange within the architecture.

We found the Drools package to be an exceptionally elegant choice for the representation of system behaviors. This is primarily because Drools is a modern enterprise language that leverages many Java-based packages to provide a very minimalistic, yet powerful top-tier overlay to declare facts and apply inference engine operations to them. It should be noted that Drools' inferencing is extremely efficient. The algorithms employed allow evaluation of enormous numbers of rules with relatively little processing, as witnessed by benchmarking activities that we performed on a Gumstix Overo Fire Comm processor, which demonstrated <1% processor utilization to execute one of the Drools-based AMM Agents for Energy Management.

THE ROLE OF STATECHARTS

The use of a rules-based paradigm becomes even more compelling when integrated with a full-service tool capability that allows behaviors to be specified by engineers or system users in graphical state charts, automatically generated to flight-ready software, and verified in robust mission simulation before deployment. State machines appear in many more software applications than most people realize in operational systems. Stateful, deterministic behaviors are requirements for most deployed applications, especially satellites, where risk tolerance is extremely low because of the difficulty of patching problems once the platform has slipped Earth's bounds and is only accessible via a thin, slow communications pipe. The agent-based autonomous software that PnP Innovations has developed for AMM and other space customers such as the Operationally Responsive Space Office is a good example of a situation where the statechart-to-code paradigm is extremely applicable. Every agent has an established stateful pattern like the one in Figure 4. Even though the states are not many, the potential transitions between them are quite numerous. As it turns out, the logic associated with the transitions is different for every case to which the agent is applied – sensor support, communications contact planning, power management… all need to maintain the concept of a common collection of agent states, but what causes transitions between states is highly varied.

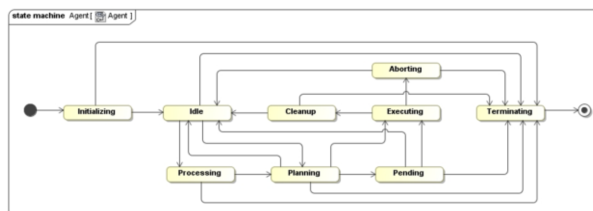

Figure 4: The complexities of transitions between major states of a software agent

So a great opportunity is presented to allow the standard "agent" state template to be grabbed from a library and then interactively manipulated with an editing tool to formulate the transition logic and actions associated with the supported states.

LEVERAGING STAARS

STAARS (STate-based Architecture and Auto-coding for Real-Time Systems) is a JPL model-based engineering process for software development.[2] Model-based engineering tools and processes provide a means to capture complex system behavior as high-level models. STAARS extensively utilizes UML Statechart (e.g. Harol State Machine Diagrams), which have proven useful for expressing and automatically deploying complex behaviors. STAARS comprises five pieces that can be customized for the needs of specific application. They include:

- ***MagicDraw*** – A UML CASE (Computer Aided Software Engineering) tool for entering models
- ***Statechart Framework*** – A state machine design pattern and associated event messaging middleware framework
- ***Autocoder*** – JPL developed state machine code generator
- ***Test-Harness*** – State machine simulation and run-time visual monitoring
- ***Promela*** – Model checking language produced from UML via Autocoder

The evolution of the STAARS process to its current form resulted from a gradual, well-considered effort at JPL to bridge the gap between the process approaches of systems engineers and software developers. Developing the integrated capability sought to find a cost-effective way to leverage lightweight Commercial Off-The-Shelf (COTS) tool sub-components to effectively merge model-based design approaches with the rigors of traditional software development practices. At JPL, there has been widespread buy-in from many projects because the resulting environment is highly flexible, easy to use, and very inexpensive to deploy to a large number of users.

The STAARS process was utilized on the Mars Curiosity Rover project at JPL (shown in Figure 5) and as such, both the process and generated flight software products (which supported the mission since launch and continues to run onboard during rover operations) have high heritage and Technical Readiness Level (TRL). In the Curiosity Mars Science Lander (MSL) program, the generated rules governed the high-level spacecraft modes (more than 50 primary states) including the cruise phase management of attitude control and maneuvering for nominal and off-nominal conditions, launch phase, Mars entry, descent, landing, and ultimately the "rover" mode. One of the key benefits realized in the course of using STAARS for the Curiosity program was that it enabled the accommodation of late-breaking requirements changes by adapting code automatically without software developers needing to intervene. Despite the late addition of mission operational features and responses, the entire design was naturally kept consistent with the flight software, allowing the program to be agile and (obviously) execute successfully. MSL used the STAARS Statechart Autocoder Rev1. Since then they have developed a Rev2 which has been re-engineered to include support for many different back-end coding languages, including C, C++, Python and Promela. In addition there are many more UML state-machine features supported, such as sub state-machines, termination states etc. This new Autocoder is actively been used on JPL's current flagship mission - SMAP (Soil Moisture Active Passive), an earth orbiting satellite**.**

Figure 5: STAARS specified state-based behaviors for the NASA JPL Curiosity Mission

PnP Innovations specifically leveraged two components of the STAARS framework for use in the AMM development effort, the UML Modeling Tool and the AutoCoder. The role of each is summarized next.

UML Modeling

The MagicDraw UML Modeling CASE tool is a commercially available product that has been incorporated into STAARS to facilitate the specification of system behaviors in the way that systems engineers can directly relate to their requirements. This consists of UML-standard State Machines and UML-standard Sequence Diagrams. Events passed between objects (state machines) in a sequence diagram typically become the events on state transitions in a statechart diagram, and these specifications map one-to-one with the constructs that

must be resident in software to meet the stated UML behaviors.

AutoCoder

JPL developed an autocoder in Java to convert the MagicDraw Statechart UML directly to C++ code according to their established software design patterns. This auto-generated code is highly readable and understandable and cannot be differentiated from code developed by software programmers to established architecture patterns. JPL abstracted a plug-in interface to facilitate support of multiple languages to target the needs of individual programs (as illustrated in Figure 6). The JPL Autocoder Rev. 2 back-end was extended using its plug-in interface to map MagicDraw Statechart UML models to either Python or Promela language implementations. For implementation languages such as C, C++ or Python a unique Python graphical widget is also generated that presents the original StateChart models as graphical displays. These Statechart Python widgets provide the ability to "animate" and display the Statechart while executing the autogenerated state machine implementation, providing quick run-time verification of operation.

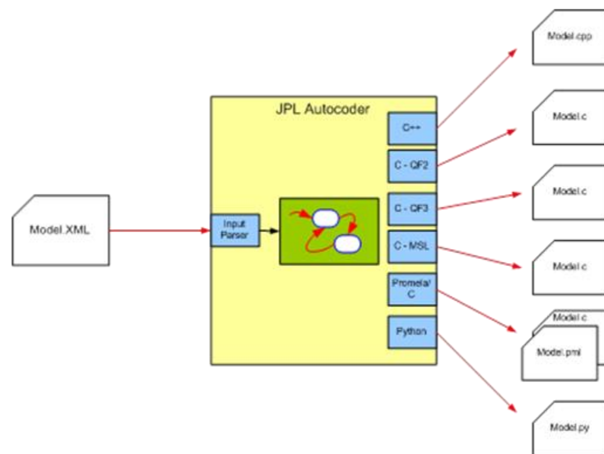

Figure 6: Auto-coding of model XML to flight code/rules in multiple languages

INTEGRATION WITH AMM

Figure 7 depicts the feature areas of the AMM architecture and the relationship of STAARS to those elements.

Starting on the right side of the figure, a deployed AMM-based system is a collection of software modules (or hardware devices providing AMM-compliant data interfaces) communicating by means of a network-oriented messaging layer. That layer, ASPIRE, facilitates the connection of these data sources based upon XML-based descriptions of AMM module data interfaces. The formalized interfaces in these xTEDS facilitate the Service Oriented Architecture (SOA) features that are desired for the composition of robust system capabilities leveraging high (or even complete) re-use of previously validated modules.

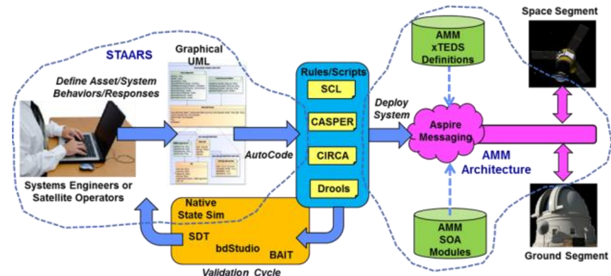

Figure 7: Role of the STAARS components in the AMM Architecture

AMM software modules pulled from the "library" include services such as "Object Catalog Management", "Pointing Director", "Slew & Keepout Planning Support", etc. These "commodity" modules will work on any type of platform utilizing underlying hardware infrastructure. The interfaces they present "upward" allow mission/system-level behaviors a uniform base to execute upon. Going forward, we presume these behaviors to be based on a variety of approaches, but in the figure we refer to them as the "Rules/Scripts" of the languages cited in the blue box. To be compliant with the AMM architecture, they must utilize the underlying ASPIRE message layer to inter-communicate.

What STAARS adds to the mix is the ability to graphically specify system behaviors and generate the products to implement those behaviors in the AMM architecture. The output of that process results in modules that are already compatible with AMM systems (by meeting the AMM message standard and ontology). One of the strong selling points for STAARS is the built-in feedback loop that allows the behavioral specifications to be validated (with ample visual presentation of the success or lack thereof). STAARS has a "native" state simulation capability, but it can be extended to use external simulation resources.

The task that PnP Innovations undertook was to create a new "back end" plug-in for the JPL auto-coder that converted the auto-coder's internal meta-model of states and state transitions into Drools rules. Actions within states (executed upon entry-to or exit-from that state) and the conditions that result in transitions between states are represented as rules that operate on *Facts* in the Drools *Knowledge Base*. Facts are either internal data items that manage the traversal of the statechart, or are bound to data that arrives or departs in ASPIRE messages. So two levels of auto-coder "back-

end” development were really required – the stamping of the rule patterns associated with statechart management, and the mapping of facts to AMM messages via the J-ASPIRE API. The result was a new back-end for the auto-coder that would translate UML diagrams with defined states and transitions into a Java module that could be dropped into an AMM system, locate messages that it needed to operate, and issue system control or produce event reports according to the statechart logic.

Figure 8 shows the stack-up of layers utilized to realize statechart execution in an AMM compliant system. At the top is auto-coded Drools language that uses rules to control state actions and transitions. It sits atop an IntelligentAgent class that declares knowledge-base facts and maps them to ASPIRE messaging in the layer at the bottom via the Java-based J-ASPIRE API. Through the ASPIRE middleware, data is reached that can drive the transitions in the statecharts (telemetry points compared to threshold values or system events). Actions within states (or resulting during transitions) can also be mapped to control of system resources, or provide events that signal a planning activity.

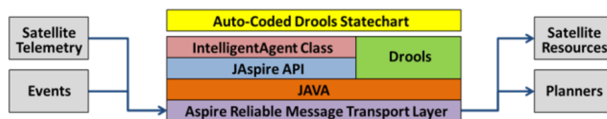

Figure 8: Stackup of a Drools Statechart and relation to AMM system data and services

The result of our work was a completely functional capability to create a graphical UML statechart interactively (defining states, sub-states, entry/exit conditions, transitions, guards, orthogonal regions – all standard UML statechart constructs), and use property fields in the MagicDraw tool to bind certain statechart data items to ASPIRE messages (see Figure 9 for snapshot of MagicDraw User Interface). Messages available in the AMM architecture can be imported based upon their xTEDS, which includes metadata about the message delivery rate, its variables, and other relevant parameters. Once imported, messages can be identified as “provider” or “consumer” (from the perspective of the module implementing the statechart behaviors). Provider messages result in the assertion of the statechart’s ability to deliver those messages to a subscriber, while consumer messages result in the statechart issuing ASPIRE queries to locate suitable matches to messages it needs to operate.

The nominal deployment process is to invoke the auto-coder to process the file created by the MagicDraw tool once it has been saved. Doing so will parse the XML file to build a meta-model of the statechart that is language agnostic – then call the back-end plugin specified by the user to select the programming language to be used to generate code. In our case, this is Drools/J-ASPIRE. Output of the autocoder is then uploaded to the target execution platform (in our case a testbed, but in final form a satellite’s mission processor). Once the file transfer is complete, the Java Runtime Environment (JRE) is called with the files. The module joins the running system by registering its own messages, locating needed messages, and then beginning to apply its statechart-described behaviors.

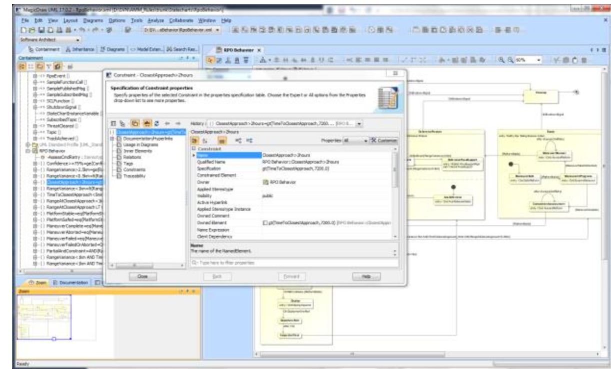

Figure 9: MagicDraw User Interface

As part of the research, we were able to verify the ability to dynamically update a statechart. A typical modification might involve editing some aspect of the UML diagram, such as adding an additional state and “re-wiring” the state transitions to use it. Once complete, the revised diagram is pushed through the same process described above, but at the end, the original module is terminated and the new module is immediately started. The ASPIRE dynamic message binding features allow this to be accomplished seamlessly. We utilize data browsing tools that provide us visibility into telemetry from the statechart modules that allow us to verify that behaviors are being properly represented. As soon as the new statechart module is installed it can be verified that the “re-wired” behaviors have been incorporated. This cycle takes all of a few minutes to accomplish (based on the complexity of the statechart modification).

APPLICATION USE CASES

We created statecharts designed to represent behaviors at several levels of satellite systems and in one case a domain outside of space operations.

One way to apply the charts is within an individual space asset (single satellite). To this end, we have implemented several examples.

In one case, we created a statechart that subscribes telemetry data associated with the satellite power

management system (battery charge level, voltage, current), as well as the interfaces to manage the attitude of the satellite and finally an agent arbitrator that ensures that satellite resources are not over-booked (to protect interfering with a higher-priority process that is utilizing attitude control, for example). The statechart monitors battery condition, checking for a variety of conditions that indicate the need for dedicated charging. If those conditions are satisfied, the statechart module interacts with the agent management function to schedule a slew to orient the solar panels to collect power. Considering eclipse conditions, the current rate of battery discharge, and other factors are all part of the logic that is described in the statechart.

The AMM working group has worked extensively on systems of federated assets. Another statechart was intended to determine courses of action for conditions where another visiting satellite was detected in the vicinity of the satellite hosting the statechart (using a variety of sensing techniques). The state knowledge associated with the visiting satellite and the confidence in that state knowledge combined with other sources of data to determine courses of action. These actions could be to request observation of the visitor by other federated assets, to take a defensive posture to avoid being compromised, or to maneuver to maintain a minimum separation distance. The nice thing about representing these actions in statecharts is that once they are complete it is trivial to tweak parameters used in the state logic. Changing threshold values for a critical separation distance (as an example) and actuating the change is easily accomplished with the flexibility of the ASPIRE messaging framework (or with the UML statechart editor for more structural updates).

These statecharts have broad utility and can be applied to many domains, not just space. We have used the same toolchain and ASPIRE middleware to assemble demonstrations of capability that apply to widely distributed data systems. One example was to orchestrate “checklist style” planning and cueing for courses of action (COA) for military operations (Figure 10 shows this particular UML statechart diagram – it is representative of the appearance of a typical collection of states and transitions). It is possible to create ASPIRE modules that access remote web-based services to acquire relevant data used in the planning process (weather conditions, predictions of GPS accuracy, etc.).

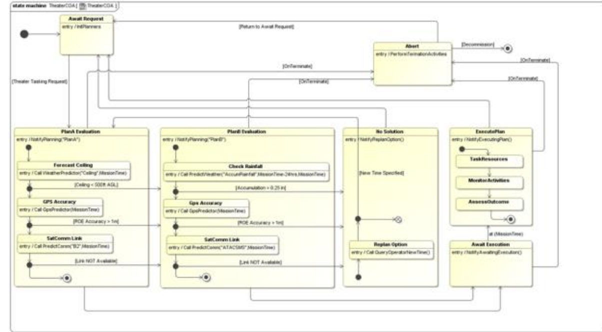

Figure 10: Sample UML Statechart defining courses of action based on external events and conditions

This data is then used within time windows to determine possible courses of action using fall-through logic approaches until a mission can be satisfied. Essentially this is automating procedures that humans perform manually in today’s world. Applying these tools to capture the routine tasks (while leaving the real brainwork intensive logic functions to the specialists) has the potential to significantly streamline COA planning processes by automating a large portion of the activities in the procedure manual.

RESULTS

We tried an exercise at the conclusion of our statechart-to-code technical development program to see just how much benefit this process capability could offer. Internally, several of our programmers have worked with ASPIRE for years, and regularly author code to implement state machine behavior in agents that manage one category of mission behavior for an operational satellite activity. The time to implement an agent for a new case (using state data, the attitude control subsystem, and onboard thrusters to maintain a user-defined orbit) was measured to be 43 hours (and that is the time required for a skilled programmer that knows exactly how to realize a desired set of capability). We used the statechart capture and auto-coder to conduct the same activity. Granted, there is a learning curve associated with using the UML tool (something we plan to streamline for better productivity in future efforts), but once it is understood how to bind statechart constructs to ASPIRE messages and services, specification of behaviors happens very quickly. Capturing a basic agent comparable in complexity to the hand-coded implementation only took 4 hours… That is over a factor of 10 decrease in time-to-implementation over the traditional approach, and the exercise can in practice be accomplished by a systems engineer or a specifically trained operator with no software programming experience!

CONCLUSION

The developed capability has shown great promise as a new approach to the development of mission-level flight software. The ability to capture a set of behaviors in an interactive tool and generate straight to code has the benefits of ensuring that the design remains consistent with the deployed implementation, since the code created is not generated by human programmers and thus subject to human errors. In addition to eliminating coding errors the implementation timeline shrinks to zero, so there are significant lifecycle development savings available. Furthermore, it opens the doors toward allowing non-programmers to capture and maintain mission-level behaviors of operational space platforms. This allows systems engineers and ground-ops personnel to be more tightly in the loop with satellite systems behavioral specification.

ACKNOWLEDGEMENTS

PnP Innovations is performing this research under the sponsorship of Paul Zetocha, Section Chief of the AFRL/RVSVC Space Vehicles Directorate at the Air Force Research Laboratory, Kirtland Air Force Base, Albuquerque, New Mexico. The associated SBIR is FA9453-13-M-0081, "Easy-to-Employ Satellite and Space System Robustness".

Many thanks to Garth Watney and Owen Cheng at NASA JPL for their support in acquiring the STAARS framework and assisting in getting up to speed on the auto-coder component.

REFERENCES

1. Center, K.B., Countney, P., Adams, R., Musliner, D.J., Pelican, M.J., Hamell, J., Kortenkamp, D., Hudson, M.B., Fausz, J.L., Zetocha, P., "Improving Decision Support Systems Through Development of a Modular Autonomy Architecture," Proceedings of the 2012 I-SAIRAS Conference, Turin, Italy, September 2012.
2. Mindock, J. and Watney, G., "Integrating System and Software Engineering Though Modeling," IEEEAC Paper No. 1666, Version 4, October 2007.