Verifying Executable Specifications of Spacecraft Autonomy

Michael Pekala, George Cancro, Jay Moore
The Johns Hopkins University Applied Physics Laboratory
{mike.pekala, george.cancro, jay.moore}@jhuapl.edu

Abstract

Traditionally, autonomous spacecraft fault protection systems are difficult to develop and test. This difficulty is due in large part to implementations that are not amenable to direct inspection by system engineers and a dependency on having the full spacecraft available for testing, which occurs late in the program lifecycle. In response, JHU/APL has recently been investigating a graphical, state-based approach to fault autonomy, which results in designs that are easily reviewable and amenable to formal analysis. This paper describes our recent work applying modern model checking tools to verify these graphical designs.

1. Introduction

On-board autonomous fault protection systems form a last line of defense to save a spacecraft in the event of anomalous conditions. A variety of approaches exist for implementing on-board fault protection, ranging from simple limit checks to sophisticated AI techniques that infer spacecraft state and recommend control actions [1]. Current practice tends toward the simpler approaches, due to their flight heritage and a natural aversion to risk associated with newer technologies.

However, even these relatively simple fault protection systems are notoriously difficult to develop and field. There are two main reasons for this. First, the implementations of these systems typically do not allow for adequate review by system engineers, resulting in a potential discrepancy between system-level requirements and actual system behavior. As a result, there is a significant reliance on the scenario-driven testing performed during spacecraft integration. This testing is expensive, as spacecraft test time is a limited resource, and also tends to uncover issues relatively late in the project lifecycle. This practice leads to the second main problem with these legacy approaches, which is difficulty quickly assessing the system-wide impact of potential modifications to the autonomy system, both pre- and post-launch.

The Johns Hopkins University Applied Physics Laboratory (JHU/APL) has recently been exploring executable specification techniques, whereby the state-based design formalisms utilized by system engineers also constitute the operational functionality. Instead of translating specifications into code, an on-board interpreter directly executes an uploadable form of the state-based design. This approach circumvents intermediary steps in the design process, where information may be lost or corrupted, resulting in a process benefit where system engineering and domain experts can collaborate on and adequately review a design with the guarantee that the spacecraft will act as the design intended.

The state-based design also has the advantage of segregating the data and control portions of the fault protection system. This allows for independent testing of the control logic earlier in the project lifecycle, even if the full details of the corresponding data representation haven’t been finalized. The control logic can be evaluated through visual inspection, interactive simulation and automatic, exhaustive testing.

This paper describes our recent work applying modern model checking tools to provide this latter automated testing capability. We discuss our approach for translating these designs into a form suitable for the NuSMV [2] model checker and provide preliminary experimental results for a representative spacecraft domain and an unmanned air vehicle (UAV) domain.

2. ExecSpec

Our overarching objective is to achieve an understandable autonomy design that can be readily tested and modified, whether through discovering new techniques or modifying existing technologies. The resulting approach, termed ExecSpec [3], is a combined flight and ground system based on the Bell
Labs concept of a Virtual Finite State Machine (VFSM) [4]. The relevant subset of ExecSpec formalism is described here; for further details, the reader is referred to [3].

2.1. VFSM Model

The VFSM methodology consists of two main components: the control logic, defined using finite state machines (FSMs), and a virtual environment, which insulates the control logic from data manipulation issues. In the ExecSpec adaptation of the VFSM model, all control logic signals are Booleans; the virtual environment converts telemetry values to Boolean input signals and maps Boolean output signals to spacecraft commands (see Figure 1). Although verification techniques could be applied to various software components within the VFSM framework, our focus to date has been exclusively upon verifying the control logic.

![Figure 1. VFSM framework](image)

2.2. FSM Model

The VFSM framework provides a default FSM model for specifying control logic, but expressly allows for alternative models. The first-generation ExecSpec FSM model adds a number of features to the default VFSM model for design convenience and alters the execution model for backward compatibility with the legacy fault protection system in order to facilitate transition into next-generation flight systems.

ExecSpec’s control model consists of multiple, communicating Mealy machines, similar to those described in [5]. Within a single FSM, transitions are conditioned on one or more signals, where a signal can be thought of as a Boolean variable that can only change value at precise times. FSMs are evaluated synchronously in a two-step algorithm. In the first step, input signals are evaluated and transitions selected for each FSM. In the second step, all FSMs change state and emit any associated output signals. Thus, each FSM executes a single transition each evaluation cycle. FSMs communicate by overlapping the output signals of one FSM with the input signals of another. From the perspective of an individual FSM, it does not matter whether the input signals are generated by other FSMs or externally from the virtual environment. Currently, no higher-level synchronization primitives are included in ExecSpec, but such primitives do exist in the literature and may be incorporated at a later date.

At design time, control logic diagrams are represented as a set of FSMs with hierarchical states, similar to Statecharts without AND nodes (i.e. one state per machine) [19]. The use of hierarchy allows for a more visually compact and intuitive representation of the logic. For example, consider an instrument that may have $N$ operational states where the instrument is powered ‘on’. With hierarchical states, a fault response that applies any time the instrument is powered ‘on’ can be encoded as a single transition from a higher-level ‘on’ state rather than $N$ separate transitions. Part of the process of converting a design to uploadable form consists of “flattening” these hierarchical designs to an equivalent set of interacting Mealy machines.

The ground component of ExecSpec includes a design tool that provides a library-driven model development environment and additional graphical programming constructs [6]. As with hierarchy, these constructs are reduced to equivalent structures that are permitted by the ExecSpec FSM formalism during the conversion to uploadable form.

3. Verification

A benefit to encoding control logic as FSMs is this representation facilitates the use of formal verification techniques. Although many formal methods exist, we have focused on model checking [7]. In the model checking approach, an engine exhaustively examines the state space of a system to determine the consistency of a user-defined temporal logic specification. When a specification fails to hold, the engine provides a concrete counterexample. The automatic nature of model-checking is particularly compelling for our desideratum of quickly ascertaining whether proposed modifications will have adverse effects on the overall system without resorting to labor-intensive manual proof techniques. A downside of model checking is tractability; checking large or complex designs is not always feasible and, in these cases, one must first employ some form of abstraction.

For our initial study, we selected the NuSMV model-checking engine, a successor to the SMV model
checker pioneered by Carnegie-Mellon University. Other popular options exist ([8], [9]); however, the NuSMV modeling language seemed like the best match for ExecSpec’s FSM execution model. Additionally, NuSMV offers a variety of useful features (e.g. BDD- and SAT-based checking, support for CTL and LTL formulae, automated abstraction techniques, interfaces to multiple SAT solvers).

To employ NuSMV, one must first translate the control logic into the SMV input language. This is a straightforward exercise and prior work has already demonstrated mapping a broader subset of Statecharts to SMV [10]. Each state machine within the autonomy design maps to a single module in SMV and each signal maps to a single Boolean variable. One less obvious feature we adopted from [10] was the concept of variable “monitors”. SMV disallows multiple assignments to the same variable, and variable monitors are used to mediate in cases where a signal is manipulated by multiple FSMs.

This translation process also optionally implements certain abstractions. One abstraction includes replacing signals local to a single FSM (such as timers) with non-deterministic choice. This variable deletion increases the set of possible behaviors (as we are under-constraining the model). For checking safety properties, increasing the set of behaviors is acceptable as, if undesirable behaviors are precluded in the more permissive model, they are also precluded in the more restrictive model [7][11]. In the case where safety specifications fail to hold, however, one must ensure the corresponding counterexamples are not a result of this relaxation.

Currently, the translation process supports multiple input sources, including StateFlow diagrams (that adhere to certain restrictions and diagramming conventions) and design files generated by the ExecSpec ground tool.

4. Experimental Results

4.1. STEREO Fault Protection

STEREO (Solar TERrestrial RElations Observatory) is the third mission in NASA’s Solar Terrestrial Probes program. STEREO uses two spacecraft, one leading Earth’s orbit and the other following, to take stereoscopic measurements of the Sun for the purpose of studying coronal mass ejections (CMEs) [12].

From an autonomy standpoint, the STEREO spacecraft employ relatively simple designs. The focus of the STEREO autonomy system is upon maintaining high availability, as CME events are brief and difficult to anticipate; therefore capturing them requires constant observation. The STEREO autonomy system uses a legacy approach of encoding fault protection as a set of rules (which check for anomalous conditions) and corresponding macros (corrective actions). STEREO’s fault protection implementation consists of 161 rules and 150 macros.

As an initial feasibility study, the authors re-encoded the STEREO fault protection logic using our restricted subset of Statecharts. The current incarnation consists of 46 charts and covers the full functionality of the original rule/macro system. The corresponding SMV model consists of approximately 60 input signals and 53 FSMs having 4.5 states on average. To exercise the NuSMV model checker on this system, the authors developed a set of 19 temporal logic specifications based directly on the STEREO fault protection requirements document (spanning 13 of approximately 90 total requirements). As an example, consider the STEREO requirement designed to prevent moving the antenna while the transponder is on:

“The spacecraft shall command the Traveling Wave Tube Amplifier (TWTA) into standby mode, prior to switching an antenna relay. After completing all relay commands to select the desired antenna, the spacecraft shall command the TWTA into transmit mode.”

Figures 2 and 3 show state diagrams for the Antenna and TWTA. A corresponding LTL specification can be written directly (in this specification G is a temporal combinator where Gφ means φ is holds for all time):

\[ G(! (\text{twta.state} = \text{RADIATING} \& (\text{antenna.state} = \text{SW\_TO\_MZ} \mid \text{antenna.state} = \text{SW\_TO\_PZ})) ) \]

Figure 2. TWTA model
The authors conducted simple timing tests using this suite of specifications. Initially, these timing results were run using the BDD-based algorithm with dynamic variable reordering enabled. While the runtimes were tractable, they were also less than ideal given the relatively simple autonomy design. Subsequently, we enabled cone of influence (COI) reduction, one of the automated abstraction techniques included with NuSMV. The intuition behind COI reduction is that many variables within the full model may be irrelevant for the purpose of proving a particular specification and one can instead prove the specification on the (hopefully much smaller) relevant subset of the model. COI reduction is known to be particularly effective for models consisting of multiple, loosely coupled processes [13].

For the STEREO model and our suite of specifications, COI reduction resulted in drastic improvements in runtime, up to a factor of ~10,000 in some cases (see Figure 4, note the log scale). Figure 5 shows the number of variables in the resulting cone of influence for each specification.

While these empirical results seem to indicate that model checking may indeed be practical for real-world ExecSpec autonomy designs, it is unclear whether the runtimes obtained using COI reduction generalize across the full set of STEREO requirements and whether the loosely coupled property of the STEREO design that presumably makes COI reduction so effective would be exhibited by other, more complex, fault protection designs. Additionally, in cases where there are choices in how to implement a particular autonomy design, it would be advantageous to understand how different design approaches impact the resulting verification task. Answering these questions is the current focus of our efforts in automated verification. In addition to continuing to work with the STEREO autonomy model, there are ongoing efforts to perform this same type of analysis on the New Horizons fault protection system. The New Horizons mission is functionally more complex and the fault protection system needs to manage a larger set of redundant capabilities than STEREO’s.

The authors also briefly experimented with analyzing randomly generated hierarchical designs; however, difficulty in insuring these designs were generated with the appropriate distributions and parameters so as to manifest the properties of actual autonomy designs led the authors to temporarily abandon this approach in favor of manually generated real-world examples. Similar issues with randomly generated models arise in other related domains, such as propositional SAT [14].

Although small, our set of test specifications did succeed in uncovering errors, ranging from simple typographic errors to more interesting design flaws. One of the more interesting design flaws emerged
when testing the Antenna/TWTA safety specification described previously. In the original design, when the TWTA was actively RADIATING and the antenna configuration needed to change the control logic correctly placed the TWTA in standby while reconfiguring the antenna array. However, the model checker discovered that if the antenna configuration were already underway prior to the TWTA entering the RADIATING state, it was possible for the TWTA to begin transmitting before the antenna array was finished reconfiguring. Adding additional conditions to the transition from READY to RADIATING in the TWTA model remedied this error. This counterexample is not merely a byproduct of our state-based representation and was in fact confirmed to be a defect within STEREO’s rule/macro design as well.

4.2. UAV Tactics

In order to demonstrate the broader applicability of ExecSpec and to increase the scope of our empirical verification results, the authors also created control logic designs for the domain of air-to-air combat. Unlike STEREO, where the focus is on fault protection, in this domain the ExecSpec state charts are used to capture logic pertaining to combat tactics. The scenario is a one-on-one engagement between two aircraft, at least one of which is autonomously controlled. The aircraft will be referred to as “blue” and “red”. Both blue and red aircraft have identical capabilities in terms of speed, turn rate, weapons, etc. For simplicity, maneuvers in this scenario are confined to the horizontal plane. Each vehicle is equipped with sensors that provide it access to the location, orientation, and velocity of the other. From this information, either aircraft is assumed to be able to deduce the control inputs that the opposing aircraft must have applied. The sensors have limited range, and if the distance between vehicles exceeds the maximum sensor range, the engagement is deemed to have ended in a draw. Each vehicle also carries an air-to-air missile. The missile can be fired whenever the target vehicle is within 22.5° left or right of the firing vehicle's nose, the firing vehicle is within 67.5° of the target's tail, and the range is within the missile’s maximum range. If a missile is fired, it is assumed to always destroy its target, ending the engagement in favor of the launching aircraft.

ExecSpec charts were used to encode defensive strategies for the blue aircraft and NuSMV was used to prove that a particular strategy would result in a draw at worst, or alternately to show a counter-example engagement in which the red aircraft was victorious. The red aircraft's strategy was left unconstrained, to represent an unpredictable adversary. In order to preserve physical laws and the capabilities of the aircraft, it was necessary to encode the dynamics and relative geometry of the aircraft as finite state machines; if left unconstrained, the model checker would be free to produce counterexamples corresponding to physically infeasible behaviors. The relative geometry was represented by a 17x17 square grid. The origin of the grid is defined as the position of the blue aircraft, with the positive x-axis in blue's direction of travel and the positive y-axis on blue's right wing. The relative position and orientation of the red aircraft is described by one of 17 x-position states, one of 17 y-position states, and one of 8 directional states. Each aircraft can have one of two speed states (slow or fast), and one of three turning states (left, straight, or right). The pilot or autonomous control system of each aircraft can choose desired speed and turn commands.

Each engagement begins with the aircraft in a nose-to-nose geometry at maximum sensor range. At each time step, the following sequence of phases takes place:

1. Both aircraft choose speed and turn commands.
2. The commands are used to update the aircraft's speed and turn states.
3. The effect of blue's motion on the relative geometry is applied.
4. The effect of red's motion on the relative geometry is applied.
5. The relative geometry is checked to see if either aircraft can fire its missile or if contact has been lost.
6. If the outcome of the engagement is still undecided, return to phase 1.

As previously stated, the strategy by which the red vehicle chooses its commands was left unconstrained. Two strategies were initially investigated for the blue vehicle. In the Copycat strategy, the blue vehicle chooses as its speed and turn commands for time t the same commands that red chose for time t-1. The Mirror strategy is similar except that blue chooses to turn left at t if red turned right at t-1, and vice-versa. It was hypothesized that one or both of these strategies would be provably “safe” defensive strategies for the blue aircraft in the sense that they would lead at worst to a draw, regardless of the strategy employed by the red aircraft. The model checker was, however, able to find counterexamples to the specification “red does not win” for both strategies, as illustrated in Figures 6 and 7 (In these figures, circles show locations of the two
aircraft. Lines show their orientations. The grid is always centered on the blue aircraft with its direction defining the positive vertical axis). The commands chosen by each aircraft in the counterexamples are shown in Table 1.

![Figure 6. Counterexample showing red winning against blue's Copycat strategy](image)

![Figure 7. Counterexample showing red winning against blue's Mirror strategy](image)

**Table 1. Sequence of commands in the counterexamples**

<table>
<thead>
<tr>
<th>Time</th>
<th>Blue</th>
<th>Red</th>
<th>Blue</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>slow, straight</td>
<td>fast, left</td>
<td>slow, straight</td>
<td>fast, left</td>
</tr>
<tr>
<td>1</td>
<td>fast, left</td>
<td>slow, right</td>
<td>fast, right</td>
<td>fast, straight</td>
</tr>
<tr>
<td>2</td>
<td>Red Wins</td>
<td>fast, straight</td>
<td>slow, straight</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>slow, straight</td>
<td>slow, straight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Red Wins</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In examining the counter-examples, the authors came to suspect that the amount of turn that could accumulate before blue could apply its copied or mirrored commands might be a key feature enabling red to win. To validate this insight, two more scenarios were checked. In the Copycat No Delay scenario, the first phase was broken into two phases, with red choosing its commands first, then blue choosing commands identical to red’s, thus eliminating the time delay between red’s and blue’s maneuvers. In this case, the model checker confirmed that red can never win. The Copycat Slow Turn scenario used the original Copycat strategy, including the 1-turn delay in copying commands, but the maximum turn rate of both vehicles was reduced from 135° per time step to 90°. In this case also, the model checker confirmed that red can never win. The evidence from the model checker supports the notion that the interplay of delay time and turn rate is important to the safety of the Copycat and Mirror defensive strategies.

Figure 9 shows timing results from the four scenarios. In each case, the model checker was asked to prove that the red UAV does not win the engagement (this specification holds in the “No Delay” and “Slow Turn” scenarios and fails in the others). For each case, NuSMV was used with and without COI reduction (blue bars indicate runs with COI reduction enabled). In these scenarios, COI does not provide nearly the same benefit as in the STEREO model. This is likely due to the smaller, more coupled structure of the model (14 state machines and no input signals), which presents fewer opportunities for abstraction. Our next steps involve expanding the sophistication of the tactics portion of the model and revisiting this analysis, as currently the bulk of the model consists of dynamics constraints.
Although the UAV tactics domain is fairly far removed from spacecraft fault autonomy, the UAV model does suggest that analyzing the performance of fault autonomy dealing with dynamic or physical constraints may be worth investigating as part of future work.

5. Related Work

Hierarchical state-based design is often standard practice in engineering fault protection systems and our effort is not the first to derive flight implementations directly from these designs. Deep Space 1 utilized fault autonomy software which was auto-generated from StateFlow Statecharts [15]. More recently, this same auto-coding approach has been used for flight software on the Space Interferometer Mission (SIM) [16]. While ExecSpec’s Statechart output differs (ExecSpec produces a FSM definition that will be interpreted on board instead of C/C++ code) these two approaches share the same underlying intuition.

Additionally, ours is not the first effort to employ model checking techniques to verify spacecraft fault protection systems. The SIM fault protection effort [16] describes plans to use SPIN to verify state-based designs as future work. Earlier work also cites success employing SPIN for fault autonomy systems [11]. SPIN excels at verifying asynchronous concurrent processes typically found in software verification problems [8], and may provide a complementary or alternative tool to NuSMV (especially if ExecSpec’s underlying FSM model evolves in a more asynchronous direction).

SMV also has heritage in spacecraft fault protection. Researches have used SMV to verify model-based autonomy designs [17]. In the avionics domain, NuSMV has been used to verify properties of the TCAS II collision avoidance system [18].

6. Conclusion

The main contribution of this work in terms of verification is to provide a piece of empirical evidence that suggests the plausibility of using Statecharts in conjunction with model checking in the domain of autonomous spacecraft fault protection. This work is ongoing, and current results suggest that more needs to be done in terms of analyzing and characterizing the structural properties of fault protection designs. Our goal is to identify typical patterns that may allow us to anticipate the feasibility of automated verification for future designs and suggest fruitful directions for abstraction techniques. We are taking steps in this direction by continuing our analysis of real-world fault protection designs.

7. Acknowledgments

The authors would like to thank Professor Edmund Clarke and his research group for their helpful comments and suggestions. We would also like to thank the other contributors to ExecSpec, including Bill Innanen, Lillian Nguyen, Mike Trela, Dan Wilson, Chris Olson, Chris Monaco and Russell Turner.

8. References


