# Creating Formal Characterizations of Routine Contingency Management in Commercial Aviation

Natasha Neogi[1], Jon Holbrook[2], Daniel Greissler[3] and Sid Bhattacharyya[4]
*NASA Langley Research Center, Hampton, VA, 23666, USA*
*Florida Institute of Technology, FL*

## Abstract

The identification, modelling, and analysis of root causes of accidents and incidents dominate conventional safety management approaches. However, the effect of humans' safety-producing behavior on the overall resilience of the system is often neglected. Additionally, emerging aviation markets are giving rise to concepts of operation, such as urban air mobility and optionally piloted air cargo operations, that are leading to a shift in locus of control between humans and automation. Without an understanding of the human contribution to safety, it is difficult to assess the effects of these novel role allocations on overall system safety. In this work, safety-producing behaviors are identified and abstracted into resilient performance strategies. Production rules that encapsulate these strategies are then generated and classified in the Soar cognitive architecture. The strategies are then applied to a remotely-operated air cargo example to demonstrate how safe learning is facilitated. The learned rules and strategies are then formally verified.

## I. **Introduction**

Traditional approaches to safety management focus on collection of data describing unwanted states (i.e., accidents and incidents) and analysis of undesired behaviors (i.e., faults and errors) that precede those states. Thus, in the traditional view, safety is both defined and measured by its absence, namely the lack of safety. In extremely high confidence systems like commercial air transport, opportunities to measure the absence of safety are relatively rare. Ironically, a critical barrier to measuring safety and the impact of mitigation strategies in commercial aviation is the lack of opportunities for measurement.

While traditional approaches to safety that focus only on minimizing undesired outcomes have proven utility, they represent an incomplete view of safety in complex sociotechnical domains such as aviation. For example, pilots and controllers successfully manage contingencies during routine, everyday operations that contribute to the safety of the national airspace system. However, events that result in successful outcomes are not systematically collected or analyzed. Characterization and measurement of routine safety-producing behaviors would create far more

[1] Research Scientist, Safety Critical Avionics Systems Branch, NASA Langely, AIAA Associate Fellow.
[2] Research Scientist, Crew Systems and Aviation Operations Branch, NASA Langley.
[3] Undergraduate Student, Department of Computer Engineering and Sciences, Florida Institute of Technology.
[4] Assistant Professor, Department of Computer Engineering and Sciences, Florida Institute of Technology.

opportunities for measurement of safety, potentially increasing the temporal sensitivity, utility and forensics capability of safety assurance methods that can leverage these metrics. Additionally, emerging commercial aviation markets may encompass novel roles and responsibilities for human machine teams, such as optionally-piloted air cargo operations. These operations could greatly benefit by being able to transition means and mechanisms for creating the desired safety producing behaviors previously evinced by the pilot into the new human machine teaming paradigm.

## II. **Capturing Pilot and Controller Strategies to Manage Contingencies**

The current study describes an initial effort to characterize how pilots and controllers manage contingencies during routine everyday operations and specify that characterization within a cognitive architecture that can be transformed into a formal framework for verification. Rather than focus on rare events in which things went wrong, this study focused on frequent events in which operators adjusted their work to ensure things went right. Namely, this study investigated how operators responded to expected and unexpected disturbances during Area Navigation (RNAV) arrivals into Charlotte Douglas International Airport (KCLT). Event reports submitted to NASA's Aviation Safety Reporting System (ASRS) that referenced one or more of the KCLT RNAV arrivals were examined. The database search returned 29 event reports that described air carrier operations on one of the RNAV arrivals. Those 29 event reports included 39 narratives, which were examined to identify statements describing safety-producing performance using the Resilience Analysis Grid (RAG) framework [1]. The RAG identifies four basic capabilities of resilient performance: anticipating, monitoring for, responding to, and learning from disruptions. Analysis of the 39 ASRS narratives revealed 99 statements describing resilient behaviors, which were categorized to create a taxonomy of 19 resilient performance strategies, summarized in Table 1 [2].

**Table 1.** Identified resilient performance strategies employed in routine aviation contexts.

| | Strategy | Description |
|---|---|---|
| **Anticipate** | Anticipate procedure limits. | Predict when current context inhibits normal use of a procedure, regulation, policy, norm. |
| | Anticipate knowledge gaps. | Predict whether crew member or other actor lacks required knowledge or information. |
| | Anticipate resource gaps. | Compare level of available resources (e.g., time, fuel, workload) to perceived resource needs. |
| | Prepare alternate plan and identify triggering conditions. | Have an actionable plan ready within the time available. |
| | Conduct pre-action briefing. | Discuss planned action and identify variables that might affect that plan. |
| **Monitor** | Monitor environment for cues signaling change from normal operations. | Identify triggering variables that signal something has changed from what was expected. |
| | Monitor environment for cues signaling need to adjust or deviate from current plan. | Identify triggering variables that signal something will not continue to work as planned. |
| | Monitor own internal state. | Self-assess physiological state, emotional state, workload, knowledge. |
| **Respond** | Adjust current plan to accommodate others. | Help others in system by changing timing or other action. |
| | Adjust or deviate from current plan based on risk assessment. | Change plan based on monitoring of triggers associated with safety boundaries. |

| | | |
|---|---|---|
| | Negotiate adjustment or deviation from current plan. | Work with others to accommodate competing goals and come to mutually acceptable solution. |
| | Defer adjusting/deviating from plan to collect information. | Continue with current plan because acting without critical information may worsen situation. |
| | Manage available resources. | Preserve finite resources by adjusting controllable aspects of the situation. |
| | Recruit additional resources. | Obtain resources locally or externally. |
| | Manage priorities. | Change goals, task order, task content, or pace of operation to accommodate resource limitations. |
| **Learn** | Leverage experience and learning to modify or deviate from plan. | Compare formal expectations and experience to current situation to develop real-time assessment of acceptability or risk. |
| | Understand formal expectations. | Understand applicability of laws, procedures, policies, and cultural norms. |
| | Facilitate others' learning. | Share information with others to increase their immediate understanding and long-term learning. |
| | Conduct after-action debriefing. | Discuss performance after mission has concluded to foster understanding and identify opportunities for improving future performance. |

## III. Modeling of Strategies in a Cognitive Framework

The strategies in this taxonomy can be classified, tagged and then formally described as scenarios that lead to either the preservation or degradation of safety. Cognitive architectures have traditionally been used to model aspects of human-machine interaction, specifically in a teaming context, where both agents may be learning as the activity progresses. Formal analysis of the previously identified strategies and their learning components generates confidence in their ability to be modelled and implemented in increasingly autonomous systems.

Cognitive architecture-based production systems are a popular method in Artificial Intelligence for producing intelligent behavior that is understandable to the program operator. Common rule-based reasoning systems include the General Problem Solver (GPS) [3], the Adaptive Control of Thought-Rational Theory (ACT-R Theory) [4] and the Soar cognitive architecture [5]. Rule-based expert/reasoning frameworks facilitate the representation of knowledge in the form of rules. Learning in such rule-based systems occurs by creating new rules, fine tuning the parameters in the rule or modifying rule order. Creating new rules can occur by identifying a new set of conditions under which an action should occur (called new rule learning) or by combining a number of rules together to form a single rule that makes execution of the set more efficient (called chunking). Rule order can be modified through reinforcement learning, which optimizes rule order based on a criterion such as minimum number of rule firings to reach a goal or minimizing overall time to execute [6].

The majority of these cognitive architecture frameworks express agent behavior in a non-formal fashion, which does not lend itself to assurance activities. The ability of complex, increasingly autonomous aviation systems to proliferate in the safety-critical domain of contingency management will depend on being able to verify and validate their behavior with the correct level of confidence. So, the design and development of assurance methods for intelligent contingency management is a necessity for these algorithms to be deployed in safety-critical operations [7]. Thus, we focus on the formal verification of the knowledge base by explicitly identifying human-specific strategies, then representing and investigating the change in knowledge that occurs due to learning in human oriented settings.

## A. Soar Cognitive Architecture

Several rule-based reasoning systems were surveyed as candidates for modeling human-automation interactions [3, 4, 5, 8]. Soar was selected because it encompasses multiple memory constructs (e.g., semantic, episodic, etc.) and learning mechanisms (e.g., reinforcement, chunking etc.). Soar production rules are expressed in first-order logic, which makes them amenable to verification. Finally, Soar is a programmable architecture with an embedded theory. This enables the execution of Soar models on embedded system platforms and allows the study of the design problem through rapid prototyping and simulation.

Every Soar production rule starts with the symbol *sp*, which stands for Soar production. The remainder of the rule body is enclosed in braces. The body consists of the rule name, followed by one or more conditions expressed in first-order logic (FOL), then the symbol →, which is followed by one or more actions (also in FOL). In Soar, a state variable (expressed as <variable>) can have multiple features or attributes, where features or attributes are indicated by the symbol ^. An attribute can take on a value, which is stated in the string following the attribute. So, the Soar expression: (<*s*> ^superstate nil) means that the state variable *s* has a feature, called superstate, whose value is nil [7]. An example Soar rule is:

sp{*proposeInitialize*(state <*s*> –^*name* ^*superstate nil*)→

(<*s*> ^*operator* <*o*>)(<*o*> ^*name initialize*)}

The Soar rule *proposeInitialize* has the condition where the state variable *s* has the attributes *name* (whose value is unassigned) and *superstate* (whose value is *nil*). The Soar feature *superstate* is an internal mechanism that Soar can use as part of its processing of goal-subgoal hierarchies. In this case, the precondition is that no superstate exists and that there is no pre-existing name for the state <*s*>. The right hand side (RHS) of the rule is the post condition or action, which indicates that given the left hand side (LHS) is true, an operator <*o*> is associated with the state <*s*>, and that an attribute of the operator is its *name*, which has a value *initialize*. Soar production rules are held in Soar's long term memory structures.

Soar production rules commonly execute in pairs of *propose* and *apply* rules. However, in Soar, the definition of an operator is distributed across multiple rules. A *propose* rule is one that proposes an operator. The *propose* rule creates a data structure in working memory representing the operator along with an acceptable preference. This preference encapsulates the ordering in which the operator is considered for selection. Similarly, an *apply* rule acts to apply the operator by making changes to working memory that reflect the actions of the operator. These changes may be purely internal or may initiate external actions in the environment. Thus, the *propose* rule checks which Soar production rules are eligible to be executed, and the corresponding *apply* rule executes one of the eligible rules.

At the lowest level, Soar's processing consists of matching and firing rules. Rules provide a flexible, context-dependent representation of knowledge, with their conditions matching the current situation and their actions retrieving information relevant to the current situation. Most rule-based systems choose a single rule to fire at a given time, and this serves as the locus of choice in the system – where one action is selected instead of another. However, there is only limited knowledge available to choose between rules, namely the conditions of the rules, the data matched by the rules, and possibly meta-data, such as a numeric score, associated with the rules. Soar allows additional knowledge to influence a decision by introducing operators as the locus for choice and using rules to propose, evaluate, and apply operators. In Soar, rules are able to fire in parallel.

Soar currently consists of long term memory (LMT), which is encoded as production rules, and working memory (WM), which is encoded as a symbolic graph structure so that objects can be represented with properties and relations. Symbolic working memory holds the agent's assessment of the current situation derived from perception and via retrieval of knowledge from its long-term memory. Action in an environment occurs through creation of external communication commands (e.g., motor commands, etc.) in a buffer in short-term memory [9, 10]. The decision procedure selects operators and detects impasses, which are described in the next section.

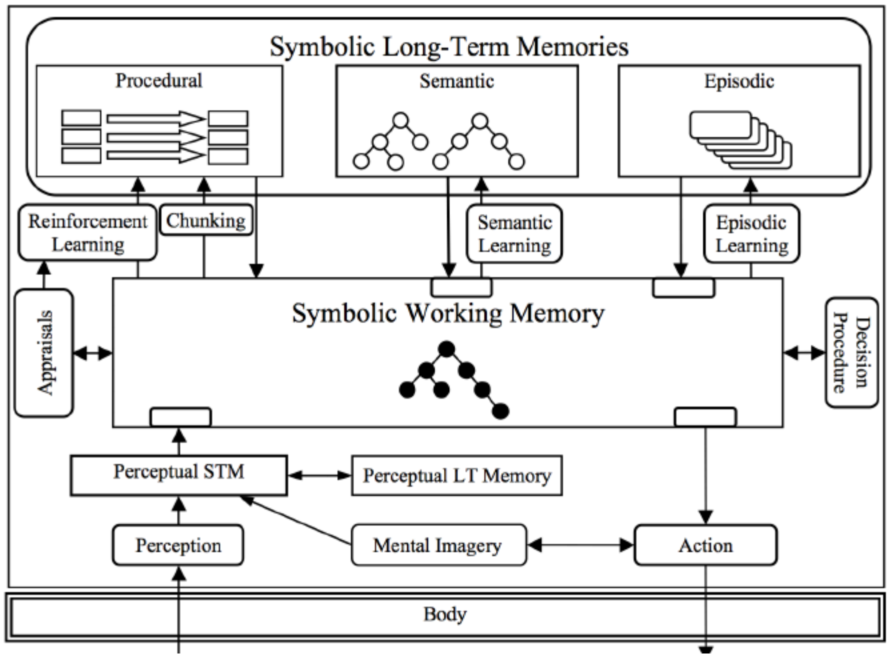

**Figure 1: Memory Structures in Soar (modified from [9])**

## IV. Decision Making in Soar

A decision cycle is a fixed processing mechanism in the Soar architecture that proceeds in five phases: *input, elaboration, decision, application,* and *output*. During input, working memory elements are created that reflect changes in perception. During elaboration, the contents of working memory are matched against the "if" parts of the rules in long-term memory. All rules that match from procedural memory, fire in parallel, resulting in changes to the features and values of the state in addition to suggestions, or *preferences*, for selecting the current operator. As a result of the working memory changes, more rules may fire, which includes aspects from procedural, semantic or episodic memory (see Figure 1). Elaboration continues in parallel waves of rule firings until no more rules fire. Quiescence in elaboration, i.e., the absence of any further firings of LTM rules, signals the start of the decision phase. The introduction of preferences allows the architecture to collect all the available evidence for potential changes to the state before actually producing any change. When the preferences added to working memory are evaluated by a fixed architectural decision procedure, the decision procedure can be applied to the vocabulary of preferences, independent of the semantics of the domain [10, 11, 12].

*1. Impasses*

If the preferences for the operators are insufficient to specify the selection of a single operator, or there are insufficient rules to apply an operator, an *impasse* arises. An impasse is a situation under which progress cannot be made. There are four types of impasses that can arise from the preference scheme: (1) Tie Impasse, (2) Conflict Impasse, (3) Constraint-failure Impasse and (4) No Change Impasse. A tie impasse arises if the preferences do not distinguish between two or more operators that have acceptable preferences. If two operators both have best or worst preferences, they will tie unless additional preferences distinguish between them. A conflict impasse arises if at least two values have conflicting better or worse preferences (such as A is better than B and B is better than A) for an operator, and neither one is rejected, prohibited, or required. A constraint-failure impasse arises if there is more than one required value for an operator, or if a value has both a require and a prohibit preference. These preferences represent constraints on the allowable selections that can be made for a decision, and if they conflict, no progress can be made from the current situation, and the impasse cannot be resolved by additional preferences. A no-change impasse arises if a new operator is not selected during the decision procedure. There are two types of no-change impasses: state no-change and operator no-change [9,10].

In response to an impasse, a substate is created in working memory, with the goal being to resolve the impasse. Additional procedural knowledge can then propose and select operators in the substate to gain more knowledge, and either create preferences in the original state or modify that state so the impasse is resolved. Substates provide a means for on-demand complex reasoning, including hierarchical task decomposition, planning, and access to the declarative long-term memories. Once the impasse is resolved, all of the structures in the substate are removed except for any results. Soar's chunking mechanism compiles the processing in the substate which led to formulating these results into rules. In the future, the learned rules automatically fire in similar situations so that no impasse arises, incrementally converting complex reasoning into automatic/reactive processing [9, 10].

*2. Chunking*

Chunking is a learning mechanism that acquires rules from goal-based experience. Chunking converts the results of problem solving in subgoals into rules – compiling knowledge and behavior from deliberate to reactive. Although chunking is a simple mechanism, it is extremely general and can encompass multiple types of rule-based knowledge [13].

Chunking is a Soar learning mechanism that represents the conversion of problem-solving acts into long-term memory. The goal when a decision-making impasse occurs is to learn the conditions that led to the impasse, and learn rules that can reduce their occurrence. An impasse may occur when several operators have been proposed, but no knowledge is evoked to prefer one option to another. The decision procedure records in working memory the type of the impasse, and generates a subgoal for resolving that impasse. The conditions of the new production (i.e., chunk) that solves the subgoal consist of aspects of the working memory state just before the impasse, and the actions of the production consist of the new knowledge that resolved the impasse (for example, an assertion that one of the proposed operators is to be preferred to the other in the current situation). Upon encountering a similar situation in the future, the production will automatically match and retrieve the knowledge, avoiding the impasse [13].

## V. Impasse Resolution and Rule Preferences

### B. Preference Resolution

When multiple operators are eligible to fire at once, preferences are used to determine which operator should be selected. Soar supports the following operator preferences: (1) Required, (2) Acceptable, (3) Prohibit, (4) Reject, (5) Better-Worse, (6) Best, (7) Worst, and (8) Indifferent. During the decision phase, operator preferences are evaluated in a sequence of 8 steps (only 5 of which provide termination) [9, 10, 12].

**Algorithm 1: Preference Resolution in Soar**

```
Algorithm 1: Generate $O_p$ and IT from PreferenceArray(m,n,p) and $O_f$
REQUIRE $0<card(O_f)<m$
ENSURE $O_p \subseteq O_f$
IT = $\varnothing$
$O_p = \varnothing$
        Required Preference Test:  Checks for required candidates in preference memory and also constraint-
        failure impasses involving require preferences
    1.  IF $\exists$ $o \in O_f$ for which PreferenceArray(o) = Required
    2.      $O_p = O_p \cup o$
    3.  ENDIF
        Termination point 1
    4.  IF $card(O_p)=1$ $\wedge$ PreferenceArray($o \in O_p$) = Prohibit
    5.      IT ={Constraint Failure} and Terminate
    6.  ELSE
    7.      RETURN $O_p$ and Terminate
    8.  ENDIF
    9.  IF $card(O_p)>1$ $\wedge$ $\exists$ $o \in O_p$ such that PreferenceArray(o) = Prohibit
    10.     IT={Constraint Failure} and Terminate
    11. ELSE
    12.     IT={Constraint Failure}
    13.     RETURN IT, $O_p$ and Terminate
    14. ENDIF
        Acceptable Preference Test:  Builds a list of operators for which there is an acceptable preference in
        preference memory
    15. IF $\exists$ $o \in O_f$ where PreferenceArray(o) = Acceptable
    16.     $O_p = O_p \cup o$
    17. ENDIF
        Prohibited and Rejected Preference Test:  Removes the candidates that have prohibit or reject
        preferences in memory
    18. IF $\exists o \in O_p$ where PreferenceArray(o) =Prohibited $\vee$ Rejected
    19.     Op = Op \ o
    20. ENDIF
        Termination point 2
    21. IF $card(O_p)<1$
    22.     IT ={No Change} and Terminate
    23. ELSIF $card(O_p)=1$
    24.     RETURN $O_p$ and Terminate
    25. ENDIF
        Better or Worse Test: Removes any operator $o_j$ that is worse than any other operator $o_i$
    26. FOR i = 1 to $card(O_p)$
```

```
27.     FOR j = 1 to card($O_p$)
            IF PreferenceArray($o_j$)<PreferenceArray($o_i$)
                $O_p = O_p \setminus o_j$
            ENDIF
28.     END FOR
29. END FOR
    *Termination Point 3*
30. IF card($O_p$)<1
31.     IT = {Conflict}
32.     RETURN IT, $O_p$ and **Terminate**
33. ELSEIF card($O_p$)=1
34.     RETURN $O_p$ and **Terminate**
35. ENDIF
    *Best Test: If a candidate operator has a best preference, this test removes all other candidates that do
    not have a best preference. If there are no best preferences for any of the current candidates, no changes
    are made.*
36. IF $\exists\ o \in O_p$ where PreferenceArray(o) = Best
37.     $\forall o_j$ such that PreferenceArray($o_j$)$\neq$ Best
38.     $O_p = O_p \setminus o_j$
39. ENDIF
    *Termination Point 4*
40. IF card($O_p$)<1
41.     IT = {No Change} and **Terminate**
42. ELSEIF card($O_p$)=1
43.     RETURN $O_p$ and **Terminate**
44. ENDIF
    *Worst Test: Removes any operators that have a worst preference. If all remaining candidates have
    worst preferences or there are no worst preferences there is no effect.*
45. IF ($\exists\ o_i \in O_p$ where PreferenceArray(o) = Worst) $\wedge$ ($\exists\ o_j \in O_p$ where PreferenceArray(o) $\neq$ Worst)
46.     $\forall o_i$ such that PreferenceArray($o_i$) = Worst $\wedge$ ($i \neq j$)
47.     $O_p = O_p \setminus o_j$
48. ENDIF
    *Termination Point 5*
49. IF card($O_p$)<1
50.     IT = {No Change} and **Terminate**
51. ELSEIF card($O_p$)=1
52.     RETURN $O_p$ and **Terminate**
53. ENDIF
    *Indifference Test: Tests to see if remaining operators are mutually indifferent to one another*
54. IF ($\exists\ o_i \in O_p$ where PreferenceArray(o) = Indifferent) $\wedge$ ($\exists\ o_j \in O_p$ where PreferenceArray(o) $\neq$
    Indifferent) $\wedge$ ($i \neq j$)
55.     IT = {Tie}
56.     RETURN IT, $O_p$
57. ELSE select among set of mutually indifferent operators $O_p$ using predetermined scheme set by user in
    Soar (e.g., random, epsilon greedy etc.)
58. ENDIF
59. RETURN IT, $O_p$
```

Input to **Algorithm 1** is: (1) the set of current operators $O_f$ eligible to fire and (2) an $m$ x $n$ x $p$ preference array, where $m$ is the number of operators eligible to fire, $n$ is the cardinality of the set of multi-attribute variables over all operators eligible to fire, and $p$ is the cardinality of the set of values those variables can take on. The output consists of: (1) a subset of the candidate operators $O_p$, which is either the empty set, a winning operator, or a set of operators that may be conflicting, tied, or indifferent and (2) an impasse type *IT* (e.g., Constraint Failure, No Change, Conflict, Tie) [10].

An impasse is resolved when processing in a subgoal creates results that lead to the selection of a new operator for the state where the impasse arose. When an operator impasse is resolved, Soar has an opportunity to learn, and the substate (and all its substructure) is removed from working memory. For instance, a tie impasse can be resolved by productions that create preferences that prefer one option (better, best, require), eliminate alternatives (worse, worst, reject, prohibit), or make all of the objects indifferent (indifferent). A conflict impasse can be resolved by productions that create preferences to require one option (require), or eliminate the alternatives (reject, prohibit). A constraint-failure impasse cannot be resolved by additional preferences, but may be prevented by changing productions so that they create fewer require or prohibit preferences. A substate can resolve a constraint-failure impasse through actions that cause all but one of the conflicting preferences to retract. A state no-change impasse can be resolved by productions that create acceptable or require preferences for operators. Finally, an operator no-change impasse can be resolved by productions that apply the operator, change the state so the operator proposal no longer matches, or cause other operators to be proposed and preferred [10].

We demonstrate how strategies can be leveraged to resolve impasses in the context of an unmanned aerial vehicle (UAV) undergoing a lost link procedure in the following section.

## VI. Translating Strategies into Soar For Learning

### C. UAV Lost Link Example

UAVs, including mid- to large-sized, are beginning to enter the national airspace. A primary cause of incidents and accidents related to these operations involve lost link events. These events include violations of assigned altitude clearances and unexpected heading changes [14]. For a UAV, or any aircraft for that matter, changing course or altitude without clearance can greatly increase air traffic controller workload and decrease safety. The contingency behaviors outlined in this paper's lost link procedures were designed following a review of existing UAV contingency management documentation (which included an MQ-9 flight manual and the Joint Unmanned Aircraft System Concept of Operations) and semi-structured interviews with UAV pilots [15]. These behaviors are described in Table 2 below.

**Table 2: Lost Link Contingency Behavior [15]**

| ID | Event | Environment | Behavior | Timer |
|---|---|---|---|---|
| C1 | Baseline | N/A | No Change | N/A |
| C2 | Lost Link | Populous to Lightly Populous | Return to Base | 1 min |
| C3 | Lost Link | Lightly Populous | Return to Base | 8 min |
| C4 | Lost Link | Unpopulated | Drop Altitude Maintain Course | 1 min |

Briefly, when the link is lost between the ground station and the UAV, a timer is set. If the overflown area is populous to lightly populous, a 1 minute timer is set and the UAV returns to base when it expires if the link does not return. Similarly, if the overflown area is lightly populous, an 8 minute timer is set and the UAV returns to base if the link is not re-established. Finally, if the overflown area is unpopulated, a 1 minute timer is set and the UAV drops altitude to try and re-establish link. After the timer expires, the UAV resumes mission altitude and continues its mission so long as the overflown area does not change. There is an overlap in C2 and C3 that occurs when the vehicle loses link over a lightly populated area. Since the goal state is to attempt to regain link and continue the mission, there is an opportunity to learn new rules and enhance mission success rates by developing multiple timers while flying over a lightly populated area under lost link conditions. The safety constraint limiting this learning behavior comes from the governing concern that the UAV does not overfly a populated area under lost link conditions for more than 1 minute [13].

## D. Strategies for Impasse Resolution and Learning

Soar allows for learning when impasses are found. While Soar will create a subgoal, work to resolve the impasse through preference resolution, and codify the means by which the resolution was achieved in a chunk, it is also possible to provide a ‘supervisory’ framework for learning by seeding the Soar rules with these strategies. This acts to guide the structure of the chunk to be learned.

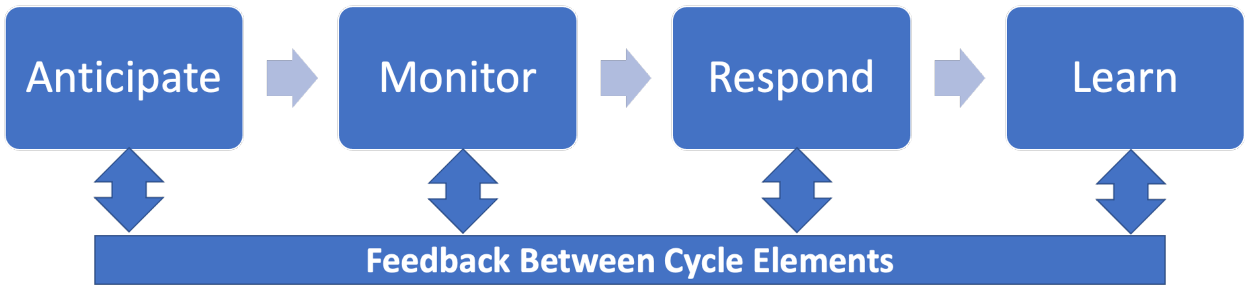

**Figure 2: Basis of Resilient Performance [16]**

The implementation of the resilient performance strategies in the Soar agent follows the structure outlined by Hollnagel [16], namely following the cycle of anticipate, monitor, respond, and learn (see Figure 2). There are ample instances in which the strategies identified in Table 1 are evinced in a concretized procedural rule.

The Soar agent begins by formally implementing the rules contained in the contingency plan (Table 2). However, through learning mechanisms employed during execution, the Soar agent has the ability to refine or alter the plan. That is, the Soar agent can recognize that lost link is a possible condition that it may encounter, and thus it has a baseline plan for what to do if the link is lost (as specified by the contingency conditions in Table 2). Soar recognizes that when the link is lost over different population densities, different actions should be taken. Thus, the Soar agent can identify the population density of the overflown area as a potential triggering condition to execute a given contingency. The relevant strategy can be found in Table 1, as *ANTICIPATE 4 – PREPARE ALTERNATE PLAN AND IDENTIFY TRIGGERING CONDITIONS*. A Soar rule corresponding to this strategy is shown below.

```
sp {drone*apply*outOfTimeMarkContingency
    "We are out of time so we need to pick a new contingency scenario and
mark the current one as complete"
    :o-support
    (state <s> ^name droneFlight ^startNewTimer yes ^maxTimers <mT>
^currentTimers {<cT> >= <mT>} ^addressedProblem <ap>)
    -->
    (<s> ^contingencyComplete <ap>
         ^startContingency yes)
    (write (crlf) |Marked as complete: | <ap>)
}
```

The Soar agent is continuously monitoring for changes in environmental conditions. The Soar agent uses sensor data in conjunction with a population density database and a projection of the agent's current state to determine whether the sensor data agrees with the projected population density of the overflown area. The relevant strategy that the Soar agent is using (from Table 1) is *MONITOR 1 – MONITOR FOR CUES SIGNALING CHANGE FROM NORMAL.* The Soar rule that encapsulates this strategy is stated below.

```
sp {drone*markInPopulatedArea
    "Removes timer start command and creates local willBeInPopulatedArea
wme"
    :o-support
    (state <s> ^name droneFlight ^io.output-link <out> ^io.input-
link.flightdata <fd>)
    (<fd> ^removeCommand <rc>
         ^willBeInPopulatedArea <wPA>)
    (<out> ^command <rc>)
    (<rc> ^name timerChecker)
    -->
    (<out> ^command <rc> -)
    (<s> ^willBeInPopulatedArea <wPA>)
    (<s> ^acknowledgedCommands timerChecker)
    (write (crlf) |Removing timerChecker command!|)
    (write (crlf) |Calculated WillBeInPopulatedArea as : | <wPA>)
}
```

The UAV receives the cue via projection that it will be overflying a populated area before the timer expires. The above rule proposes that the Soar agent create a variable in working memory (i.e., `^willBeInPopulatedArea`) that monitors whether or not the UAV will be in a populated area (either via sensor data or via projection of current state and population database knowledge). Additionally, Soar uses sensor data (e.g., internal vehicle health data) to monitor whether the lost link event has occurred. This corresponds to using the strategy *MONITOR 3 – MONITOR OWN INTERNAL STATE*. If the UAV has indeed lost link with the ground control station, Soar uses sensor data in conjunction with population database knowledge and the projection of the current state to determine changes in the timer status (i.e., since a lost link event has occurred, a timer has been set) and population density state (e.g., variable `^populated` changes value from lightly to fully). The relevant strategy for the Soar agent is *MONITOR 2 – MONITOR FOR CUES SIGNALING NEED TO ADJUST*. It is important to note that the Soar agent is capable of autonomously turning the vehicle around (via activation of the autonomous control agent). Example Soar rules for MONITOR 2 and 3 are omitted for brevity.

Given knowledge of lost link status (M1), how well encountered conditions map to projected conditions (M1), current population density state (M2), and timer status (M2), the Soar agent then adjusts the length of the timer and executes a contingency behavior (e.g., no change, drop altitude and attempt to re-establish link, return to base, etc.). The relevant strategy from Table 1 is *RESPOND 2– ADJUST BASED ON RISK ASSESSMENT*. For example, a Soar rule can be proposed that truncates the 8 minute timer if the projected overflown area is going to transition from lightly populated to populated within the next 2 minutes. The rule corresponding to this strategy is shown below.

```
sp {drone*apply*operator*C6-Lost-Link-Start-fully
    "In lightly populated area, should start a 2 minute timer and then
turn around"
    (state <s> ^name droneFlight ^startContingency yes -
^contingencyComplete C6-Start ^io.input-link.flightdata <fd>)
    (<fd> ^takeOver yes ^populated fully)
    -->
    (<s> ^operator <o> +)
    (<o> ^name C6-Start
         ^timerLength 2)
    (write (crlf) |PROPOSE C6!|)
}
```

It can be seen in the LHS of the rule that the variable `^populated` is either sensed or projected to take on the value 'fully' in the time horizon of the executing timer.

Additionally, Soar evaluates the efficacy of its responses. That is, the outcome of selecting different timers (e.g., 1, 2, 4, and 8 minute timers) is optimized over the flight trials that are flown. This leads to the addition of new contingency behaviors. Thus, a flight may encounter a lost link event over a lightly populated area and start a 4 minute timer, given learned behavior. This embodies the strategy *LEARN 1- LEVERAGE EXPERIENCE TO MODIFY PLAN*. Using this strategy, the Soar agent learned to create additional timers of 2 and 4 minutes. The chunk for the learned, 4 minute timer is shown below.

```
sp {chunk-1*d1471*tie*1
    :chunk
    (state <s1> ^operator <o1> +)
    (<o1> ^timerLength 4)
    (<s1> ^operator { <o2> <> <o1> } +)
    (<o2> ^timerLength 2)
    -->
    (<s1> ^operator <o2> > <o1>)
}
```

The Soar agent also learned to prefer shorter timers (e.g., 2 minute timer preferred above 4 minute timer, etc.). The chunk that encapsulates this preference is shown below.

```
sp {chunk-1*d1471*tie*1
    :chunk
    (state <s1> ^operator <o1> +)
    (<o1> ^timerLength 4)
    (<s1> ^operator { <o2> <> <o1> } +)
    (<o2> ^timerLength 2)
    -->
    (<s1> ^operator <o2> > <o1>)
}
```

This behavior arose due to the uncertainty in the projection of the populated area. The agent learned to minimize the risk of overflying a populated area for over a minute under lost link conditions by simply checking the condition of the overflown area every minute. While this seems like a trivial solution to the impasse between contingencies C2 and C3, it is actually quite subtle.

The preference that was learned was not actually related to the length of the timer, but for an inclination to check or monitor the variable that provided the cue that nominal operations could not continue any longer.

At this point, the agent cycles back to the beginning of the process (i.e., *ANTICIPATE 4 – PREPARE ALTERNATE PLAN AND IDENTIFY TRIGGERING CONDITIONS*), as the new timers enable the Soar agent to plan for additional contingency behaviors and then continues through the Monitor, Respond and Learn cycle (See Figure 2). This cycle repeats iteratively throughout the mission.

Finally, the Soar agent facilitates other agents' learning by logging its system state and environmental conditions each time the action is taken to turn the UAV around and turn to base. These data can be shared with other Soar agents performing similar operations, and the decision becomes part of a database that acts to improve decision making during future flights regarding what timers should be employed and when overflight of populated areas is imminent. These decision points thus act to refine contingency selection among all agents. The relevant strategy from Table 1 is *LEARN 3- FACILITATE OTHERS' LEARNING*. The Soar rule illustrating this strategy is shown below.

```
 sp {drone*apply*acknowledgeReverseAndSaveDecision
    "After reverse command has been served, send saved decision to be
cataloged"
    (state <s> ^name droneFlight ^io.output-link <out> ^io.input-
link.flightdata <fd> ^currentTimers <cT> ^maxTimers <mT> ^operator <o>
^addressedProblem <ap>)
    (<o> ^name saveDecision ^value <v> ^removeCommand <rc>)
    -->
    (<out> ^command <rc> -)
    (<out> ^command <com>)
    (<com> ^dref <ap>
           ^setValue <v>
           ^name saveDecision)
    (<s> ^maxTimers <mT> - ^currentTimers <cT> - ^addressedProblem <ap> -)
    (write (crlf) |Removing reverse command!  Sending saved decision! |)
}
```

## VII. Formal Verification of Chunks Learned via Strategies

The use of formal verification, including techniques such as model checking, enables the concrete specification of desired properties, as well as mathematically rigorous means by which to ensure them. Formal verification requires that these procedural strategies can be abstracted and translated into (possibly temporal) logic formulae, which serve as procedural rules in a knowledge database. We considered several formalisms such as NuSMV [17], Uppaal [18] and PVS [19] when considering the formal verification environment. We chose Uppaal [18], due to its ability to model temporal logic properties as well as its ability to generate and visualize counterexamples. Uppaal also allows the execution of requirements as temporal logic queries to exhaustively check the satisfaction of relevant safety properties. This translation procedure outlines the means by which a set of relational rules that represent the communal knowledge of the system (e.g., knowledge base) are captured and analyzed. The translation procedure has been implemented (see [7] for further technical details) and automatically translates Soar agents into Uppaal.

Because system safety is an emergent property, it is important to consider the robustness of the verification process, specifically the effect of environmental assumptions being violated, or unforeseen inputs being encountered. This relies on examining execution traces which generate the new elements of the knowledge base, and then evaluating them with respect to the strategies being captured. We focus our work on formally checking whether the knowledge base is consistent and create classes and subclasses which allows for generalization of a particular strategic instance. The verification process involves the generation of evolving traces from the timed automaton graphs which represent the procedural (Soar) rules in Uppaal [18]. Note that the process by which the Soar model is translated into Uppaal is detailed in [7, 14]. The evolving trace is then expanded based on a Breadth First Search (BFS), in order to encompass potential learning behavior, and an exhaustive search is performed, as the model is allowed to execute.

For this example, five requirements related to the contingencies were verified: (1) If the UAV loses ground station link, the autonomous agent starts a timer until the link is reconnected; (2) If the UAV loses ground station link in a lightly populated or populated area, the autonomous agent should start contingency 2; (3) If the UAV loses ground station link in a lightly populated area, the autonomous agent should start contingency 3; (4) When the autonomous agent has the variable 'takeover' with value 'yes' and the population is null, the autonomous agent starts contingency 4; (5) When the autonomous agent has the variable 'takeover' with value 'yes' and it is a turnaround emergency (e.g., 1 minute timer is expired and UAV over populated area), the UAV command is set to reverse the vehicle path. Further detail about the translation of these queries into temporal logic formula in Uppaal and their verification times can be found in [14].

A sixth requirement for verification was added relating to the resolution of impasses. Requirement 6 states that impasses caused by operator ties must be resolved via learned rules. This requirement was broken into two subparts: (6a) An operator tie occurs in an impasse, and (6b) Ties are resolved by selecting the best learned solution. In this case, learning occurred via chunking. Requirement (6a) was verified in Uppaal over the translation of the Soar model in an average of 1.21 seconds with a standard deviation of about 0.054. The maximally observed worst case execution time over 100 runs was 1.33 seconds. Similarly, (6b) had an average, standard deviation, and worst case execution time of approximately 20.85 sec, 1.19 sec, and 23.97 seconds respectively over the 100 runs. Note that the verification was performed on a MacBook Pro with a 2.8 GHz Intel Core i7 processor, and memory of 16GB 1600MHz DDR3. The verification was executed in the 64 bit Mac OS X version of Uppaal 4.1.24.

These results demonstrate the scalability of the approach. Since Soar agents have been used in a multi-UAV setting for path planning operations [20], a future direction for this work would involve extending the approach to handle multiple (potentially heterogeneous) agents simultaneously working towards a common goal.

## VIII. Conclusions and Future Work

In this work, several safety-producing behaviors were identified and abstracted into resilient performance strategies that were applicable to a remotely-piloted air cargo operation. These strategies aided in the classification of several pre-existing Soar rules in the example that resulted in safety producing behavior. Additionally, several strategies were encoded into Soar rules to assist the resolution of impasses. This resulted in learning on the behalf of the agent that was guided by these resilient strategies. The Soar agent's learned behaviour was not just to avoid an undesired state, but to adapt its functioning to facilitate desired states. As a result, the Soar agent's performance became more resilient. The Soar agent, including the learned rules and resilient performance strategies, was then formally verified. As future work, the creation of resilient strategy stereotypes in Soar and their application across multiple embedded, cyber-physical applications (e.g., automotive, medical, etc.) will be examined. This could facilitate novel role allocations between humans and increasingly autonomous agents in general contexts. Additionally, the use of resilient strategies in multi-agent settings will be examined. Specifically, the evaluation of the effects of resilient strategies on multi-agent teaming performance are an area of particular interest for human-machine teams.

## References

[1] Hollnagel, E. "RAG – The resilience analysis grid." In: E. Hollnagel, J. Pariès, D.D. Woods and J. Wreathall (Eds). Resilience Engineering in Practice. A Guidebook. Farnham, UK: Ashgate (2011).
[2] Holbrook, J., Prinzel, L.J., Stewart, M.J., Kiggins, D. "How do Pilots and Controllers Manage Routine Contingencies during RNAV Arrivals?" To appear in the Proceedings of the 11th International Conference on Applied Human Factors and Ergonomics (2020, July).
[3] Newell, A., Shaw, J.C., Simon, H.A., "Report on a general problem-solving program", In: Proceedings of the International Conference on Information Processing, pp.256–264 (1959).
[4] Anderson, J.R., Matessa, M., Lebiere, C., "ACT-R: A theory of higher-level cognition and its relation to visual attention", Hum.-Comput. Interact. (4), 439–462 (1997).
[5] Laird, J., The SOAR Cognitive Architecture, MIT Press (2012).
[6] Barto, R.S..B., Reinforcement Learning, MIT Press (2008.
[7] Bhattacharyya, S., Neogi, N., Eskridge, T., Carvalho, M., Stafford, M., "Formal Assurance for Cooperative Intelligent Agents", In: NASA Formal Methods Symposium, LNCS, vol. 10811 (2018).
[8] Buchanan, B.G., Shortliffe, E.H., "Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project", In: The Addison-Wesley Series in Artificial Intelligence., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1984).
[9] Lehman, J. F., Laird, J., Rosenbloom, P., "A gentle introduction to Soar, an architecture for human cognition", 2006 update (2006). https://web.eecs.umich.edu/~soar/sitemaker/docs/misc/GentleIntroduction-2006.pdf
[10] Laird, J., "The Soar 9 Tutorial", 2017 update (2017). https://soar.eecs.umich.edu/downloads/Documentation/SoarTutorial/Soar%20Tutorial%20Part%201%20-%20Simple%20Agents.pdf
[11] Wray, R.E., Jones, R.M. An introduction to Soar as an agent architecture. In: R. Sun (ed), Cognition and Multi-agent Interaction: From Cognitive Modeling to Social Simulation, Cambridge University Press, pp 53--78 (2005).
[12] Laird, J. E., "Extending the Soar Cognitive Architecture", In Proceedings of the First Conference on Artificial General Intelligence (2008).
[13] Steier, D., Laird, J.E., Newell, A., Rosenbloom, P.S., "Varieties of Learning in Soar", In: P. Langley (ed), Proceedings of the Fourth International Workshop on Machine Learning, Kluwer, (1987).
[14] Neogi., N., Bhattacharyya, S., Greissler., D., Kiran, H., Carvalho., M., "Assuring Intelligent Systems: Contingency Management for UAS", In: Y. Wan, E. Atkins, et. al. (eds), Special Issue of IEEE Transactions on Intelligent Transporation Systems Unmanned Aircraft System Traffic Management, In Review (2021).
[15] P. U. A. S. C. of Excellence, "Joint concept of operations for unmanned aircraft systems airspace integration." (2011).
[16] Hollnagel, E., Wears, R. L., Braithwaite, J., "From Safety-I to Safety-II: A White Paper", The Resilient Health Care Net. Published simultaneously by the University of Southern Denmark, University of Florida, USA, and Macquarie University, Australia (2015).
[17] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M.,Sebastiani, R., Tacchella, A., "NuSMV 2: An Open Source Tool for Symbolic Model Checking", Springer Berlin Heidelberg, pp. 359–364 (2002).
[18] Uppsala Universitet and Aalborg University, Uppaal, website: http://www.uppaal.org (2010).
[19] Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M., "PVS: Combining specification, proof checking, and model checking", Springer Berlin Heidelberg, pp. 411–414 (1996).
[20] Stenger A., Fernando B., Heni M. (2012), Autonomous Mission Planning for UAVs: A Cognitive Approach, Proceedings des Deutschen Luft- und Raumfahrtkongress, Berlin, 10.09.-12.09.2012, DLRK (2012).