Formal Assurance for Cognitive Architecture Based Autonomous Agent

Siddhartha Bhattacharyya, Thomas C. Eskridge, Natasha Neogi, and Marco Carvalho

Florida Institute of Technology,
School of Computing, Melbourne, FL
NASA,
Langley
{sbhattacharyya,teskridge,mcarvalho}@fit.edu
{natasha.a.neogi}@nasa.gov

Abstract. Autonomous systems are designed and deployed in different modeling paradigms. These environments focus on specific concepts in designing the system. We focus our effort in the use of cognitive architectures to design autonomous agents to collaborate with humans to accomplish tasks in a mission. Our research focuses on introducing formal assurance methods to verify the behavior of agents designed in Soar, by translating the agent to the formal verification environment Uppaal.

Keywords: formal verification, human-machine team assurance

1 Introduction

Autonomous systems are increasingly being designed in different modeling paradigms. Each of these focus on specific aspects of the design. For example, some methods are based on an architectural modular approach with rule based design, whereas others are focused on finite automata based models. In our research effort we focus on the design of autonomous agents based on the Soar cognitive architecture, which is a rule based system. The benefits of Soar are as follows:

- Allows for the representation and modeling of procedure oriented autonomous agents,
- Enables the modeling of human machine interactions.

These environments provide the paradigm to model autonomous agents, but lack the capability for rigorous analysis to prove the satisfaction of properties. In our approach, we discuss the initial version of the automated translator we have developed, and more specifically how this translation is achieved through a model transformation from Soar to Uppaal. This technique provides a proof of concept for how the formal assurance of cognitive models could allow verified agents to be designed for safety critical applications. In section 2, we discuss briefly the design of an autonomous agent in Soar. We then follow this with a

discussion of decision procedures in section 3, and elaborate on the production system executing rules in Soar. The formal verification environment Uppaal is discussed in Section 4, followed by a description of the automated translation process in Section 5. Finally we discuss the engine out case study in Section 6.

Previous work Research efforts in the area of verification of adaptive architectures include work done on Rainbow CMU [1], [2] which focuses on dynamic system adaptation via architecture models. It also investigates formal guarantees for synthesis of adaptive strategies. Research conducted by Topcu [3] focuses on constraining the inputs to learning systems in order to synthesize systems that are correct by construction. Sharifloo in [4] describes the use of light weight formal verification methods for runtime assurance when a system is updated with a new component. None of these efforts have provided support for rigorously analyzing existing adaptive decision procedures implemented through cognitive architectures. Thus, the approach in this paper is unique in performing formal verification of the production rules of a potentially learning system in a cognitive architecture.

2 Autonomous agents in Soar

Our investigation into the formal verification of adaptive intelligent systems focuses on agents constructed using cognitive architectures, such as Soar or ACT-R [5, 6]. Such agents can be used to develop new adaptive, intelligent systems for use in cybersecurity command and control [7, 8] or adaptive, intelligent processing [9].

In this research effort the Soar cognitive architecture is used to design an autonomous agent that performs the decision-making processes of a co-pilot of a large, multi-engine aircraft. We have modeled the co-pilot decision-making responsibilities for normal take off and single engine out procedures. The current research is based on earlier work developing Soar agents that modeled reinforcement-based learning of optimal item ordering of take-off checklists and of drone navigation in enclosed areas [10]. Figure 1 shows a block diagram of the latter work, where the Decision Agent could learn acceptable flight paths by watching the training paths flown by a human operator.

3 Production System Decision Procedures

Production systems have been a popular method in Artificial Intelligence for producing intelligent behavior that is understandable to the program operator [11, 12, 5, 6]. Productions systems represent knowledge as a set of "if-then" rules that map between states of the system and actions that can initiate sensing and take actions. For our drone navigation application, we developed a set of rules that navigated a quad-copter through a flight space with simple obstacles (i.e., regular, flat walls). The rules monitored the location and aircraft speed,

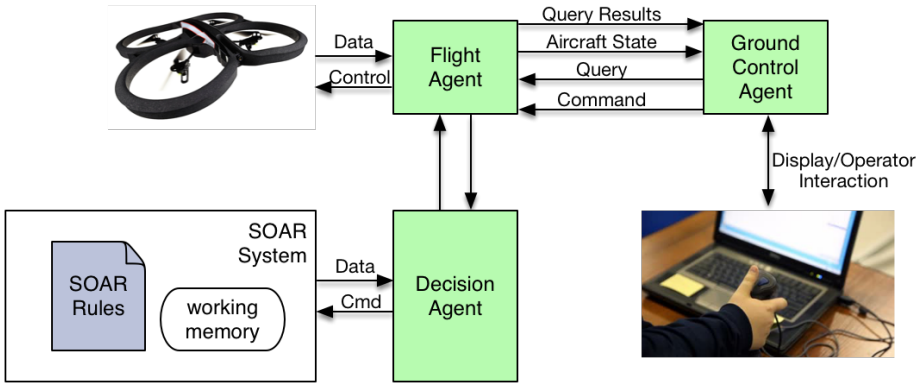

Fig. 1. Overview of the prototype system.

direction, and attitude on the "if" side of the rules (the condition variables), and made direction and attitude adjustments on the "then" side (the modified variables).

Adaptation in the Decision Agent occurs in two ways: First, a reinforcement learning procedure is executed to determine the best ordering of individual steps in a checklist [13]. The adaptation from reinforcement learning here does not alter the content of the rules, but instead alters the order of execution of the rules to result in the least overall time spent [14].

Second, Soar's "chunking" feature is used to collapse a number of rules that are commonly executed as a sequence into a single rule. This feature does modify the contents of the rules executed by the Decision Agent, and therefore introduces some uncertainty in the validity of the new rules.

The question that is answered by this research is "Is the ruleset that results from the adaptation valid?". That is, does it meet the progress and safety requirements demanded by the operator or mission, while improving its efficiency or performance?

4 Formal Verification: Uppaal

The goal of the translation is to map the cognitive model into a formal language, where progress and safety requirements can be verified completely. There are several options for the formal language. We have chosen to map to the formalisms of the language supported by Uppaal. Models in Uppaal are finite state machines. The composition of the rules as finite state machines allows the representation of formal modeling using temporal logics. This modeling paradigm allows the execution of requirements as temporal logic queries to exhaustively check the satisfaction of the properties. We describe the mathematical representation and the different properties that can be verified next.

4.1 Mathematical Representation in Uppaal

The rules are translated to formal, mathematically rigorous representations known as timed automata, a subset of hybrid automata. According to timed automata, one of the essential requirements in the design is to model the time associated with the execution of operations or rules. To represent time, the components need be modeled as timed automata. A timed automaton is a finite automaton extended with a finite set of real-valued clocks. Clock or other relevant variable values can be used in guards on the transitions within the automata. Based on the results of the guard evaluation, a transition may be enabled or disabled. Additionally, variables can be reset and implemented as invariants at a state. Modeling timed systems using a timed-automata approach is symbolic rather than explicit and is similar to program graphs. This allows verification of safety properties to be a tractable problem rather than an intractable infinite one with continuous time. So timed automata consider a nite subset of the innite state space on-demand, i.e., using an equivalence that depends on the property and the timed automaton, which is referred to as the region automaton.

Timed automata can be used to model and analyze the timing behavior of computer systems, e.g., real-time systems or networks. Methods for checking both safety and liveness properties have been developed. It has been shown that the state reachability problem for timed automata is decidable, which makes this an interesting sub-class of hybrid automata. Extensions have been extensively studied, among them stopwatches, real-time tasks, cost functions, and timed games. There exists a variety of tools to input and analyze timed automata and extensions, including the model checkers Uppaal, Kronos, and Temporal Logic Actions (TLA). These tools are becoming more and more mature. Formal representation of timed automata is defined below.

> Formally, timed automata can be defined as (Q, inv, , C, E, q0) where Q is finite set of states or locations inv are location invariants Σ is finite set of events or actions C is finite set of clocks E a set of edges, where an edge is a tuple (q, g, , r, q) defining a transition from state q to state q with a guard or clock constraint g, an action or event , and an update or reset r.q0 is the initial state or location

4.2 Formal verification of autonomous behaviors

The translation of a cognitive model to a formal methods environment supporting temporal logics allows the formulation of properties as described next.

- E<> p it is possible to reach a state in which p is satisfied, i.e., p is true in (at least) one reachable state (Figure 2 a).
- A[] p p holds invariantly, i.e., p is true in all reachable states (Figure 2 b).
- A<> p: The automaton is guaranteed to eventually reach a state in which p is true, i.e., p is true in some state of all paths (Figure 2 c).
- E[] p p is potentially always true, i.e., there exists a path in which p is true in all states (Figure 2 d).
- q − > p satisfaction of q eventually leads to p being satisfied (Figure 2 e)

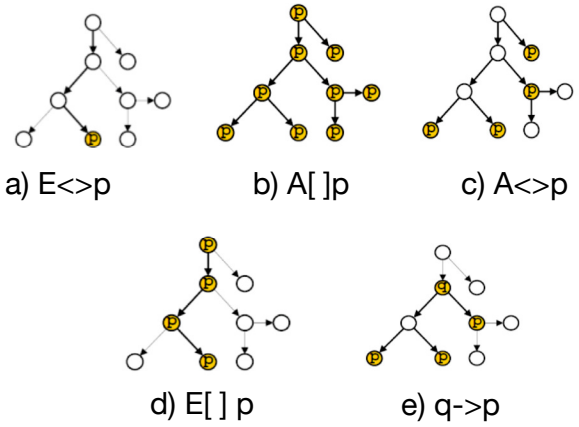

Fig. 2. Temporal formulas in Uppaal

5 Automated Translation from IA to FE

The process of translation from Soar to Uppaal captures our understanding of the differences in the cognitive model and its formal representation. We have automated the translation for a subset of the Soar models. The steps in the translation process include:

1. Developing the Soar grammar in Antlr
2. Generating the Soar parser
3. Storing the information from Soar into an intermediate data structure
4. Generating the xml for Uppaal

Figure 3 shows the sequential operations the translator goes through which are lexical analysis, semantic parsing, followed by symbolic and syntax analysis and then generating the Uppaal .xml file.

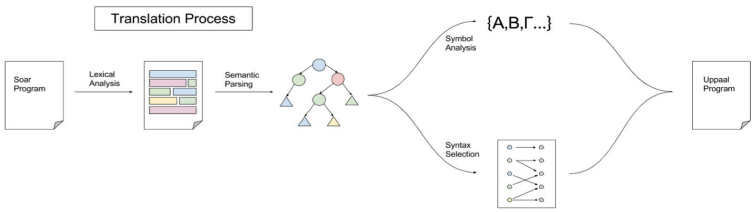

Fig. 3. Soar rule to initialize a variable

5.1 Soar model

An simple Soar rule is shown below in Figure 4 for a counting system (example counter). The example counter consists of three rules: the first rule initializes the values of the variables, the second rule increments the value of counter, and the third or goal rule checks if the final value has been reached. The rule in Figure 4 is the initialization rule for the counter. The left hand side (lhs) of the rule begins with the required Soar symbology *sp*, which stands for Soar production, along with the name of the production rule, followed by a pre-condition, for evaluation purposes. Soar has a superstate, which is an internal mechanism that Soar can use as part of its processing of goal-subgoal hierarchies. The condition where the superstate is nil and there is no operator indicates that Soar has just started and no processing has been done yet. This subgoal hierarchy capability is not used in the example, and therefore the superstate is only used to initialize the agent processing. So, 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, which indicates that, given the lhs is true, an operator is associated with the state <s> and that the name of the operator is *initialize counter*.

```
sp {counter*propose*initialize-counter
   (state <s> ^superstate nil
              -^name)
-->
   (<s> ^operator <o> +)
   (<o> ^name initialize-counter)
}
```

Fig. 4. Soar rule to initialize a variable

The grammar for Soar is input to Another Tool for Language Recognition (Antlr), in order to generate the Soar parser. Scripts from the Soar grammar are shown in Figure 5. Once the parser is created, it parses the Soar file to generate the graphical tree for a Soar rule. The graphical representation of the lhs of the above Soar rule is shown in Figure 4 and the rhs is shown in Figure 5. These two figures describe the tree structure associated with the Soar rule.

5.2 Automated Translation to Uppaal

DEFINITION 1: Rules in cognitive model (CM) are represented as a tuple rname(pre(), post()) where Pre() are the preconditions within a rule, Post() are the post conditions within a rule DEFINITION 2: The representation of a rule in a formal model is a tuple tname(S,Init,E,G,U) where:

– S: are states in the formal model

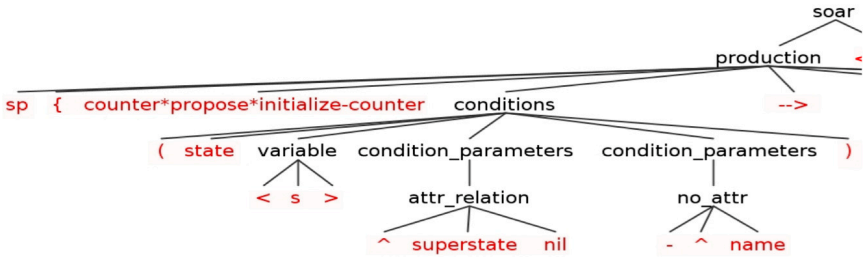

Fig. 5. Soar rule to initialize a variable

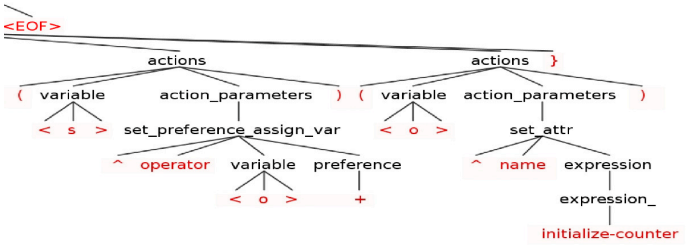

Fig. 6. Soar rule to initialize a variable

- Init: is initial state
- E SxS: represents the edges for transition
- G: represents the guard condition to be satisfied for the transition
- U: represents the reset value or actions or updates

Algorithm:

1. Translate individual rules rname(pre(),post())
   (a) Identify name of the rule in CM translate to template or process or component name in the formal environment (FE)
   (b) Precondition in CM translates to Guard in FE
      i. Identify the variables in CM in pre() create global variables in FE
      ii. Identify the logical comparisons, bindings in CMcreate similar logical equations and assignments in FE
   (c) Postcondition translates to updates
      i. Identify the variables and actions update the value of variables as the actions
2. Generate scheduler schd(s,,e,g)
   (a) Identify the lifecycle of execution
      i. Select rules based on satisfaction of pre conditions
      ii. Execute the rules
      iii. Execute the goal test

One of the challenges in the translation is that Uppaal doesn't support string data types. So any string comparison or pattern matching in Soar is translated into constant integers with the same name as the string in Soar.

Translated Uppaal model The Soar model and its corresponding Uppaal model is as shown in Figure 7 below. The name of the Soar model counter∗propose∗ initialize-counter in Uppaal is a template name. The preconditions in Soar states that no superstate exists and it does not have a name for the state s superstate nil and state sˆname is translated into a guard S_Superstate == nil and S_name == nil. The postcondition() state <s>ˆoperator<o> ˆname initialize-counter gets translated into s_operator_o_name = initialize_counter. For the first rule another update s_Superstate = not_nil is automatically added as Soar implements that behavior implicitly.

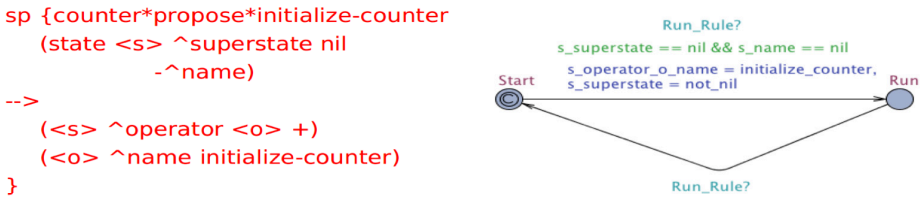

Fig. 7. Soar rule to initialize a variable

The general rule scheduler implements the process lifecycle. It contains three states: Start, Run and Check, as shown in 8. The transition from the start state to run state executes the initialization step for the system by broadcasting Run_Rule!. The initialization rule responds to it by initializing. The next step is execution of the goal rule. If the goal is not satisfied, the scheduler transitions from the Run to Check state on the guard condition, which is a negation of the goal guard. Then from the check state, it transitions to Run state by sending the broadcast Run_Rule! to which the relevant rules respond.

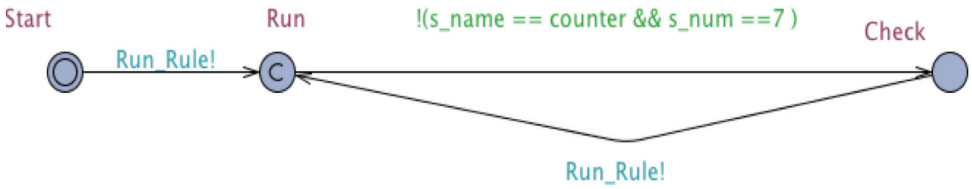

Fig. 8. Generic Scheduler

The scheduler is so designed such that it meets the following criteria:

- Configurable to meet different cognitive architecture
- The satisfaction of the precondition selects the rule to be executed

– Tests the goal condition to see if problem is complete

The property checked in the example of the simple counter was that all paths eventually terminated when the counter reached the value 7. Another example involved the design of a pilot agent in Soar that executes a sequence of tasks for its mission which involved: preflight checks, flight planning, filing the plan, launch, navigation, refuel, landing and arrival. The flight plan module in Uppaal is as shown in Figure 9. The properties checked for the pilot agent involved verifying that the pilot executed the appropriate sequences and also waited for completion of previous tasks before iteratively executing the next sequence of contingency tasks. The models translated from Soar generated counterexamples that indicated that the pilot agent could get into a refueling task before ever reaching a waypoint (and presumably landing). This was due to inadequately designed guards in the Soar agent.

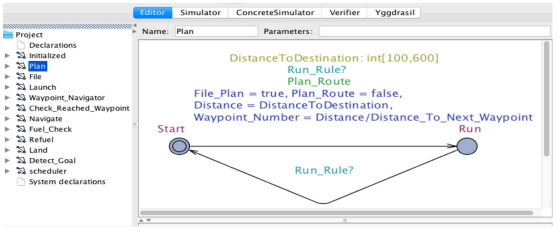

Fig. 9. Autonomous agent modules

Properties checked:

– All paths eventually lead to reaching destination state **A<> Goal.Goal**
– Does there exist a point when the next waypoint is not reached but the navigator says it has been reached **E<> t< Time_To_Next_Waypoint and Navigate_0.Run**
– Does there exist a condition where the next waypoint is not reached but the UAV is trying to refuel **E<>Waypoint_Navigator_0.Run and Refuel_0.Run**
– Does there exist a path where fuel is not checked **E[] Check_Fuel == false**

6 Example Case Study: Engine Out Contingency During Takeoff

The example used to illustrate this technique was that of an engine-out contingency upon takeoff. Conventionally defining the term pilot flying (PF) as the agent designated as being responsible for primary flight controls in the aircraft

(e.g., stick and surface inputs), and pilot not flying (PNF) as the agent not responsible for primary flight controls, has the Soar agent assuming the role of pilot not flying for this example. Thus, the Soar agent monitors procedure execution for off nominal or contingency situations, as well as performs secondary actuation tasks, similar to those performed by a copilot. However, it is important to note that the human pilot is always ultimately responsible for the overall safe execution of the flight [15].

For illustrative purposes, consider the scenario of a large cargo aircraft (such as a Boeing 737) during takeoff which experiences an engine failure, whereby the engine is delivering insufficient power after the aircraft brakes have been released, but before the aircraft takeoff has been successfully completed. Prior to takeoff, the speed $V1$ is calculated, which is defined by the FAA as "the maximum speed in the takeoff at which the pilot must take the first action (e.g., apply brakes, reduce thrust, deploy speed brakes) to stop the airplane within the accelerate-stop distance [16]. Thus, $V1$ is a critical engine failure recognition speed, and can be used to determine whether or not the takeoff will continue, or result in a rejected takeoff (RTO). $V1$ is dependent of factors such as aircraft weight, runway length, wing flap setting, engine thrust used and runway surface contamination. If the takeoff is aborted after the aircraft has reached $V1$, this will likely result in a runway overrun, that is, the aircraft will stop at a point in excess of the runway. Thus, $V1$ is also seen as the speed beyond which the takeoff should continue: the engine failure is then handled as an airborne emergency.

Prior to the takeoff procedure, a minimum of one person qualified to operate aircraft engines must be seated in a pilot's seat when an aircraft engine is started, or running. Prior to taking the active runway for takeoff, the PF performs the following actions, and briefs the PNF with respect to:

1. special factors influencing this takeoff (wet runway, anti-icing requirements, crosswind, deviations from the norm, etc.),
2. verification of airspeed settings (bugs) and power settings,
3. verification of navigation equipment setup,
4. verification of initial flight clearance (headings, altitudes, etc.), and
5. review of the emergency return plan.

Thus, there must be a shared situation awareness regarding the contingency plan at this point. In this standard briefing of the emergency return plan, the issue of engine failure is discussed. For takeoffs that experience any warning light or reason before 80 Knots-Indicated-Airspeed (KIAS), the takeoff is aborted [17]. After exceeding this lower threshold, but before attaining $V1$, the takeoff is only aborted in the case of engine fire or failure, thrust reverser deployment, aircraft control problems and warning conditions.

A conventional takeoff, whereby two humans fill the roles of the pilot flying and pilot not flying proceeds as follows. Both pilots review any changes in the ATC clearance prior to initiating the Before Takeoff (BT) checklist. All Before Takeoff checklist items must be completed before the takeoff roll commences. Once the checklist is completed, the following tasks are performed (see Table 1 below). Note that the aircraft parking brake must be released on the active

Task/Initiation Cue	PF	PNF
Takeoff		
Aircraft in position on the active runway. BT checklist completed and cleared for takeoff.	Hold brakes. Advance power to takeoff N1/Engine Pressure Rating, per AFM.	Call, "Power set, instruments stabilized." Monitor engines and systems indications.
PNF calls, "Power set, instruments stabilized."	Release brakes.	
Takeoff Roll		
Below 80 KIAS	Maintain directional control	Steady the control yoke with the right hand. (as applicable to aircraft type)
Positive airspeed indication		Call, "Airspeed alive"
PNF calls, "Airspeed alive."	Verify airspeed.	
At 80 KIAS		Verify 80 knots indicated on both PF and PNF airspeed indicators. Call, "80 knots cross-checked."
PNF calls, "80 knots crosschecked".	Move left hand from nose steering to control yoke and call, "My yoke". (as applicable to aircraft type)	
PF calls "My yoke". (as applicable to aircraft type)		Release control yoke. (as applicable to aircraft type)
At V1		Call, "V1."
PNF calls, "V1."	Move right hand to control yoke.	
At VR		Call, "Rotate."
PNF calls, "Rotate."	Rotate aircraft to pitch attitude per AFM.	

Fig. 10. Nominal Takeoff Procedure [17]

runway, and that the takeoff power (approximately 95 % $N1$, which is the revolutions per minute of the low pressure spool of the engine) must be set prior to attaining 60 KIAS.

During the Takeoff procedure, the PF and PNF perform the following sequential actions described in Figure 10, often waiting for a task initiation cue that comes from one another.

It can be seen that there is a great deal of interplay between the PF and PNF, especially in terms of affirming tasks and settings through callouts. These callouts also serve to initiate the subsequent task in the procedure. Thus, any tasks that are delegated to an automated PNF, performing the copilot role, must mimic this annunciation structure, in order to preserve situation awareness in the cockpit, and foster teamwork in the human-automation crew. Now, in the case of an engine failure at a speed of less than V1, but above the lower threshold speed of 80 kts, the actions shown in Figure 11 are taken.

Note that the contingency procedure is imbedded in the nominal procedure, and thus must be called from the nominal procedure.

Task/Initiation Cue	PF	PNF
Engine Out Rejected Takeoff		
Engine Out Detected		The PNF closely monitors essential instruments during the takeoff roll and immediately announce abnormalities or any adverse condition significantly affecting safety of flight. Call "Engine Fire", "Engine Failure" etc.
PNF calls, "Engine Failure" below V1 (and above 80 kts)	Call "Abandon" and take control of the aircraft.	
PF calls, "Abandon"	Close thrust levers and disengage simultaneously autothrottle.	Verify thrust levers close and autothrottle disengaged. Call out omitted action items.
Autothrottle disengaged	Verify automatic RTO braking (above 90 kts) or take maximum manual braking (below 90 kts) as required if deceleration is not adequate or if Autobrake Disarm light is illuminated.	Note the brakes on speed. Call "Autobrake Disarm" - Call out omitted action items.
PNF calls, "Autobrake disarm"	Raise speed brake lever.	Call "Speedbrakes Up" or Call "Speedbrakes Not Up"
PNF calls, "Speedbrakes Up"	Apply maximum reverse thrust consistent with runway conditions and continue maximum braking until certain airplane will stop on the runway.	Verify thrust reverser Call out speed: "100 Kts, 80 kts, 60 kts..."
PNF calls, "10 Kts"	Stop aircraft on runway heading or consider turning into wind if the takeoff was rejected due to fire warning.	At alternating red and white runway lights call "900 meters" of runway remaining. At steady red lights call "300 meters" of runway remaining
Aircraft stopped	Set Parking Brake	Select Flaps 40 when parking brake is set. Inform ATC including information on airplane position and alert if necessary the fire brigade

Fig. 11. Contingency Procedure For Engine Out on Takeoff [17]

6.1 Simulation and Experimentation with the Autonomous Pilot Agent

To test the Autonomous Pilot Agent in a number of different scenarios, we connected the commercial X-plane aircraft simulation [18] with a shim that reads the relevant aircraft state variables (e.g., speed, altitude, attitude, position) and injects them into Soar's working memory. The rules for normal takeoff or engine out takeoff monitor the state of working memory to ensure that the appropriate actions are taken when conditions warrant it. Figure 12 shows the connection

between the aircraft state variables and the Java-based Soar Autonomous Pilot Agent

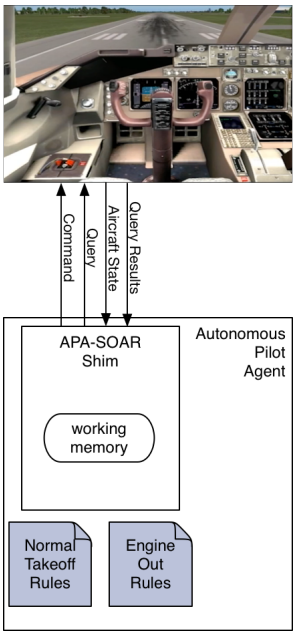

Fig. 12. Block diagram of the Simulation configuration

6.2 Formal Verification of the Autonomous Pilot Agent

The Soar rules for the autonomous agent that modeled the procedures followed by a copilot were translated into Uppaal. The rules executed based on inputs received from the flight simulator; this necessitated the creation of inputs such as airspeed. In order to provide changes to the airspeed, a new input template was created in which the airspeed was updated at every step of execution. This was followed by proving properties such as:

- airspeed greater than 80 is followed by applying rule for airspeed alive **R1 state_io_input_link_flightdata_airspeed == 80 −− > state_operator_name == callaa**
- All paths eventually lead to calling out Airspeed Alive **R2 A<> state_operator_name == callaa**
- All paths eventually lead to calling out rotate **R3 A<> state_operator_name == callrotate**

While attempting to formally verify the above properties in Uppaal, we encountered an out of range exception, as shown in Figure 13. This error was generated as follows: While Soar is executing in a busy waiting state, it is updating a counter variable without any bounds. It was never captured in the Soar environment, as events always occurred in the environment before the variable would overflow. But this unbounded variable overflow was captured while proving the property R1 in Uppaal. Hence, in periods where no events are taking place, it is possible for the copilot agent to *time out*, in some respects. Thus, translation of these rules provide a way for understanding these cognitive environments for designing safety critical autonomous agents.

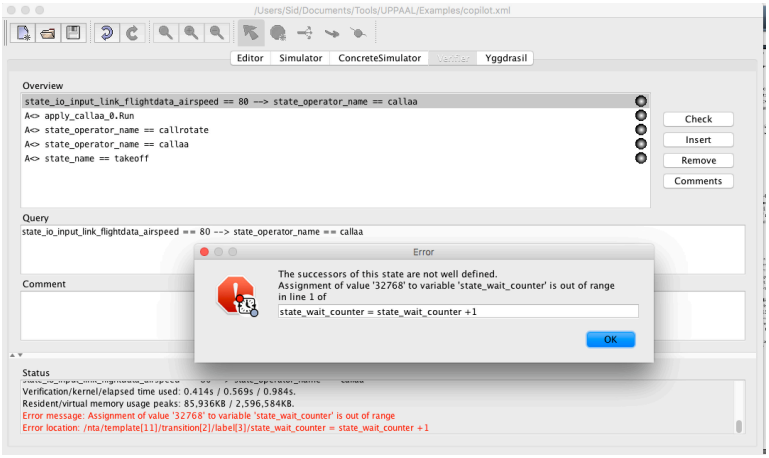

Fig. 13. Error

7 Conclusion and Future work

This research shows a method for the formal verification of intelligent agents designed with cognitive architectures that implement collaborative interactions between humans and autonomous systems. We have successfully developed an automated translator that translates a Soar agent into a formally verifiable Uppaal model. This enables formal verification of models developed in Soar. We plan to extend the translation to handle other constructs in Soar. This approach can be extended to the verification of other cognitive architectures.

References

1. Schmerl B. Aldrich J. Garlan D. Kazman R. and Yan Y. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7), 2006.

2. Garlan D. Cheng S. Huang A. Schmerl B. and Steenkiste P. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), 2004.
3. Topcu U. Wen M., Ehlers R. Correct-by-synthesis reinforcement learning with temporal logic constraints. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015.
4. Spoletini P. Sharifloo A.M. Lover: Light-weight formal verification of adaptive systems at run time. *Lecture Notes in Computer Science*, 7684:170–187, 2013.
5. J.E. Laird. *The SOAR Cognitive Architecture*. MIT Press, 2012.
6. John R. Anderson, Michael Matessa, and Christian Lebiere. Act-r: A theory of higher level cognition and its relation to visual attention. *Hum.-Comput. Interact.*, 12(4):439–462, December 1997.
7. M.M. Carvalho, T.C. Eskridge, L. Bunch, A. Dalton, R. Hoffman, J.M. Bradshaw, P.J. Feltovich, D. Kidwell, and T. Shanklin. Mtc2: A command and control framework for moving target defense and cyber resilience, 2013.
8. Marco Carvalho. Resilient command and control infrastructures for cyber operations. In *Proceedings of the 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 97–, Washington, DC, USA, 2015. IEEE Computer Society.
9. A. Granados and M.M. Carvalho. Overview of rails. Technical report, Florida Institute of Technology, 2015.
10. T.C. Eskridge, M.M. Carvalho, S. Bhattacharyya, and T. Vogl. Verifiable autonomy final report. Technical report, Florida Institute of Technology and Rockwell Collins, 2015.
11. Allen Newell, John C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing*, pages 256–264, 1959.
12. Bruce G. Buchanan and Edward H. Shortliffe. *Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (The Addison-Wesley Series in Artificial Intelligence)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.
13. R.L. Sutton & B. Barto. *Reinforcement Learning*. MIT Press, 2008.
14. T.C. Eskridge and M.M. Carvalho. Automated pilot model (apm) final report. Technical report, Florida Institute of Technology, 2015.
15. Natasha A. Neogi. Capturing safety requirements to enable effective task allocation between humans and automaton in increasingly autonomous systems. In *Proceedings of the AIAA Aviation Forum*. 16th AIAA Aviation Technology, Integration, and Operations Conference, 7 2016. AIAA 2016-3594.
16. Code of Federal Regulations. Title 14 aeronautics and space chapter 1 definitions and general requirements. Federal Register, 5 1962. http://www.ecfr.gov/cgi-bin/text.
17. The Boeing Company. Boeing 737 pilots operating handbook. Continental Airlines, 11 2002. http://air.felisnox.com/view.php?name=737.pdf.
18. Official x-plane website, http://www.x-plane.com, 2016.