Formal Assurance for Cooperative Intelligent Autonomous Agents

Siddhartha Bhattacharyya[1(✉)], Thomas C. Eskridge[1], Natasha A. Neogi[2], Marco Carvalho[1], and Milton Stafford[1]

[1] School of Computing, Florida Institute of Technology, Melbourne, FL, USA
{sbhattacharyya,teskridge,mcarvalho,mstafford2012}@fit.edu
[2] NASA Langley Research Center, Hampton, VA, USA
natasha.a.neogi@nasa.gov

Abstract. Developing trust in intelligent agents requires understanding the full capabilities of the agent, including the boundaries beyond which the agent is not designed to operate. This paper focuses on applying formal verification methods to identify these boundary conditions in order to ensure the proper design for the effective operation of the human-agent team. The approach involves creating an executable specification of the human-machine interaction in a cognitive architecture, which incorporates the expression of learning behavior. The model is then translated into a formal language, where verification and validation activities can occur in an automated fashion. We illustrate our approach through the design of an intelligent copilot that teams with a human in a takeoff operation, while a contingency scenario involving an engine-out is potentially executed. The formal verification and counterexample generation enables increased confidence in the designed procedures and behavior of the intelligent copilot system.

Keywords: Formal verification · Intelligent agents
Human-machine teams

1 Introduction

Autonomous systems are increasingly being designed to collaborate with humans to accomplish safety critical tasks. These cooperative agents are typically designed in modeling paradigms that emphasize human-machine interactions, efficiency, learning and performance improvement. However, operators mitigate safety and operational risks in an adaptive manner on a frequent basis, and the system often relies on this mitigation.

Our research focuses on the development of assurance for cooperative agents that can execute tasks autonomously, work with environmental variations and improve their own performance and the performance of the cooperative system overall. We translate the knowledge representations used by the cooperative agent into a formally verifiable representation to ensure that any modification to the agent behavior learned during operation will conform to the individual and system-level requirements specified during design and deployment. In this paper, we discuss an initial version of our automated translator and how this translation is achieved through a model transformation from an agent knowledge representation into a formally verifiable framework. Our technique provides a proof of concept for how the formal verification of autonomous agents could be utilized for safety critical applications.

Contribution. Frameworks for specifying autonomous cooperative agents usually lack the inherent capability for rigorous analysis, such as the formal verification and validation of safety critical properties. Our approach is novel in that it allows for the direct use of formal tools to verify safety properties of an autonomous agent designed in an existing cognitive architecture (Fig. 1). Thus, the formal verification process can be integrated into the design process of the agent. We believe this to be a unique contribution, which is, to the best of our knowledge, not currently otherwise utilized between other common intelligent agent frameworks (e.g., ACT-R, EPIC, PRS etc.) and formal methods (e.g., SMV, PVS, IC3 etc.).

We next review related work in Sect. 2. A target cognitive architecture and formal verification framework is selected in Sect. 3. We outline an example human-cooperative agent scenario involving an aircraft engine-out case study in Sect. 4. A formal description of the automated translation process between the cognitive architecture and verification formalism is provided in Sect. 5, along with implementation particulars. Section 6 discusses the results of the verification and validation efforts on the case study, and modifications made to the agent as a result. Conclusions and future work are detailed in Sect. 7.

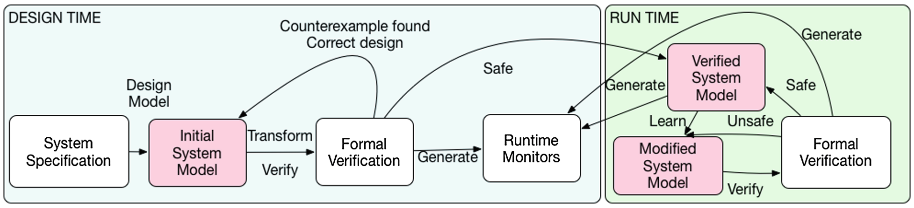


Fig. 1. System modeling and verification process

2 Related Work

Rule-based reasoning systems, or production systems, have been 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) [1], the MYCIN knowledge based inference system [2], the Adaptive Control of Thought-Rational Theory (ACT-R) [3] and the Soar cognitive architecture [4]. Rule-based reasoning frameworks facilitate adaptation in two ways: creating new rules and 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) [4]. Rule order can be modified through reinforcement learning, which optimizes rule order based on a criteria such as minimum number of rule firings to reach a goal or minimizing overall time to execute [5]. However, none of these modeling frameworks, which support learning behaviors, possess a formal semantics, and do not naturally support formal verification. Research efforts [6] extend ACT-R based on a discrete event simulation formalism, however, this does not encompass systems with continuous dynamics.

Research efforts in the area of verification of adaptive architectures for autonomy include work done on Rainbow CMU [7], which focuses on dynamic system adaptation via architecture models, along with potential formal guarantees. Research conducted by Wen [8] focuses on constraining the inputs to learning systems in order to synthesize systems that are correct by construction. Sharifloo and Spoletini [9] describes the use of lightweight formal verification methods for runtime assurance when a system is updated with a new component. In [10], Curzon discusses the development of a formal user model to develop a generic approach to cognitive architecture, but does not integrate this work with an existing architecture. Additionally, the formal verification of an autonomous system is discussed by O'Connor [11], and is based on the design of a mission controller with timed automata; however this does not enable the modeling of cognitive components. None of these efforts provide support for rigorously analyzing existing cognitive decision procedures implemented through an existing cognitive architecture.

NASA has developed PLan Execution Interchange Language (PLEXIL) [12], which has been successfully deployed for several autonomy applications. PLEXIL's operational semantics has been formally specified in the Prototype Verification System (PVS) [13] and properties such as determinsim and compositionality were mechanically verified in PVS [14]. Architecturally, the executable semantics in PLEXIL are specified in the rewrite logic engine Maude [15] for formal verification of the plans. Several efforts have focused on the use and analysis of PLEXIL, such as Strauss's efforts in analysing execution semantics in Haskell [16]. Balasubramanian et al. have developed Polyglot, a Statecharts analysis framework for PLEXIL, and are investigating formal analysis of a Statechart-based semantics of PLEXIL [17]. Verdejo and Martí-Oliet [18] have investigated the development of tools from the operational semantics specified in Maude. In our approach we utilize a similar paradigm of having the plans represented as rules in Soar, which are translated into Uppaal for verification. The additional benefit provided by our approach is that Soar has learning capabilities embedded within its architecture. The authors have also developed a navigation agent for drones in enclosed areas via the proposed methodology for direction selection [19]. In this paper, we extend this previous work by designing and implementing an autonomous agent that performs decision-making processes in the context of human teamwork. We next discuss the modeling formalisms used to support this work.

3 Modeling and Formal Verification

The goal of the research is to be able to model and ultimately design human and automated agent interactions, thereby enabling a shifting locus of control from human agents to automation in a variety of domains. This will entail automated agents performing complex safety-critical tasks in conjunction with human supervisors, which may result in possibly inimical emergent properties. In order to avoid this, we wish to map cognitive task models or operations into a formal language, where safety and liveness invariants can be verified. We considered several cognitive frameworks in order to model human interactions, as well as several formalisms for verification purposes. We detail the process of choosing a framework and formalism in Subsects. 3.1 and 3.2.

3.1 Cognitive Architectures and Frameworks: Soar

Cognitive architectures, which are often candidates for a general theory of cognition, can be regarded as architectures for the expression of intelligent behavior. We surveyed several rule based reasoning systems as candidates for modeling human-automation interactions [1–4]. Soar was selected based on its ability to encompass multiple memory constructs (e.g., semantic, episodic, etc.) and learning mechanisms (e.g., reinforcement, chunking etc.). Furthermore, Soar production rules are expressed in first order logic, which makes them amenable to verification. Finally, Soar takes the form of a programmable architecture with an embedded theory; this leads to the ability to execute Soar models on embedded system platforms, which enables the study of the design problem through the use of rapid prototyping and simulation.

Production Rules Expressed in the Soar Representation. 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, then the symbol →, which is followed by one or more actions (also expressed in first order logic). In Soar, a state variable (expressed as <variable>) can have multiple features or attributes, where features or attributes are indicated by the symbol ˆ. An attribute can take on a value, which is stated in the string following the attribute. So, the Soar expression: (<s> ˆ superstate nil) means that the state variable *s* has a feature, called superstate, whose value is nil. An example Soar rule is:

$$\begin{aligned} sp\{proposeInitialiaze(state <s> -\hat{}name \quad \hat{}superstate \quad nil) \rightarrow \\ (<s> \hat{}operator <o>)(<o> \hat{}name \quad initialize)\} \end{aligned}$$

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. The condition where the feature *superstate* holds the value *nil*, and there is no associated operator, indicates that Soar has just been invoked and no processing has been done yet. This subgoal hierarchy capability is not used in the example in Sect. 4, and therefore the superstate is only used to initialize the agent that is 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 or action, which indicates that given the 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 commonly execute in pairs of *propose* and *apply* rules. The *propose* rule checks which Soar production rules are eligible to be executed, and the corresponding *apply* rule executes one of the eligible rules. In Sects. 4 and 5, we used the Soar production system framework to encode rules describing takeoff procedures for an automated copilot in a commercial aircraft. The Soar framework facilitated development of the automated copilot agent. Furthermore, the first order logic representation of the production ruleset facilitates its translation into an appropriate modeling formalism for formal verification, as detailed in the next subsection.

3.2 Formal Languages and Verification: Uppaal

In order to choose the correct platform to translate Soar models into for the purpose of verification, several formalisms such as NuSMV [20], Uppaal [21], PVS [22] and Z3 [23] were considered carefully. We chose Uppaal [21,24,25], due to its ability to model timing aspects that are critical for cyberphysical systems, as well as its ability to generate and visualize counterexamples. Uppaal models are represented by timed automata, and the compositionality enabled by the Uppaal formalism supports model checking over networked timed automata using temporal logics. This modeling paradigm allows the execution of requirements as temporal logic queries to exhaustively check the satisfaction of relevant safety properties. We next describe the timed automata formalism used by Uppaal.

Mathematical Representation in Uppaal. Uppaal uses timed automata [26], a subset of hybrid automata, as a modeling formalism. One of the essential requirements in the design of human-agent teams is to be able to model the time associated with the execution of operations or rules. 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 automaton. 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 allows for the consideration of a finite subset of the infinite state space on-demand, i.e., using an equivalence relation that depends on the safety property and the timed automaton, which is referred to as the region automaton. There also exists a variety of tools to input and analyze timed automata and extensions, including the model checkers Uppaal, and Kronos [27]. For the purpose of this paper, we represent timed automata formally as follows.

Definition 1 ***Timed Automaton (TA).*** *A timed automaton is a tuple* (L, l_0, C, A, E, I)*, where* L *is a set of locations;* $l_0 \in L$ *is the initial location;* C *is the set of clocks;* A *is a set of actions, co-actions and the internal* τ*-action;* $E \subseteq L \times A \times B(C) \times 2^C \times L$ *is a set of edges between locations with an action, a guard and a set of clocks to be reset; and* $I : L \to B(C)$ *assigns invariants to locations.*

We define a clock valuation as a function $u : C \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^C$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. If we consider guards and invariants as sets of clock valuations (with a slight relaxation of formalism), we can say $u \in I(l)$ means that u satisfies $I(l)$. We can now define the semantics of a timed automaton as follows.

Definition 2 ***Timed Automaton (TA) Semantics.*** *Let* (L, l_0, C, A, E, I) *be a timed automaton* TA*. The semantics of the TA is defined as a labelled transition system* $\langle S, s_0, \to\rangle$*, where* $S \subseteq L \times \mathbb{R}^C$ *is the set of states,* $s_0 = (l_0, u_0)$ *is the initial state, and* $\to \subseteq S \times \{\mathbb{R}_{\geq 0} \cup A\} \times S$ *is the transition relation such that:*

1. $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$ *and*
2. $(l, u) \xrightarrow{a} (l', u')$ *if* $\exists e = (l, a, g, r, l') \in E$ *such that* $u \in g, u \; = [r \mapsto 0]u$ *and* $u' \in I(l)$*,*

where for $d \in \mathbb{R}_{\geq 0}, u + d$ *maps each clock* x *in* C *to the value* $u(s) + d$*, and* $[r \mapsto 0]u$ *denotes the clock valuation which maps each clock in* r *to* 0 *and agrees with* u *over* $C \setminus r$*.*

Note that a guard g of a TA is a simple condition on the clocks that enable the transition (or, edge e) from one location to another; the enabled transition is not taken unless the corresponding action a occurs. Similarly, the set of reset clocks r for the edge e specifies the clocks whose values are set to zero when the transition on the edge executes. Thus, a timed automata is a finite directed graph annotated with resets of, and conditions over, non-negative real valued clocks.

Timed automata can then be composed into a network of timed automata over a common set of clocks and actions, consisting of n timed automata $TA_i = (L_i, l_{i0}, C, A, E_i, I_i)$, $1 \leq i \leq n$. This enables us to check reachability, safety and liveness properties, which are expressed in temporal logic expressions, over this network of timed automata. An execution of the TA, denoted by $exec(TA)$ is the sequence of consecutive transitions, while the set of execution traces of the TA is denoted by $traces(TA)$. We next consider a simple flight example involving the interaction of human pilot with an autonomous copilot in a contingency situation where an engine becomes disabled during aircraft takeoff. We shall use this example to illustrate the process by which a Soar model is translated into Uppaal, and then we shall attempt to verify the design of the automated copilot model, with respect to simple safety and reachability properties.

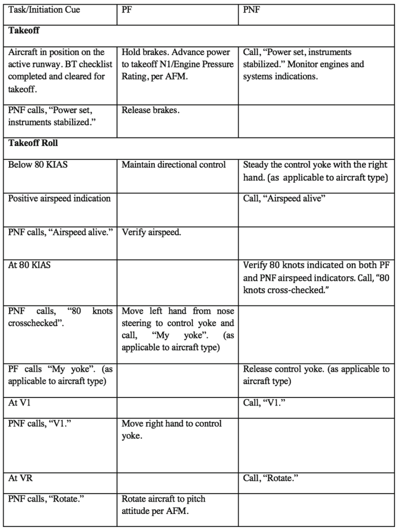
4 Example Case Study: Engine Out Contingency During Takeoff

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 PNF. 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 [28].

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" [29]. 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 on 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.

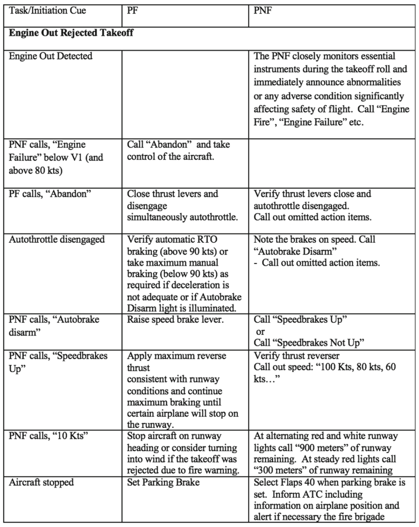
A conventional takeoff, whereby humans fill the PF and PNF roles, proceeds as follows. Both pilots review any changes in the ATC clearance prior to initiating the Before Takeoff (BT) checklist. All Before Takeoff checklist items must be completed before the takeoff roll commences. Once the checklist is completed, the Takeoff procedure is performed as detailed in Fig. 2. 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 team. In the case of an engine failure at a speed of less than $V1$, but above the lower threshold speed of 80 kts, the Contingency procedure shown in Fig. 3 is called from within the nominal Takeoff procedure.

These two operational procedures can be used to create a Soar production system, which models the behavior of the copilot, and can be executed, thereby creating an automated copilot function for takeoff. This process results in the creation of 15 Soar production rules. We now detail the process whereby the set of Soar production rules is translated into a network of timed automata, in Sect. 5.



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. 2. Nominal takeoff procedure [30]



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. 3. Contingency procedure for engine out on takeoff [30]

5 Automated Translation from Cognitive Architecture to Formal Environment

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 Soar to Uppaal translation for a subset of Soar models. This includes translation of conditions in a production rule represented with variables, operators, disjunctions, conjunctions as well as for action items adding new elements to working memory, and creating preferences for two among twelve preferences. Further support needs to be added for multi-attribute rules, creating the remaining ten preferences and implementing translation of other action items such as mathematical operations.

Figure 4 shows the sequential operations the translator goes through, which are (1) lexical analysis, (2) semantic parsing, (3) symbolic and syntax analysis and (4) generating the Uppaal .xml file.

Given the grammar describing the Soar productions, Another Tool for Language Recognition (ANTLR) was used to parse Soar, resulting in a syntax tree for further translation. From this tree, symbols—local variables used in productions—are extracted to add to the Uppaal model. Each Soar production can then be mapped to one or more Uppaal actions, which must then fire in a sequential fashion; meanwhile each Soar syntactical element must be mapped to the corresponding Uppaal element. Once the parser is created, it parses the Soar file to generate the graphical tree for a Soar rule.

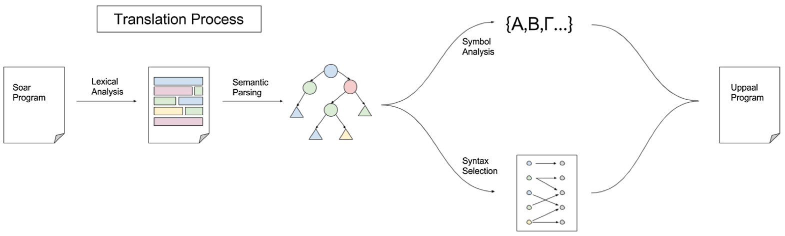


Fig. 4. Model transformation from Soar to Uppaal

In the Antlr grammar, the rule is parsed based on identifying if it is a *Soar rule* followed by if it is a *Soar production*. Once it is confirmed to be a Soar rule then the parser identifies if it is the LHS or RHS of the rule. If it is the LHS the *condition parameters* and the *variables* are identified. If it is the RHS of the rule, *action parameters* are identified along with the *variables*, *expression* and *preferences*. Once the trees are generated, the Soar rule is translated into a Timed Automaton (TA).

5.1 Automated Translation to Uppaal

In order to give a formal representation of the translation process, we consider the restricted subset of Soar production systems whose rules are represented using first order logic. We can then define a Soar production rule as a function of a finite set of variables $V \in v_i, i = 1, 2, 3 \ldots n$, whose valuation $val(V)$ represent the state of the system, along with a finite set of well formed formulae (WFF) $\Phi = \{\phi_1, \phi_2, \ldots \phi_m\}$, representing the left hand side of the Soar production rule (e.g., the preconditions), and a finite set of WFF $\Psi = \{\psi_1, \psi_2, \ldots \psi_r\}$, representing the actions embodied by the right hand side of the Soar production rule. We use the following formal definition for Soar production rules.

Definition 3. *An individual rule in cognitive model CM is represented as a tuple* $rname(V, Pre(\Phi), Post(\Psi))$ *where there are* $i = 1...n$ *variables,* m *well formed formulae in the precondition* $\{\phi_1(v_j), \phi_2(v_k), ...\phi_m(v_l)\}$*, and* r *well formed formulae in the postcondition* $\{\psi_1(v_s), \psi_2(v_u), ...\psi_r(v_w)\}$*.*

Each WFF (ϕ or ψ) may depend on a subset of the variables in V, as well as constants. Preconditions and postconditions can be formed through the use of first order logical operators (e.g., $\vee, \wedge, \forall$ etc.) over WFF. The execution or firing of a production rule creates an observable change in the system state, which can be denoted by $fire(rname)$. The goal is to map the semantics of a Soar production rule onto the semantics of a Uppaal TA. For the translation to be correct, we wish to have the behavior of the cognitive model be equivalent to the behavior of the network of TAs, at least with respect to the properties being verified. Each Soar production rule generates a TA in Uppaal, and the set of all TAs compose into a networked TA that corresponds to the cognitive model embodied by the Soar production rules. However, due to the nature of timing captured in Uppaal, we must also generate a scheduler (see Fig. 5), which forces a production rule and its corresponding TA to fire if one is available to do so. After any TA in the network has fired, the scheduler evaluates whether the goal state of the cognitive model has been reached. If this is not the case, the scheduler broadcasts the action Run_Rule, which causes an automata whose preconditions are enabled to fire. The Run_Rule action also allows the TA corresponding to the previously fired production rule to reset, rendering it immediately available to fire at the next evaluation of the TA network.

Specifically, each individual TA corresponding to a production rule has a *Start* state and a *Run* state, and a single clock x, whose value is given by $u(x) = val(x)$. Roughly speaking, the guard conditions for the TA correspond to the preconditions of the production rule. Similarly, the actions of the TA can be represented by the postconditions of the production rule. For example, at the start of the TA network execution, the guard condition for the initialization Soar production rule (Fig. 6) is the only rule that is true. This corresponds to the Soar representation where the guard condition, *superstate* is equal to *nil*, is true at initialization. Thus, the TA corresponding to the initialization rule executes when the scheduler sends out the Run_Rule broadcast shown in Fig. 5. During execution of the initialization rule, the values of variables on the RHS of the rule are updated, which changes guards for other rules to become true. After any TA executes, the scheduler transitions to the *Check* state on its own guard condition, which is a negation of the goal. If the goal is not met, the scheduler transitions from the *Check* to the *Run* state, and broadcasts the Run_Rule action to all TA, enabling further TA execution.

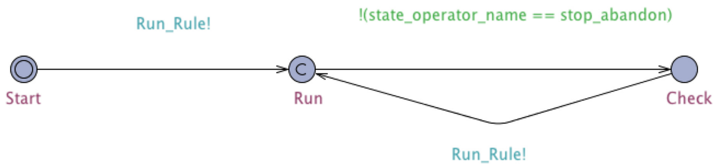


Fig. 5. Generic scheduler for timed automata derived from Soar rules

We briefly describe the algorithm as follows. The algorithm takes as its input a tuple $rname(V, Pre(\{\phi_1, \phi_2, ...\phi_m\}), Post(\{\psi_1, \psi_2, ...\psi_r\}))$, which is a rule from the Soar CM, and translates it to a timed automaton $TA = (L, l_0, C, A, E, I)$. The first line of the algorithm *requires* that all preconditions $Pre(\{\phi_1, \phi_2, ...\phi_m\})$ and postconditions $Post(\{\psi_1, \psi_2, ...\psi_r\})$ in the Soar rule be well formed formulae. It also requires that a *valuation* function $u(x)$ for the clocks x of the TA be defined over the non-negative reals. If the conditions specified in the requirements line are met, then the second line of the algorithm states that the property of the traces of the generated TA containing all the behaviors exhibited by the firing of the Soar rule is ensured.

Algorithm 1. Generate $(S, s_0, \rightarrow)$ for $TA = (L, l_0, C, A, E, I)$ from $rname(V, Pre(\{\phi_1, \phi_2, ...\phi_m\}), Post(\{\psi_1, \psi_2, ...\psi_r\}))$

```
Require: $\forall \phi(v_i), \psi(v_i) \in WFF$,, $u$: $C \rightarrow \mathbb{R}_{\geq 0}$
Ensure: $fire(rname) \subseteq traces(TA)$
  $l_0 \leftarrow \{Start\}$
  $L \leftarrow$ {Start, Run}
  $s_0 \leftarrow$ (Start, u($x_0$))
  $S \leftarrow$ {Start, Run} $\times u(x)$
  $I$(Start) = {True}
  $I$(Run) = {True}
  $G = \{\ \}$
  $A$ = {Run_Rule}
  for $j = 1$ to $m$ do
    $G \leftarrow G \wedge \phi_j$
  end for
  for $k = 1$ to $r$ do
    $A \leftarrow A \wedge \psi_j$
  end for
  if $u \in I$(Start) and $L$ = {Start} then
    $e_1$ = (Start, $A$, $G$, $u \mapsto u'$, Run)
  end if
  if $u' \in I$(Run) and $L$ = {Run} then
    $e_2$ = (Run, Run_Rule, {}, $u' \mapsto u''$, Start)
  end if
  $E = \{e_1, e_2\}$
  $S' \xleftarrow{u(x) \cup A} S$

  return $TA = (L, l_0, C, A, E, I)$
```

The next eight lines of the algorithm (lines 3–10) are used to initialize the elements of the TA and portions of its semantics, namely: (1) the set of initial locations l_0, (2) the set of locations L, (3) the set of initial states s_0, (4) the set of states S, (5) the invariant at the initial location, (6) the invariant at the location after the TA has executed its transition, (7) the set of actions of the TA, and (8) the set of guards for the TA. The Soar rule has two locations associated with it: (1) $Start = val(V(u))$, which describes the valuation of the state variables given in the Soar rule, before it has been fired, and (2) $Run = val(V(u'))$, which describes the valuation of the state variables given in rule after it has been fired. The state space is given by the cross production between the locations and clocks. There are no invariants at the locations, as currently non-determinism is handled in a sequential fashion, and the set of guards is initially set to empty. The set of actions for each TA created from a Soar rule has the base action *Run_Rule*, which is generated by the global scheduler, and used to guarantee execution. The *Run_Rule* action is received by all TA, and it forces all enabled TA (i.e. TAs whose guards are true, and thus, all Soar rules whose preconditions are true) to execute.

The first for loop (lines 11–13) is used to create the guard for the TA transition from the location *Start* to the location *Run*, and captures the *condition* portion of the Soar rule, by creating a conjunction of all of the preconditions (left-hand side) expressed by that rule. These are the preconditions that are needed for the Soar rule to fire, and are dependent on variables such as time. The second for loop (lines 14–16) is used to define the actions for the TA transition from the location *Start* to the location *Run*, and captures the *action* portion of the Soar rule, by creating a conjunction of all of the postconditions (right-hand side) expressed by that rule, along with the base action *Run_Rule*. These are the actions taken once the Soar rule fires (and the scheduler broadcast *Run_Rule* has been received), and act to change the state variables V, described in the Soar rule, of the system.

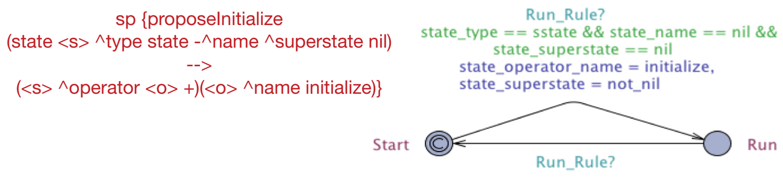
The next if statement (lines 17–19) creates the edge from the *Start* location to the *Run* location in the TA semantics. The origin and destination locations are *Start* and *Run* respectively. The set of actions A and guards G for this transition were defined by the previous two for loops. The clock is not reset on the edge, and thus its valuation advances from u to u'. Note that the conditional for forming the edge was that the initial clock valuation satisfied the invariant at the origin location of the edge. The following if statement (lines 20–22) creates the TA edge from the *Run* location to the *Start* location. This enables the TA to be reset after firing by the next broadcast of the *Run_Rule* action by the global scheduler. Thus, this mimics the behavior that the Soar rule is immediately able to be refired in the cognitive model. The origin and destination locations of the edge are *Run* and *Start* respectively. The *Run_Rule* action must be received for the reset to occur, and thus is the only action for the edge. There are no guards on this edge, and the clock is not reset, enabling time to progress from u' to u''. Lines 23–24 form the semantics for the TA by specifying the set of edges and transitions. Finally, in line 25, the algorithm returns the TA generated from the Soar rule $rname(V, Pre(\{\phi_1, \phi_2, ...\phi_m\}), Post(\{\psi_1, \psi_2, ...\psi_r\}))$.

We now walk through an example of this translation process as seen in Algorithm 1, for the rule given in Fig. 6.

5.2 Translation Implementation

The name of the Soar rule *proposeInitialize* (see Fig. 6) in Uppaal is a template name. To generate variable names, we linearize the working tree wherein each possible traversal ending in a useful value becomes the concatenated string of all visited identifiers during traversal (with an underscore as a delimiter). The preconditions in this Soar rule states that it: (1) does not have a name for the state *s^name* and (2) state *s^superstate* is *nil*. This is translated into the following guard: state_name == nil and state_superstate == nil. The action in this Soar rule is: state <s> ^operator <o> ^name initialize. This gets translated into the following TA action: s_operator_name = initialize.

The scheduler (Fig. 5) is designed to meet the following criteria: (1) It is configurable to meet different cognitive architectures, (2) Precondition satisfaction results in the selection of a rule to be executed, and (3) It tests the cognitive



```
sp {proposeInitialize
(state <s> ^type state -^name ^superstate nil)
-->
(<s> ^operator <o> +)(<o> ^name initialize)}
```

Run_Rule?
state_type == sstate && state_name == nil &&
state_superstate == nil
state_operator_name = initialize,
state_superstate = not_nil
Start
Run
Run_Rule?

Fig. 6. Mapping Soar to Uppaal

architecture goal condition to see if the goal has been reached. After the production rules have all been translated into timed automata, and the scheduler has been built, verification and validation activities can occur, as explained in the next section. Elements of non determinism due to learning (that changes rules) is challenging to translate into Uppaal as it is implicit in Soar.

6 Verification and Validation Efforts

6.1 Simulation Efforts for Validation of the Autonomous Pilot Agent

To test the Autonomous CoPilot Agent in a number of different scenarios, we connected the commercial X-plane aircraft simulation [31] 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 state variables are updated every 200 ms. The Soar agent captures the sensor value and then executes each rule within 100 ms. After experimenting with the simulation environment the change in velocity was set such that it changed every 150 ms with maximum acceleration. With this data sensor rate we minimize discrete jumps and attain a more or less continuous change in the value for the velocity. The rules for nominal takeoff and/or engine-out takeoff monitor the state of working memory to ensure that the appropriate actions are taken when conditions warrant. Figure 7 shows the connection between the aircraft state variables and the Java-based Soar Pilot Agent. This simulation was used to validate the autonomous copilot design for takeoff through multiple flown scenarios. Usability scenarios involved injecting engine faults at various times. Soar has an input/ouput link in its working memory which serves as the interface to different input output devices. The output branch for a Soar model has a memory element/node called speech. The autonomous agent adds children to this node; those children have literal string values. These values are sent to a text to speech engine as they appear thereby mimicking the interplay between the PF and PNF. The text to speech engine is a standalone Python application which is called by the Jsoar wrapper. These rules that initiate verbal interaction from the autonomous copilot are translated into Uppaal, with the verbal command translated as an update or action item for a variable representing the verbal communication.

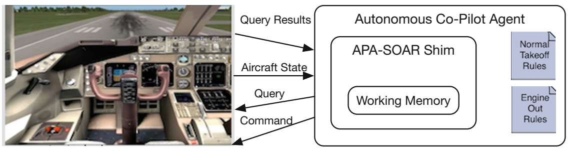


Fig. 7. Block diagram of the simulation configuration

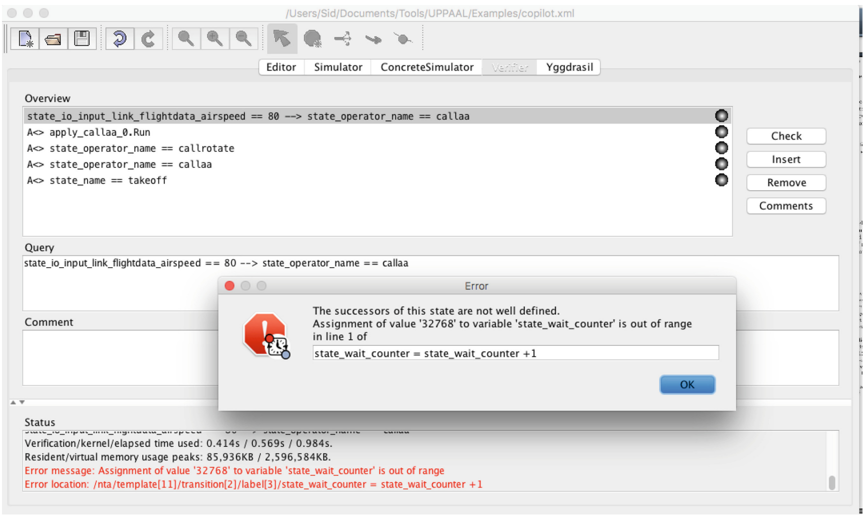
6.2 Formal Verification of the Autonomous Pilot Agent

The Soar rules for the autonomous agent that modeled the procedures followed by a copilot were translated into Uppaal. The rules executed are 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 within the Uppaal model in which the airspeed was updated at every step of execution. We used a first order model for vehicle velocity which we plan to refine. This was followed by proving properties in Uppaal 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
– TakeOff shall be abondoned if there is engine out and velocity is below the threshold
  R4 state_abandon == true $-->$ **applycallAbandon_0.Run**

The properties were proven on two sets of models. The first model represented the nominal Takeoff without any failures. The second model represented the procedures followed for the engine-out use case. While verifying the above properties such as R1, R2 and R3 in Uppaal on the first model, we encountered an out of range exception, as shown in Fig. 8. This error was generated due to the fact that at each cycle of the execution, Soar looks for a change in state caused either by new working memory data being input, or modifications to working memory data caused by rule execution. When there is no change in state it is called an Impasse, and Soar attempts to generate a sub-goal to continue to make progress towards a solution. The creation and resolution of sub-goals requires a hierarchical decomposition of the problem space, which is not necessary for straightforward examples. Instead, we introduce a counting mechanism which forces a change in state. This is effectively a busy-wait state for new data in Soar. The output of this counting can be unbounded and 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 on the nominal model. Hence, in periods where no events are taking place, it is possible for the copilot agent to *time out*, in some respects.

The properties verified on the off nominal model, such as R4, prove successfully indicating the copilot responds to the abandon event appropriately. Presently, the use case is determinsitic so uncertainity was not included in the model and translation, but both Soar and Uppaal allows probabilistic models, thus there is the potential to include uncertainity in our future research efforts. This will enable evaluating the efficiency of reordering the set of rules when a contingency arises. This also enables us to model uncertainty due to sensor observation and modeling approximation along with reaction times taken by human agents.



```
state_io_input_link_flightdata_airspeed == 80 --> state_operator_name == callaa
A<> apply_callaa_0.Run
A<> state_operator_name == callrotate
A<> state_operator_name == callaa
A<> state_name == takeoff
```

```
state_io_input_link_flightdata_airspeed == 80 --> state_operator_name == callaa
```

```
The successors of this state are not well defined.
Assignment of value '32768' to variable 'state_wait_counter' is out of range
in line 1 of
state_wait_counter = state_wait_counter +1
```

```
Verification/kernel/elapsed time used: 0.414s / 0.569s / 0.984s.
Resident/virtual memory usage peaks: 85,936KB / 2,596,584KB.
Error message: Assignment of value '32768' to variable 'state_wait_counter' is out of range
Error location: /nta/template[11]/transition[2]/label[3]/state_wait_counter = state_wait_counter +1
```

Fig. 8. Out of range error for translated Uppaal copilot agent

7 Conclusion and Future Work

Cognitive architectures have proven to be beneficial in the design of intelligent adaptive systems, as they allow the integration of learning algorithms as plug-ins within a defined architectural representation. Additionally, they allow representation of collaborative human-machine teaming by modeling the autonomous agents working with humans. Presently, systems designed with these cognitive architectures cannot be deployed for safety critical applications as the methods to assure correctness of their behavior are inadequate. A significant contribution of our work has been the development of an extensible framework for the design and formal verification of systems whose intelligent attributes can be modeled in a cognitive architecture. We have formally described and implemented the automated model transformation that translates an intelligent agent into a formally verifiable temporal logic based model. This translation enables formal verification of cognitive models developed in cognitive architectures or rule based systems such as Soar. We plan to extend this research to fully integrate multiple learning methods that are capable of modifying rules within the system and formally verifying the resulting system. We also intend to model uncertainty in the system through the incorporation of probabilistic models. We plan to evaluate merging Soar rules into one Uppaal template as otherwise the number of Uppaal templates become fairly large.

References

1. 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)
2. Buchanan, B.G., Shortliffe, E.H.: 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 (1984)
3. Anderson, J.R., Matessa, M., Lebiere, C.: ACT-R: a theory of higher level cognition and its relation to visual attention. Hum.-Comput. Interact. **12**(4), 439–462 (1997)
4. Laird, J.E.: The SOAR Cognitive Architecture. MIT Press, Cambridge (2012)
5. Sutton, R.L., Barto, B.: Reinforcement Learning. MIT Press, Cambridge (2008)
6. Mittal, S., Douglass, S.A.: Net-centric ACT-R based cognitive architecture with DEVS unified process. In: DEVS Symposium Spring Simulation Multiconference, Boston (2011)
7. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self adaptation with reusable infrastructure. Computer **37**(10), 46–54 (2004)
8. Wen, M., Ehlers, R., Topcu, U.: Correct-by-synthesis reinforcement learning with temporal logic constraints. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (2015)
9. Sharifloo, A.M., Spoletini, P.: LOVER: Light-weight fOrmal Verification of adaptivE systems at Run time. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 170–187. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35861-6_11
10. Curzon, P., Ruknas, R., Blandford, A.: An approach to formal verification of human computer interaction. Form. Asp. Comput. **19**, 513–550 (2007)
11. O'Conner, M., Tangirala, S., Kumar, R., Bhattacharyya, S., Sznaier, S., Holloway, L.: A bottom-up approach to verification of hybrid model-based hierarchical controllers with application to underwater vehicles. In: Proceedings of American Control Conference (2006)
12. Rocha, C., Cadavid, H., Muñoz, C., Siminiceanu, R.: A formal interactive verification environment for the plan execution interchange language. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 343–357. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_24
13. Dowek, G., Munoz, C., Pasareanu, C.: A small-step semantics of PLEXIL (2008)

14. Dowek, G., Munoz, C., Pasareanu, C.: A formal analysis framework for PLEXIL. In: Proceedings of 3rd Workshop on Planning and Plan Execution for Real-World Systems (2007)
15. Dowek, G., Munoz, C., Rocha, C.: Rewriting logic semantics of a plan execution language. In: EPTCS, vol. 18, pp. 77–91 (2009)
16. Strauss, P.J.: Executable semantics for PLEXIL: simulating a task-scheduling language in Haskell. Masters thesis (2009)
17. Balasubramanian, D., Pasareanu, C., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple statechart formalisms. In: Dwyer, M.B., Tip, F. (eds.) ISSTA. ACM (2011)
18. Verdejo, A., Martí-Oliet, N.: Two case studies of semantics execution in Maude: CCS and LOTOS. Formal Methods Syst. Des. **27**, 113–172 (2005)
19. Eskridge, T.C., Carvalho, M.M., Bhattacharyya, S., Vogl, T.: Verifiable autonomy final report. Technical report, Florida Institute of Technology and Rockwell Collins (2015)
20. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
21. Uppaal website (2010). http://www.uppaal.org
22. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.: PVS: combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_91
23. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
24. Larsen, K.G., Pettersson, P., Yi, W.: Model-checking for real-time systems. In: Reichel, H. (ed.) FCT 1995. LNCS, vol. 965, pp. 62–88. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60249-6_41
25. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL: a tool suite for automatic verification of real-time systems. Theor. Comput. Sci. (1996). RS-96-58
26. Alur, R., David, L.D.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1999)
27. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: a model-checking tool for real-time systems. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 298–302. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055357
28. Neogi, N.A.: 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 (AIAA 2016-3594) (2016)
29. Code of Federal Regulations: Title 14 Aeronautics and Space. Federal Register, May 1962. http://www.ecfr.gov/cgi-bin/text
30. The Boeing Company: Boeing 737 pilots operating handbook. Continental Airlines, November 2002. http://air.felisnox.com/view.php?name=737.pdf
31. Official x-plane website (2016). http://www.x-plane.com