ENHANCING AUTONOMY WITH TRUSTED COGNITIVE MODELING

S. Bhattacharyya,[*] J. Davis,[+] T. Vogl[#], M. Fix [@], A. McLean[#], M. Matessa[-] and L. Smith-Velazquez[-]

ABSTRACT

Autonomous Systems (AS) have been increasingly investigated for safety, aviation, medical, and military applications with different degrees of autonomy. Autonomy involves the implementation of adaptive algorithms (artificial intelligence and/or adaptive control technologies). Advancing autonomy involves transferring more of the decision-making to AS. For this transition to occur there has to be significant trust and reliability in the AS.

Exhaustive analysis of all the possible behaviors exhibited by autonomy is intractable. However, methods to guarantee behaviors of the intelligent agent relevant to the application need to be developed. One of the approaches we explain here is to translate from a cognitive architecture to a formal modeling environment. This approach enables developers to continue to use the tools (such as ACT-R or Soar) that are good at cognitive modeling, while gaining trust through the guarantees provided by formal verification. In this process, there are several challenges to be addressed regarding the interactions within the cognitive engine, maintaining the integrity of the interactions occurring based on the cognitive architecture, and addressing the compositional analysis of rules. We describe an approach in this paper.

INTRODUCTION

Autonomous systems (AS) have degrees of behaving independently and deciding what to do based on the situation. AS have been increasingly deployed for safety, aviation, medical, military, and other business-critical applications. Depending upon the domain of application, there have been different degrees of autonomy. Autonomy involves the implementation of adaptive algo-

[*] Sr. Research Engineer, Trusted Systems Group, Rockwell Collins, Cedar Rapids IA.
[+]Sr. Applied Mathematician, Trusted Systems Groups, Rockwell Collins, Cedar Rapids, IA
[#]Pr. Systems Engineer, Virtualized Flight Systems, Rockwell Collins
[@]Sr. Software Engineer, Precision Systems, Rockwell Collins
[-]Sr. Systems Engineer, VFS, Rockwell Collins

rithms based on artificial intelligence and/or adaptive control technologies. Some of these technologies are potentially useful for responding to unforeseen situations, but they can also be a liability for safety-critical situations; cognitive modeling in autonomy has the potential to lead to non-deterministic behavior that's difficult to be validated for situations not considered at design time or due to the evolving structure of autonomy due to learning. Enhancing autonomy leads to transferring more of the decision-making to AS. For this transition to happen there has to be significant trust and reliability in the AS. We describe a method to address some aspects of verification of AS by representing them within formal methods based frameworks where they can be verified. Application of formal methods is limited with the potential of not being able to guarantee certain behaviors due to out of memory or scalability issues. This is especially true with handling non-linear dynamics which is being thoroughly researched.

Exhaustive analysis and understanding of all the possible behaviors based on the adaptive algorithms implemented is challenging for large scale complex systems. However, methods can be developed to guarantee behaviors of an intelligent agent relevant to the application as the domain could constrain the possibilities and/or through abstraction. An intelligent agent is an agent that adapts and changes behavior based on the situation. The behavior-checks can be formalized to verify progress and safety. Verifying progress assures that the behavior modeled meets the mission goal. Checking safety verifies that the system never does anything harmful and never goes into an unsafe region. This involves exhaustive analysis of the models with appropriate refinements to diagnose safety violations.

One of the approaches to a process of developing and verifying a system is to design a Domain Specific Language (DSL [1] [2]) that provides a common input for the intelligent system as well as the formal modeling paradigm. This is then translated into each of the domains but the challenging issue is that this approach misses upon the structural flow of cognitive models which is different than the FM tools. The other approach which we explain here is to translate from a cognitive architecture to a formal modeling environment. This approach could enable developers to continue to use the tools (such as ACT-R or Soar) with which they are familiar, while gaining the guarantee provided by formal verification. In this process of translation, there are several challenges we need to address regarding the interactions within the cognitive engine, maintaining the integrity of the interactions occurring based on the cognitive architecture, and addressing the compositional analysis of rules. We describe this approach and the challenges within in the rest of the paper. This approach guarantees behaviors for example detect and avoid, follow lost communication procedures to guarantee safe operations as indicated in Figure 1 that AS avoid each other following the rules as outlined in pilot manuals. Models have been developed demonstrating this approach in the Soar cognitive architecture and the Uppaal model checker. The novelty of this approach is in the translation of cognitive models to a formal analysis environment to gain confidence in the correctness of the system. This approach could be automated and implemented, for example, as an Eclipse plug-in.

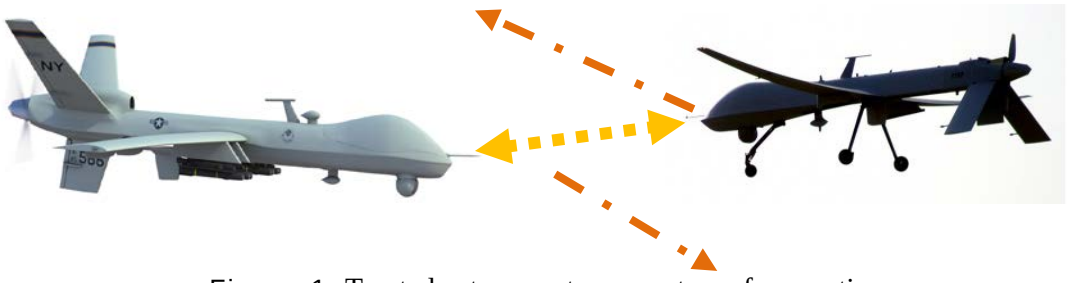

Figure 1: Trusted autonomy to guarantee safe operations

In this paper, we discuss developing autonomy using a formal approach. This approach gets us closer to licensing [3] the autonomy with a pilot's license by guaranteeing that the autonomy has the correct behaviors implemented as required for a pilot. This is especially different and challenging as compared to software verification. The Practical Testing Standard (PTS) followed by the FAA specifies the required behaviors of the pilot. Some examples of behaviors are: 1) The pilot should be capable of recovering from flight plan deviations, 2) The pilot shall be capable of recovering from lost communications and 3) The pilot shall ensure the aircraft is in proper operating order before initiating flight. The main objective is to develop pilot behavior using rule-based production systems as implemented in intelligent agent architectures (SOAR, ACT-R) and then translating to a formal analysis environment (Uppaal, ACL-2) to guarantee safety and progress properties. Progress refers to the pilot behaviors integrated together will meet the mission intent of flying from one location to the other.

How do we design trustworthy intelligent systems that guarantee safety and progress?

Motivation to Enhance Autonomy with Trust

The promise of AS (making intelligent decisions in complex situations) is attractive. Enhancing the intelligence of a system leads to complex adaptive behavior. There is a lack of understanding of the emergent complex behavior of AS within dynamic systems. So our approach proposes to prove safety properties hold even with emergent behaviors, helping us in deciding if an unsafe operation may be performed by the autonomy, via new methods to guarantee performance of autonomy [4]. The cognitive architectures typically used for implementing autonomy have some characteristics that pose serious challenges when attempting to verify a system. Specifically, two such challenges are:

1. **Implicit behavior**: Behaviors implemented as part of a cognitive architecture are not explicitly mentioned depending on the constructs allowed in the agent modeling . For example some of the constructs of a language might allow binding at runtime [5]. Leading potentially to a lack of understanding of the response.
2. **Non-deterministic behavior [3]**: Behaviors that lead to selecting one among several possible options from a state.

To deal with these characteristics of autonomous systems we propose the development of a formal methods (FM) approach. FM is a set of mathematically rigorous tools and techniques for the verification of hardware and software systems [7]. Formal verification can be used for certification credit under DO-178C Supplement DO 333. Formal methods can address implicit and non-deterministic behaviors by examining the bounds during design time. The bounds determine the runtime monitors that need to be evaluated at runtime checking safety

The FM approach to developing a system model has the potential to increase the transparency of the intelligent models. This enables one to apply methods to prove or disprove the correct and safe operation of the system.

We enumerate two approaches to trustworthy autonomy with verification in mind as shown below:

1. **Translating from a cognitive architecture to a formal verification environment**: The cognitive models are built in a cognitive architecture and then translated to a formal verification environment.
2. **Design of intelligent systems using a formal methods approach**: This involves developing a formal methods approach to design, develop, and implement the requirements for autonomy.

How do we develop formal approaches to design autonomy to guarantee correctness of autonomous behaviors?

The first approach seamlessly flows into the process presently used by developers. Therefore, it can be integrated easily and quickly. However, it requires analyzing the cognitive architectures and representations, which is a challenging task. The focus is to maintain the existing constructs in a translation to the formal methods domain, where they can be more readily verified with a challenge of validating the translation. The second approach potentially could be a complete formal approach that translates the inputs of a common DSL into the different paradigms. This is a more challenging task as it requires developing an understanding of how the DSL should be designed to represent both the paradigms (Cognitive architecture and FM) with the constructs that can be implemented. so more fundamental research efforts need to be conducted.

Cognitive Architectures

A cognitive architecture defines the interaction among the different components involved in implementing the decision-making behavior. Commonly, the architectural components are classified in three different categories: perception or external interaction, the memory or the information retrieval, and the production system rules. The perception includes different sensory modules to sense the external environment. Once this information is received, a comparison is done with the goal as defined in the Memory. Finally, based on the understanding of the situation, rules are fired to execute appropriate actions.

One of the cognitive architectures is shown below in Figure 2, which is that of Adaptive Control of Thought—Rational (ACT-R) [8]. Another architecture that can achieve some similar results is that of Soar (Figure 3). According to these architectures the production rules fire based on the situation. Common methods of rule generation include statically generated or generated through experiential learning performed by the autonomous system. In our system these rules implement the pilot behavior as captured from the PTS standards. The implementation of these cognitive architectures leads to several concerns as we found in our effort on Certification Considerations of Adaptive Systems with NASA [4] as listed:

1. Does the integration of all the rules of the composed system still lead to successfully executing the goal?
2. As rules are generated and added to the composition, can it lead to any unsafe operation or execution?
3. How do we work with the rules that may deal with some implicit information like binding a value to a variable during execution?

4. How do we deal with non-determinism where several rules could fire at the same time?

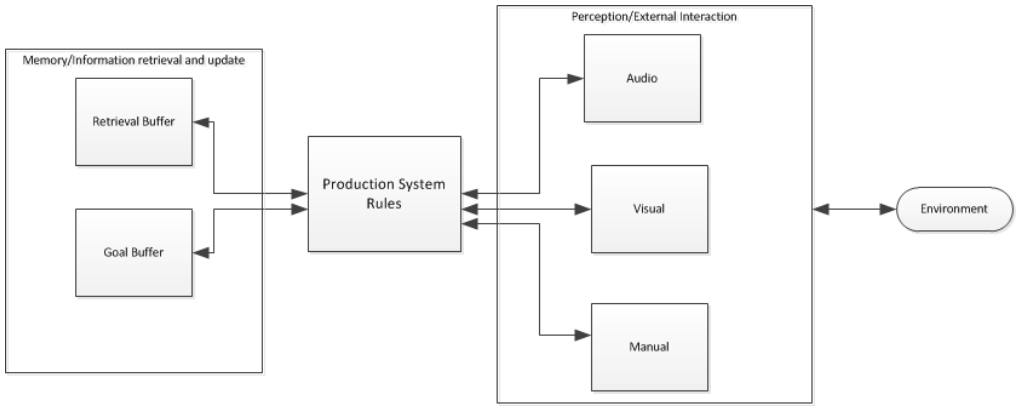

Figure 2: Cognitive Architecture ACT-R

We describe in the next section our process of translation to translate from a cognitive environment to a formally verifiable environment.

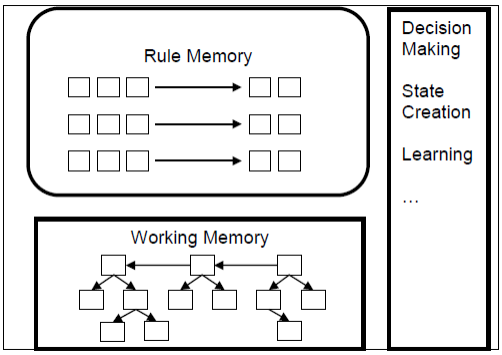

Image Source: Soar Tutorial Part 1 [9]

Figure 3: Cognitive Architecture Soar

We describe in the next section our process of translation from a cognitive environment to a formally verifiable environment.

How do we instantiate formal approaches to design autonomy to guarantee correctness of autonomous behaviors?

Formal Verification

The goal of the translation is to map the cognitive model into a formal language, where progress and safety requirements can be verified completely [6]. 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. In Figure 4, we have a representation of a cognitive architecture (rules) on the top and the formal model (finite state machines) at the bottom. The compo-

sition 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. We then discuss an alternate formal verification approach (using the ACL2 theorem prover).

Mathematical representation in formal environment

The rules are translated to formal, mathematically rigorous representations known as timed automata [6], 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 considers a finite subset of the infinite 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.

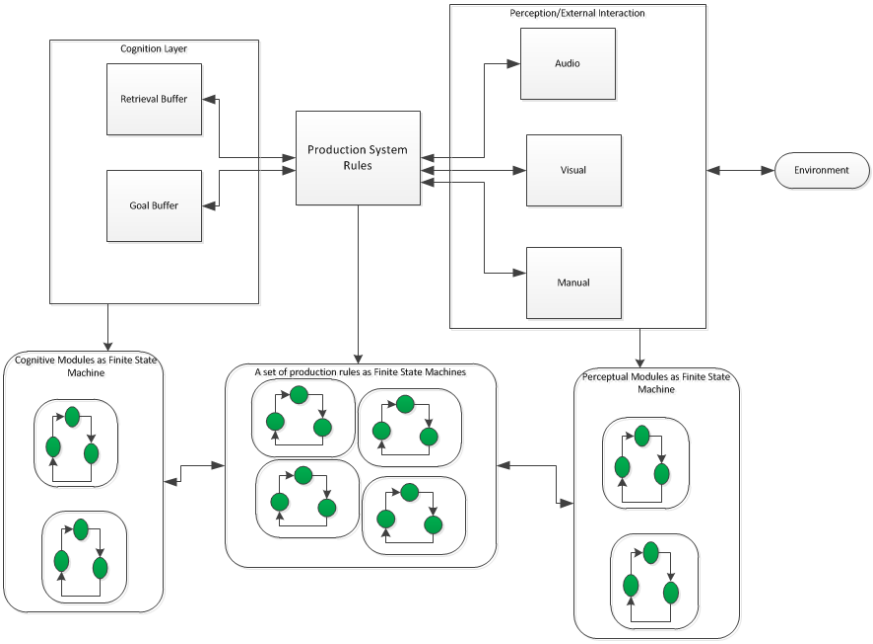

Figure 4: Translation representation

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 [7, 8],. 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. These provide capabilities in mapping real time system requirements. There exists a variety of tools to input and analyze timed automata and extensions, including the model checkers

Uppaal [11, 12], Kronos [13], and Temporal Logic Actions (TLA) [14]. These tools are becoming more and more mature. Formal representation of timed automata is defined below.

Formally, timed automata can be defined as $(Q, inv, \sum, C, E, q_0)$ where

Q is finite set of states or locations

inv are location invariants

$\sum$ 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, σ, r, q') defining a transition from state q to state q' with a guard or clock constraint g, an action or event σ, and an update or reset $r.q_0$ is the initial state or location

Figure 5: Formal Automata Description

Translation into the formal representation as described here strengthens the approach by enforcing explicitly representing all the relevant important characteristics of the system to guarantee correctness.

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.

1. **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 6).
2. **A[ ] p** – p holds invariantly, i.e., p is true in all reachable states (Figure 7).
3. **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 8).
4. **E[ ] p** – p is potentially always true, i.e., there exists a path in which p is true in all states (Figure 9).
5. q → p satisfaction of q eventually leads to p being satisfied (Figure 10)

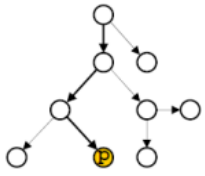

Figure 6: E<>p

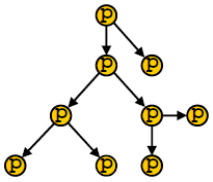

Figure 7: A[]p

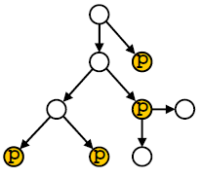

Figure 8: A<>p

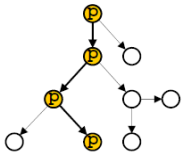

Figure 9: E[]p

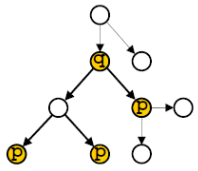

Figure 10: q-->p

Algorithms Translating a Cognitive Architecture to a Formal Methods Environment

In this section, we discuss different approaches of translation to a formal analysis environment. One of the approaches involves model checking, and the other uses a theorem prover.

Translation to the Uppaal Model Checker

We elaborate on the algorithm to translate from a cognitive architecture (ACT-R or Soar) to Uppaal, highlighting challenges in the translation process from agent-based cognition to a more mathematical, state-machine-based model. The first step in the translation process is to translate the production rules. This translation involves creating a *template* for each of the rules. Each template is a state machine model implementing the condition evaluation and the corresponding action. This translation exposes implicit references to variables, which need to be identified and explicitly indicated during the translation process. Additionally, the translation process should maintain the architectural integrity, which deals with the interaction among the components. Some of the characteristics of the cognitive engine dealing with phases of operation as well as interaction among the rules are challenges to be addressed by the translator. Once the rules are translated, models of the external components interacting with the production rules are created. This approach translates the cognitive architecture interactions to the formal environment. After generating the architectural components, instances of all the state-machine templates are created that represent the system. Figure 11 shows the algorithm for manually translating from ACT-R and Soar to Uppaal.The translation process indicates certain general properties and certain properties specific to the cognitive modeling framework. In Figure 11 we indicate the general aspects of translating from a rule-based agent cognitive modeling tool. Figure 12 elaborates on the generic approach to handling implicit information. This is the information when binding occurs at runtime. These need to be explicitly represented for FM to prove properties related to these representations.

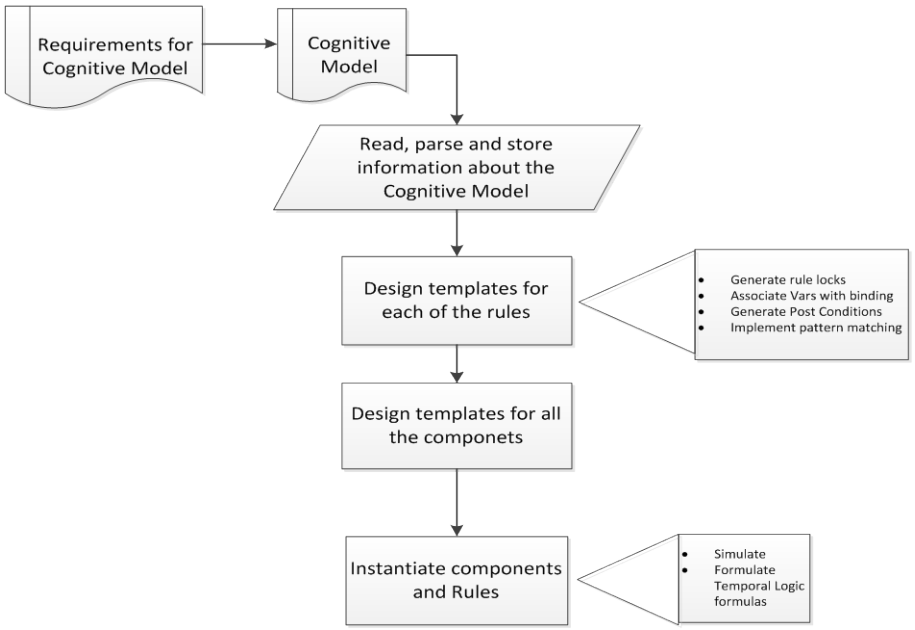

Figure 11: General modules for translation to Uppaal

Implicit to Explicit representation

- **For all the variables in the condition of a rule**
  - If value bound to variable or constant, match to value
    continue executing the rule
    else
    do not execute the rule

    endif
  - If variable is unbound,
    If value is nil,
    rule does not fire
    else
    bind variable to value and continue executing the rule

    endif

Figure 12: Generic approach dealing with implicit representation

For reference, see a simple counting example in Soar translated to Uppaal. To handle large scale complex systems this approach can be extended with relevant refinements like assumptions and guarantees. One of the rules for the example is to increment a counter. The Soar implementation of this rule is shown in Figure 13. Based on the translation described in Figure 11, the Uppaal model in the formal verification environment (Figure 14-15) is manually generated. The translated model in the Uppaal environment allows the model checker to prove that on all paths eventually we can reach the goal state. The translation allows exhaustive analysis on specific requirements for agent-based models.

```
sp {counter*apply*increment
   (state <s> ^name counter
              ^operator <op>
              ^num <c>)
   (<op> ^name increment
         ^first <c>)
-->
   (<s> ^num (+ <c> 1)
        ^num <c> -)
}
```

Figure 13: Counter increment rule in Soar

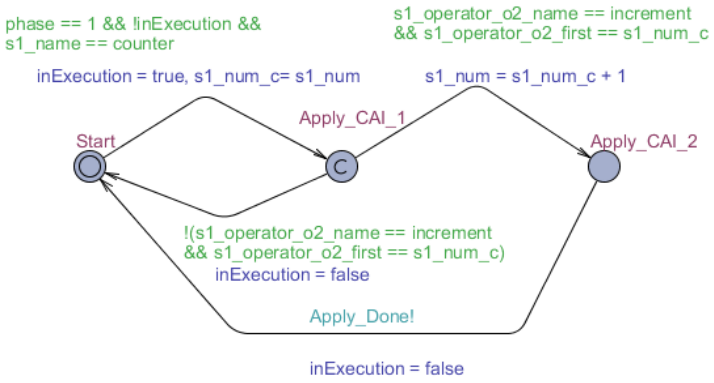

Figure 14: Counter increment rule translated to Uppaal

Translation to the ACL2 Theorem Prover

A Computational Logic for Applicative Common Lisp (ACL2) is a highly automated theorem proving system. ACL2 is a functional subset of Lisp. There are admission criteria for function definitions, and a subset of the language is executable. A formal model in ACL2 is implemented as a set of function definitions (rather than state machines). Properties to be verified are stated as theorems, and human intervention is typically required to build up the infrastructure (definitions and lemmas) so that ACL2's rewriting system can prove the theorem. Figure 16 shows the advantages (+) and disadvantages (-) of ACL2 and Uppaal. These categories of advantages and disadvantages are common to most theorem provers (such as ACL2) and model checkers (such as Uppaal). Note that Uppaal has several advantages that reduce the level of human effort required to model and prove properties. Hence, Uppaal is our preferred modeling approach.

	UPPAAL	ACL2
Effort to Prove Properties	+ Automated	- Typically requires human intervention
State Space Limits	- Large state space can result in slow proofs	+ Proof time not necessarily tied to size of state space
Eventually/Existence Properties	+ Native support	- Existence proof requires finding the witness first
Timed Properties	+ Native support for time	- Time must be mapped to a number of steps
Handling Nondeterminism	+ Random selection among multiple possible transitions is the default	- Awkward—need to use a function stub with a seed value
Simulation/Debugging	+ Out-of-the-box support	- Have to write a 'run' function
Working Memory Representation	- Working memory is flattened	+ Working memory structure is maintained
String Support	- Must be mapped to integers	+ Native type

Figure 15: Uppaal and ACL2 Comparison for the Modeling and Verification of Soar Agents

The Soar rules are logical statements of the form "If X, then do Y." These naturally translate into ACL2 functions with an if-then-else construct (where the else clause just returns the current working memory). Working memory in Soar is a directed graph structure. The elements of working memory are triples (one corresponding to each edge in the graph) of the form (identifier (node), attribute (edge), value (node)). There are no isolated nodes. Hence working memory can be represented as a list of triples in ACL2. The counter increment rule shown in Figure 13 can be translated to the ACL2 function shown in Figure 16. (Here "num" is the variable 'c', and 'wmem' is the list of triples containing the contents of working memory.)

```
(defun counter*apply*increment (wmem)
  (if (and (wmem-match wmem "s1" "name" "counter")
           (wmem-match wmem "s1" "operator" "o")
           (wmem-match wmem "s1" "num" t))
      (let ((num (nth 2 (wmem-match wmem "s1" "num" t))))
        (if (and (wmem-match wmem "o" "name" "increment")
                 (wmem-match wmem "o" "first" num))
            (let* ((wmem (cons (list "s1" "num" (+ num 1)) wmem))
                   (wmem (remove (list "s1" "num" num) wmem)))
              wmem)
          wmem))
    wmem))
```

Figure 16: Counter increment rule translated to ACL2

Rather than translating Soar rules directly to functions in ACL2, one can develop an interpreter for Soar rules in ACL2. This forces the semantics to be made explicit and makes it possible to reason about the semantics and to prove that a translation is correct. On the other hand, translators can be completed and optimized more quickly since they are less formal. We implemented some basic functionality for an ACL2 interpreter but did not complete it.

Case study on a pilot licensing approach

In this section, we discuss the implementation of the pilot behavior in the cognitive architecture Soar. Currently the behavior is pertaining to the preflight checklist conducted by a pilot. Other behaviors will be added into the modular architecture. We discuss the architecture followed by the formal verification approach with Uppaal.

Pilot behavior modeled as cognitive agent

Soar agents are being developed to conduct preflight/inflight checklists and procedures. Real-world checklists often contain ambiguities and can be interpreted to have multiple valid paths to completion. Checklists can also be misinterpreted by human operators who miss items or take shortcuts that are actually invalid. Our agents are being designed to provably avoid invalid steps, and to learn to complete checklists in the most efficient manner, given the duration times of various checks and the dependency among the tasks. The completed design still provides the agents to execute different paths to complete a behavior which allows non-determinism to be present in the system and thus makes it challenging to guarantee correctness.

In our current implementation, a Soar agent is able to traverse a portion of a preflight checklist modeled as a graph whose nodes are checklist items, and whose links are the 'next' relation. All checklist items (except for the first and last item) will have one or more outgoing next links, and

one or more incoming next links. For any given checklist item, all incoming next links must pass before that checklist item can be verified. Some checklist items are modeled to take negligible time, while others (such as starting up certain avionics components) may have a time duration associated with them. The agent is guaranteed to visit each checklist item, and utilizes reinforcement learning (RL) to explore different valid paths through a checklist over time, learning an optimal solution. The current system architecture is depicted in Figure 17.

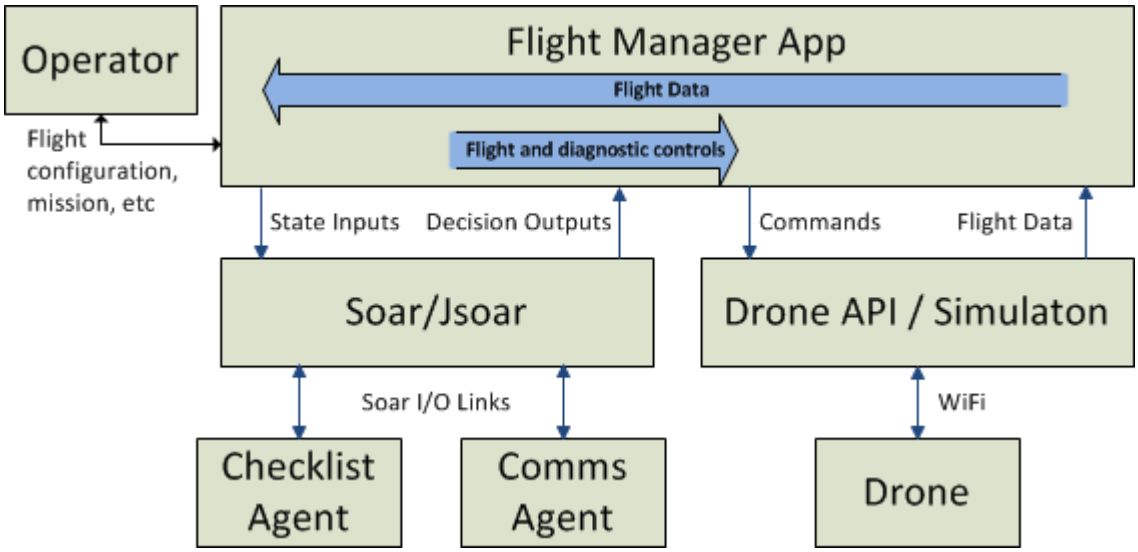

Figure 17: Software Architecture for the autonomous system

Our current implementation utilizes Soar's built-in Reinforcement Learning (RL) capabilities to attempt to learn the most efficient path through a given checklist. The Soar RL mechanism provides reward value to the agent when the checklist is completed. As more and more runs are completed, rewards accumulate in state-operator pairs that have shown to be more advantageous to achieving our goal state. After a sufficient amount of iterations over a given checklist domain, the accumulated numerical reward values should converge to a point that the optimal path is always taken.

One of the challenges in implementing learning mechanisms involves dealing with the convergence to a solution. The system in general should be able to respond to changing situations quickly, which means convergence should occur at a fast rate; but this has the potential of leading to unstable behavior. This is discussed in several research efforts [15, 16] In RL-specific Soar, configuration values (Epsilon-greedy and Boltzmann and their associated parameters) deal with aspects of convergence controlling the exploration. In many cases, it is crucial to allow a Soar RL model some flexibility to lessen the chances of settling too early on a local sub-optimal solution. Additionally, Soar includes both Q-learning and Sarsa RL policies, which can be configured to effect learning behavior along with their learning rate and discount rate parameters. Finally, there are some 'decay-rate' configuration parameters that gradually reduce the probability of exploration and/or the learning-rate over time, which should reduce some of the inherent oscillations between taking an optimal path and exploring a suboptimal path. Further enhancement is being developed in collaboration with Florida Institute of Technology [17].

The learning methods feed into the Soar rules that implement the behavior of the agent (pilot). These rules include evaluation of conditions followed by taking actions. This is similar to the rep-

resentation in formal analysis tools with state machines that evaluate guards and then execute actions. Translation of these cognitive rules into the formal environment enables compositional analysis of the rule-based agent, thus guaranteeing the pilot behavior—one of the crucial aspects of verifiable autonomy.

Guaranteeing pilot behavior with formal analysis

The pilot behavior is represented as a hierarchical architecture as described further in Figure 18. The hierarchical representation decomposes the expected behavior of a pilot into different levels: the Strategy/ Decision making behavior, the Operational behavior and the Execution behavior. The Strategy/Decision making behavior implements the highest level of abstraction of a task that is communicated to the autonomy by humans. The model decomposes the request received into tasks at lower levels. For example, the behavior of the pilot to perform a preflight checklist is represented as shown in Figure 19, Figure 20, and Figure 21. Figure 19 represents the set of equipment to be evaluated to perform the preflight checklist, followed by checking a particular category of equipment as a set of operations, as shown in Figure 20. Finally, Figure 21 shows the direct interaction with the physical equipment to send the request, followed by evaluation of the received response to find out if the equipment is functioning correctly.

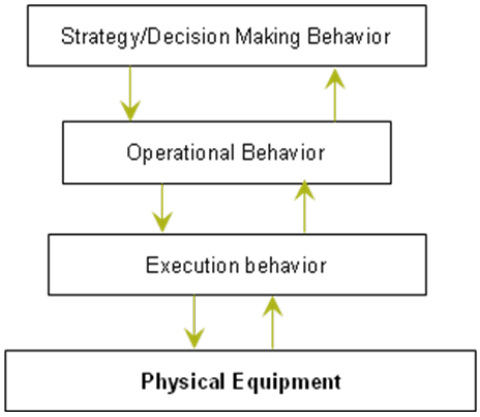

Figure 18: Architectural representation of Pilot Model

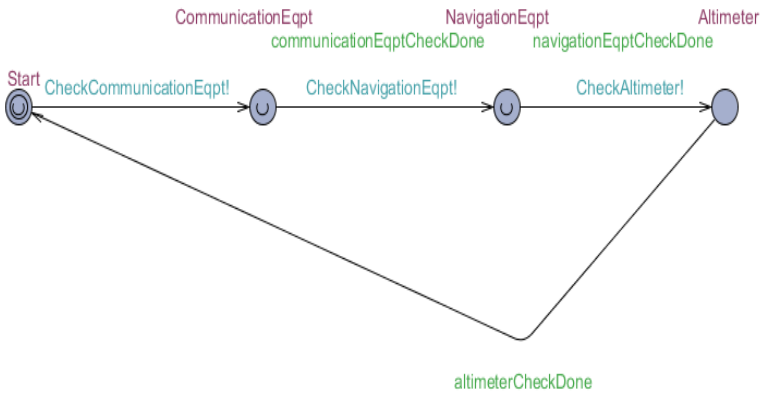

Figure 19: Preflight checklist (Strategy)

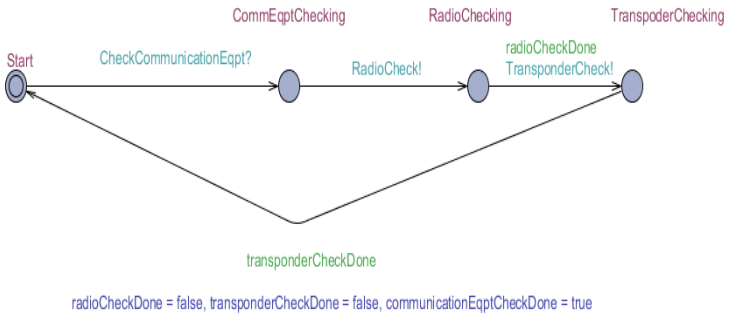

Figure 20: Checking a particular category of equipment (Operations)

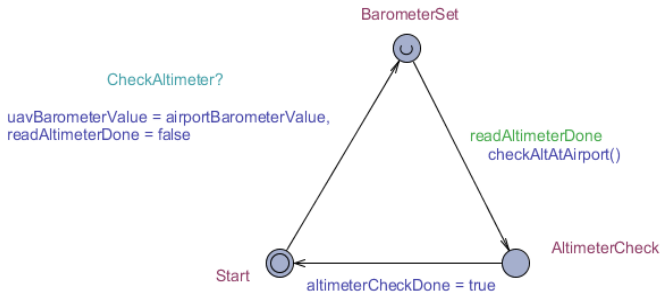

Figure 21: Executing behavior interacting with the physical equipment

Once this representation is modeled, the expected range of inputs are generated and exhaustively evaluated to guarantee the behavior of the pilot. This is conducted by formulating temporal logic queries. These temporal logic queries exhaustively evaluate all the generated traces to check if the requirement modeled as the query is satisfied by the model or not. Examples of queries are:

a. ***"Always eventually the pilot checks the equipment (instance: radio) "***

- A<> radioCheck1.RadioChecking

The final response to the query comes back as satisfied, indicating that the autonomy always checks the radio.

b. ***"Pilot responds to faulty instrument (instance: altimeter) "***

- ((airportElevationValue-altimeterValue) >=1 || (altimeterValue-airportElevationValue) >=1)&& pilotAltimeterChk1.AltimeterCheck --> altimeterFault

The response indicates that the autonomy (the "pilot") always responds to faulty equipment.

If the response is not satisfied, Uppaal comes back with a counterexample, as shown in Figure 22, with the trace of states that lead to the failure. The model can then be modified to satisfy the query—i.e., the property being verified.

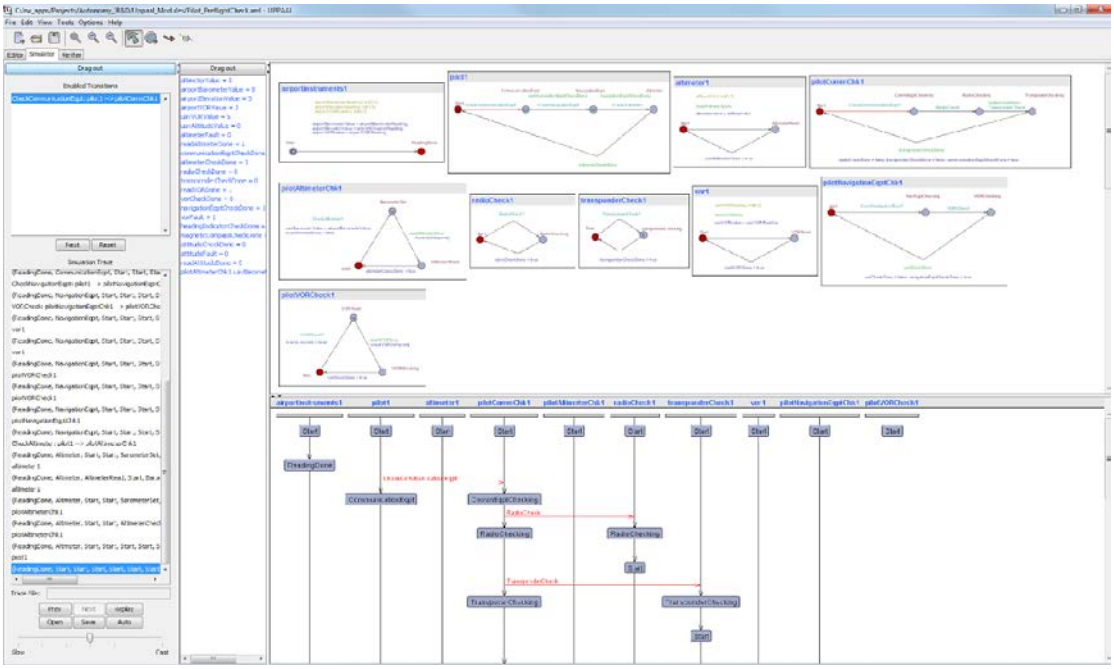

Figure 22: Trace showing counterexample

Challenges in Developing Trustable Autonomy

The fundamental difference between cognitive modeling and formal verification makes it challenging to develop cognition with trust. This is because cognitive architectures model the intelligence based on human psychology. This approach to modeling autonomy leads to several challenges: non-determinism, concurrent execution of rules, and representation of the knowledge base.

In the algorithms we have been developing for the Soar and ACT-R environments, the specific challenges we dealt with were:

- Maintaining rule formalisms in the formal environment
  - Addressed by limiting execution to one rule at a time using a locking mechanism between state machines
- Maintaining the cognitive architectural lifecycle, such as Soar's propose-select-apply cycle
  - Addressed with a scheduler state machine to manage the lifecycle and ensure all the appropriate rules are fired
- Addressing concurrency (e.g., for Soar's propose rules)
  - Addressed by attempting to fire all possible propose rules in sequence. (In our examples to date, the order that propose rules fire does not affect the outcome since they only affect working memory elements below their own operator vertex.)
- Maintaining architectural integrity
  - Addressed by implementing methods to properly order the binding of variables, manage the flow of data, and evaluate the condition during translation

Additional challenges that still need to be addressed are the following:

- Expand to support instantiation
  - For example, not “o” for an operator but “o1”, “o2”, etc., since there can be multiple operators and even different instantiations of the same operator
  - Deciphering instantiations from rules without interfacing with the working memory
    - Working memory generates a tree structure for variables during execution to assign values, even without accessing the tree structure the instances can be generated
- Representing knowledge
  - How best to represent knowledge or learned information?
  - Do we automate the knowledge representation through translation?
  - Specific knowledge information might need to be added manually

Our translation approach has not yet been implemented in an automated fashion, so additional challenges may arise during implementation. Many of the challenges described here are common among cognitive architectures, so our approach could be extended to address similar concerns and develop translators for other cognitive architectures.

Conclusion and Application

In this paper, we outlined our approach to the development of trusted autonomous systems through translation to a formal methods environment for verification. This can be a key component in realizing the promise of autonomous systems by verifying their behavior in mission-critical systems to established standards. This will lead to developing formal approaches to learning to be able to reason about learning outcomes. Therefore, we will be able to guarantee properties related to learning that would lead to successful completion of a mission or executing safer operations. Formal reasoning for autonomy can be extended to handle a system of systems, where each system is an autonomous system with its own autonomy. This would allow verification of a group of autonomous systems working collaboratively by executing local missions to accomplish a global mission. This discussion provides a formal approach to verifying autonomy to get closer to licensing autonomy as a pilot.

Fundamental differences between autonomy and formal verification need to be addressed to develop trusted autonomy

References

1. Jones, R. M., Crossman, J. A., Lebiere, C., & Best, B. J. (2006). An abstract language for cognitive modeling. In *Proceedings of the Seventh International Conference on Cognitive Modeling* (pp. 160-165). Trieste, Italy.

2. Christian Hahn and Klaus Fischer, The Formal Semantics of the Domain Specific Modeling Language for Multiagent Systems. Agent-Oriented Software Engineering IX, Lecture Notes in Computer Science Volume 5386, 2009, pp 145-158
3. S. Bhattacharyya, D. Cofer, D. Musliner, E. Engstrom and J. Mueller, “Certification Considerations for Adaptive Systems”, Technical Report, NASA/CR–2015-218702.
4. M. Fisher, L. Dennis, M. Webster, "Verifying Autonomous Systems", Communications of the ACM, Vol. 56 No. 9, Pages 84-93.
5. John E. Laird and Clare Bates Congdon et.al, The Soar User’s Manual Version 9.3.1December 5, 2011
6. M. O’Connor, S. Tangirala, R. Kumar, S. Bhattacharyya, M. Sznaier, and L.E. Holloway. A bottom-up approach to verification of hybrid model-based hierarchical controllers with application to underwater vehicles. In Proceedings of ACC’06, page 6 pp. IEEE, 2006
7. What is Formal Methods? http://shemesh.larc.nasa.gov/fm/fm-what.html
8. Bothell, D. (2007). ACT-R 6.0 Reference Manual. From the ACT-R Web site: http://act-r.psy.cmu.edu/wordpress/wp-content/themes/ACT-R/actr6/reference-manual.pdf.
9. Laird, John. The Soar 8 Tutorial. May 14, 2006. Available at: http://ai.eecs.umich.edu/soar/sitemaker/docs/tutorial/TutorialPart1.pdf.
10. Rajeev Alur, David L. Dill. 1994 A Theory of Timed Automata. In Theoretical Computer Science, vol. 126, 183-235
11. Uppaal: A toolbox for modeling, simulation and verification of real time systems, www.uppaal.com
12. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi , UPPAAL— a Tool Suite for Automatic Verification of Real-Time Systems, , BRICS Report, RS-96-58, 1996.
13. M. Bozga et.al, Kronos: A Model-Checking Tool for Real-Time Systems, Formal Techniques in Real-Time and Fault-Tolerant Systems, Denmark 1998.
14. L. Lamport , Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, , Addison Wesley Professional 2004.
15. Shai Shalev-Shwartz, Ohad Shamir, Nathan Srebro, Karthik Sridharan; Learnability, Stability and Uniform Convergence Volume 11 Journal of Machine Learning (Oct):2635−2670, 2010.
16. Nguyen NT, Robust adaptive optimal control modification with large adaptive gain American Control Conference 2009 June 10-12.
17. VanderHorn, N, Haan, B, Carvalho, M, Perez, C (2010) Distributed policy learning for the cognitive network management system. The 2010 military communications conference—unclassified program—cyber security and network management (MILCOM 2010-CSNM).