Runtime Verification of Cognitive Applications

Jonathan Springer, Donald D. Nguyen, Richard A. Lethin
{springer, nguyen, lethin}@reservoir.com
Reservoir Labs, Inc.

Introduction

Cognitive systems have been the subject of much research, and are increasingly of interest in embedded systems. However, cognitive applications have unique characteristics that make them challenging to verify, validate, and debug. A cognitive application by definition makes intelligent decisions – if it were possible to formally and precisely express its behavior under all circumstances, the cognitive system would not need to be "cognitive."

Architectures for cognitive applications commonly utilize a cognitive system layer on top of which the application is developed. Unlike a conventional application, however, a cognitive application's functionality is often encoded primarily in data, not in control structures. The internal operation of such a cognitive application thus resembles an interpreter. We seek to debug the cognitive application itself, and to do this we add specialized support into the cognitive system that runs it.

An example cognitive application is UAV mission planning. Rather than rely on ground-based operators, as is currently done, recent work seeks to add autonomy to the UAV. Space and weight constraints make the UAV platform an embedded one.

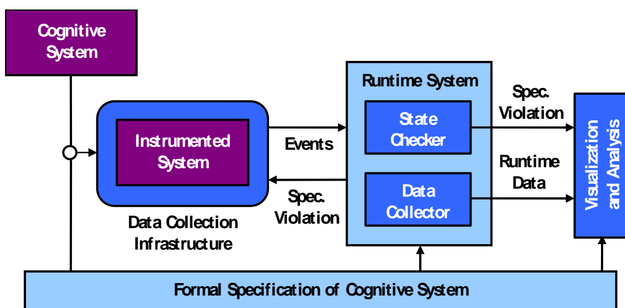

Figure 1: Cognitive V&V infrastructure

We are developing techniques for run-time verification and validation of cognitive applications. This work has three main parts:

- Formalisms for specification of safety properties of cognitive systems
- Data collection infrastructure that can be used for checking application behavior against specification
- Visualization tools for interpreting the application data in the event of an error, to aid in debugging

This work was produced with US Government support, under Air Force contract FA8750-06-C-0133. The US Government has certain rights to this work.

Specification Formalisms

The purpose of the specification language is to encode desired correctness properties. As an example, we are looking at temporal logics, which combine a base logic (e.g. first-order logic) with mechanisms for discussing state and change over time. Temporal logics have been investigated for specification of traditional programming languages as well [1].

Depending on the expressivity desired, different forms of temporal logic may be chosen. For example, linear temporal logic encodes information about a single fully-ordered timeline. Properties of the application can be asserted at certain points or ranges in its execution. On the other hand, if the applications reason about multiple timelines, computational tree logic may be most useful.

Operationally, a temporal logic specification can be transformed to a finite-state automaton through Tableau Construction [2]. As the application runs, its behavior induces movement through the automaton, and the presence of certain error states provides a simple mechanism for checking specification conformance.

A key challenge in applying temporal logic (or any formalism) is choosing a system that can encode safety properties of interest to the cognitive application programmer. The formalism must be complex enough to be able to encode sufficiently sophisticated statements while being simple enough to admit efficient checking at runtime. Cognitive applications often need to be able to handle uncertainty, and so it may be necessary for the specification formalism to incorporate probabilistic elements.

Data Collection Infrastructure

The characteristics of cognitive systems make them challenging to debug. Most conventional debugging methods rely on reproducing the circumstances that lead to a fault repeatedly in a debugger. This approach works adequately for a conventional application, but for a cognitive application that interacts with a complex environment that cannot easily be reproduced, and that may contain significant nondeterminism, the conventional approach may be insufficient.

We aim to develop the capability for first-fault debugging. A first-fault debugger attempts to capture sufficient context for an error to enable debugging without resorting to attempts to recreate the circumstances. First-fault debuggers for traditional systems [3] rely on saving the control flow of the application, as well as some key state. As noted previously, a cognitive application is primarily in the data, so the need to save data versus control flow is correspondingly greater.

Our approach is to design a first-fault debugger around the notion of producing a data trace that follows the flow of data used to compute a value of interest. This value would be identified through a state check violation, and identified with a high-level construct or concept which we term an *event*. Events have dependencies on other events, forming a DAG that highlights the data and structure of that data that is relevant to the detected error.

A key challenge for data collection is how to collect and retain significant amounts of data without imposing undue burdens on the cognitive system. These burdens can be in execution time or storage space. Techniques for managing this burden include checkpointing combined with replay and/or reconstruction, data aging, as well as simple compression and clever state encodings. Checkpointing saves the entire program (cog app) state at certain intervals. For errors that occur between checkpoints, the system can replay the execution up to the error state, or reconstruct the states leading up to the error via disassembly and analysis of the machine code. In either case, it is always necessary to save state updates that come from "outside" the system. Data aging recognizes that in a practical system, it may be impossible to store the complete program execution history, so some policy for selecting the most relevant data is needed. It is likely that a combination of these techniques would be most effective.

We have been investigating Soar as a representative cognitive system [4]. A fundamental mechanic in Soar is that working memory elements (WMEs) match productions (rules), leading to execution of corresponding production actions. Actions update the system state by changing WMEs, leading to further matching. Production firings map to our concept of an event, and are linked together through WMEs, which can be viewed as the dependencies. Specification statements can naturally be made with respect to the presence or absence of WMEs.

It is important to emphasize that our approach is intended to be general beyond Soar, however. Generalizing to other production systems is obvious, but any cognitive system in which there is a notion of causal state change over time should be compatible with our approach.

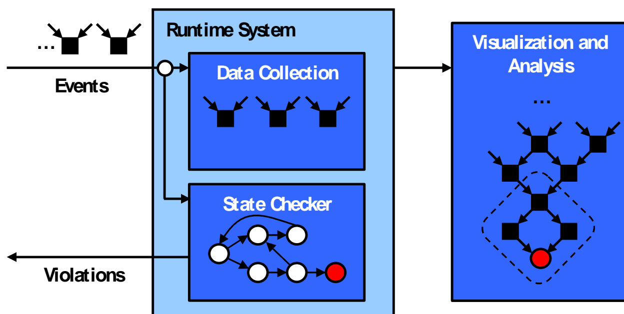

Figure 2: Data collection framework, state checker, and visualization modules

Visualization

The event dependency structure naturally lends itself to visualization, for example, as a graphical display of the event graph. The event graph is likely to be complex, such that any visualization will benefit from graph summary and other inspection techniques that allow management of graphs that cannot feasibly be viewed in their entirety at once. Visualization should also be interactive, and should interface to the collected data trace as information is requested. Many idioms from traditional debuggers can be applied at this level.

Conclusion

We are designing infrastructure to assist in validating and debugging cognitive applications. Specification languages and the associated data collection and checking provide two main benefits: additional assurance that the application behaves as desired and information that can characterize and help debug errors. We are currently experimenting with specification languages based on temporal logic, and prototyping data collection and state checking techniques. To validate our approach, we are utilizing a UAV planning scenario as a cognitive application on top of a Soar cognitive system.

Some of the open questions that we hope to address are:

- What kinds of logic (temporal or otherwise) are expressive enough to encode commonly desired specifications?
- Is the overhead from data collection acceptable in terms of application performance?
- Does the event dependency graph capture sufficient information in an appropriate form for analysis and debugging?

References

[1] K. Havelund, "An Overview of the Runtime Verification Tool Java PathExplorer", Formal Methods in System Design, 24(2), March 2004.

[2] M.C.W. Geilen, "On the construction of monitors for temporal logic properties," in RV'01 - First Workshop on Runtime Verification, volume 55 of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers, 2001.

[3] A. Ayers, R. Schooler, A. Agarwal, C. Metcalf, J. Rhee and E. Witchel, "TraceBack: First Fault Diagnosis by Reconstruction of Distributed Control Flow" In the Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2005.

[4] J. E. Laird and Paul Rosenbloom, "The Evolution of the Soar Cognitive Architecture", in Mind Matters: A Tribute to Allen Newell, Eds. D. M. Steier and T. M. Mitchell, 1996.