UC Merced
Proceedings of the Annual Meeting of the Cognitive Science Society

## Title
Concurrent Execution in a Cognitive Architecture

Permalink
https://escholarship.org/uc/item/7542d3cm

Journal
Proceedings of the Annual Meeting of the Cognitive Science Society, 31(31)

ISSN
1069-7977

Author
Choi, Dongkyu

Publication Date
2009

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

# Concurrent Execution in a Cognitive Architecture

**Dongkyu Choi (dongkyuc@stanford.edu)**
Cognitive Systems Laboratory
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305 USA

## Abstract

Many human activities happen in parallel. People read while eating, talk over a phone while walking, and steer their cars while pushing the gas pedal. Therefore, cognitive architectures, as computational theories of human cognition, should also have facilities to support this type of capability. In this paper, we introduce extensions to our architecture, ICARUS, that support concurrent execution. We also provide some preliminary observations from our recent experiments.

**Keywords: cognitive architectures, concurrent execution, parallel execution, multitasking**

## Introduction

People do multiple things at the same time. They read books while they are eating lunch. They talk over a phone while walking on a street. And when they are driving, they steer their cars while they push the gas pedal. Sometimes these concurrent activities can be serialized, and people can always read books after finishing their lunch, or stop walking to take a phone call. But some other times these activities need to happen simultaneously to accomplish their desired results. For instance, a car will not be moving properly if the driver cannot steer and push pedals at the same time. Therefore, cognitive architectures (Newell, 1990), which are computational models of human cognition, will also need the ability to support simultaneous, concurrent execution.

Until recently, one such architecture, ICARUS (Langley & Choi, 2006b) could execute only one thing at a time. It was still sufficient for many domains, including the Blocks World and FreeCell (Langley et al., in press). But when used in more dynamic domains like an urban driving domain (Choi et al., 2007), it soon became clear that the system's ability to show natural behavior was limited, if it can perform only the serial execution. For example, an ICARUS agent that delivers packages in a city should perform various maneuvers, like left or right turns. To execute a turn properly, it would constantly adjust its speed to a right level and steer its vehicle at the right moment. However, a serial system can perform only one of these activities at any given time, and it often misses the timing for steering while it is focusing its attention to its speed, or vice versa.

To remedy this type of problem, we introduce two different extensions to the architecture. The first takes a representational approach, which adds a new field in ICARUS' procedural knowledge structure. In this field, users can explicitly specify parallel procedures, for which the system will consider concurrent execution within a single cycle. The other involves the capability to manage resources during the interpretation of knowledge. The system tracks the required resources for each procedure it evaluates, and uses this information to find procedures that can occur at the same time.

In the following sections, we first present a brief review of the ICARUS architecture, and introduce the urban driving domain that we use throughout the paper for illustrative purposes. Then we describe the two extensions for concurrent execution in more detail, with some experimental observations at a preliminary level. We conclude after a review of related and future work.

## Review of ICARUS Architecture

The ICARUS architecture grew out of the cognitive architecture movement, and it shares basic features with other systems like Soar (Laird et al., 1986) and ACT-R (Anderson, 1993). It makes commitments to its representation and interpretation of knowledge, and has memories that support these. In this section, we briefly review the architecture, starting with the representational components and then continuing to the processes that work over them.

### Representation and Memories

Based on psychological evidences, the ICARUS architecture distinguishes conceptual and procedural knowledge, and it has separate long-term and short-term memories for each of them.[1] A conceptual long-term memory stores descriptive knowledge structures that are similar to horn clauses, and a skill long-term memory includes procedural knowledge structures that resemble STRIPS operators. Both of these knowledge structures are organized in a hierarchical manner, allowing multiple levels of abstraction.

Tables 1 and 2 show examples of these structures, *concepts* and *skills*, respectively. The first two concepts, `yellow-line` and `at-turning-speed` are *primitive*, so they match variables against perceived objects and their attributes, and specify conditions among the matched variables. On the other hand, the last concept, `ready-for-right-turn` refers to two other concepts, `in-rightmost-lane` and `at-turning-speed`, and therefore it is a *non-primitive* concept. Similarly, the last skill in Table 2, `in-intersection-for-right-turn` is a primitive

[1]Traditionally, ICARUS' skill short-term memory also served as a basic goal memory. Introduction of problem solving and learning modules (Langley & Choi, 2006a) required a more sophisticated description of goal-related information, and the memory has been renamed to be a *goal short-term memory* in the recent versions of the architecture. While it still includes the short-term knowledge of skills – the skill instances the system is currently working on, the new name acknowledges the goal specification and bookkeeping as its main functions. This renamed memory is identical to the goal memory mentioned later in this section.

one, in which only actions the system needs to apply in the world are mentioned. The first two skills, however, are non-primitive, since they specify subgoals that, in turn, call for other skills. In this manner, the ICARUS architecture allows complex description of the surrounding world and possible courses of action.

Table 1: Some sample ICARUS concepts for the urban driving domain.

```
((yellow-line ?line)
 :percepts  ((lane-line ?line color YELLOW)))

((at-turning-speed ?self)
 :percepts  ((self ?self speed ?speed))
 :tests     ((>= ?speed 15)
             (<= ?speed 20)))

((ready-for-right-turn ?self)
 :relations ((in-rightmost-lane ?self ?l1 ?l2)
             (at-turning-speed ?self)))
```

Table 2: Some sample ICARUS skills for the urban driving domain.

```
((on-street ?self ?tg)
 :percepts ((self ?self)
            (street ?st)
            (street ?tg)
            (intersection ?int))
 :start    ((intersection-ahead ?self ?int ?tg)
            (close-to-intersection ?self ?int))
 :subgoals ((ready-for-right-turn ?self)
            (in-intersection-for-right-turn
                             ?self ?int ?st ?tg)
            (on-street ?self ?tg)))

((ready-for-right-turn ?self)
 :percepts ((self ?self))
 :subgoals ((in-rightmost-lane ?self ?l1 ?l2)
            (at-turning-speed ?self)))

((in-intersection-for-right-turn
                             ?self ?int ?c ?tg)
 :percepts ((self ?self)
            (street ?c)
            (street ?tg)
            (intersection ?int))
 :start    ((on-street ?self ?c)
            (ready-for-right-turn ?self))
 :actions  ((*cruise)))
```

The architecture also has a *goal memory*, where it stores its top-level goals and the goal stacks associated with them. Information stored in each goal stack includes a main goal, any subgoals the system is currently working on, and retrieved skill instances for the goals. For example, if the system has a goal, `(on-street ME B)`, the system might retrieve the first skill shown in Table 2, instantiate it, and store the information in the goal memory as its current intention. Then depending on the situation, ICARUS may select the first subgoal, and include an instance, `(ready-for-right-turn ME)` in the memory.

## Inference, Evaluation, and Execution

The ICARUS architecture operates in distinct cycles, and on each cycle, it invokes a series of processes. These include the inference of the current belief state, the evaluation of its skills based on the inferred state, and the execution of actions implied by the selected skill. The processes start with the sensory input from the environment at a given time. Based on the perceptual information, the system infers instances of all its concepts that are true in the current state. It starts with the lowest level structures – primitive concepts, and moves up the hierarchy to non-primitive concepts. Hence, any naive approach to the inference process is prone to the combinatorial effect caused in domains with many objects, and there have been several efforts to alleviate this problem including Asgharbeygi et al. (2005).

Once the system infers the current belief state, it starts evaluating its skills to achieve the given goals. It begins with the first goal in its goal memory and retrieves all the skills that are known to achieve the goal. Then the system evaluates them to find the first one that is executable in the current state, with a preference to more recently acquired skills among the disjunctive definitions. A non-primitive skill, which refers to subgoals in its body, is executable when all of its preconditions are met and the first unsatisfied subgoal has at least one executable skill that will achieve it. In the case of primitive skills, however, there are no subgoals for them, and therefore, they are executable once their preconditions are met. As a result, the system effectively finds the first *skill path* that consists entirely of executable skills given the current belief state.

At the leaf node of the selected skill path is a primitive skill, which implies some actions the system needs to take in the environment. The ICARUS architecture takes these actions and applies them to make changes in its surroundings. In turn, its perceptual input on the next cycle will change, and the system repeats the processes based on the new sensory data. This allows reactive execution in ICARUS, while staying to be goal-directed. The balance between the two properties is an important feature of the architecture. In the sections below, we briefly describe the urban driving domain, and continue our discussion on the architecture and its newest extensions in more detail.

## An Illustrative Domain

Driving is a complex task. It involves the perception of both static and dynamic objects and the control of the vehicle for various maneuvers. Driving in an urban setting can be even more complicated, due to various types of objects to interact with. For this reason, the development for the urban driving domain continues, since its first use as a testbed for the ICARUS architecture (Choi et al., 2004). Figure 1 shows the driving environment in the current version of the domain. In this domain, there is a vehicle for an ICARUS agent, and there are static objects including street segments, lane lines, sidewalks, and buildings. The environment also includes dynamic objects like pedestrians and other cars. To drive around in this city, the ICARUS agent should control the steering angle and the amounts of gas and brake pedal application.

Figure 1: A screenshot of the urban driving domain.

As shown later in this paper, basic driving tasks like left and right turns are often used, but several higher-level tasks including package delivery are also composed. For all of these tasks, the agent should perceive information on various objects in the world, decide its course of action to achieve its goal, and apply control actions in the environment. In the case of package delivery, an agent can see the addresses on its packages and check the addresses of buildings around it. With no map of the city provided, it can simply move around in the city until it finds the right street, and proceed to the package address. During the execution, the agent faces many different challenges, from the basic negotiation of the roads, to collision avoidance with unexpected obstacles like pedestrians crossing the street.

The richness of the domain provides an interesting environment for testing different aspects of agents, and it is particularly useful to test their ability to perform multiple tasks simultaneously. With only a handful of control inputs, the domain is simple enough for a clear demonstration of concurrent behaviors. But the concurrency is naturally required in this domain, and the agent's capability is easily tested using both qualitative and quantitative measures of the driving behavior. For example, one can simply see how smooth an agent's driving behavior is. Alternatively, one can use the time taken for a specific task like a right turn, or the amount of deviation from the ideal driving path for the turn as a quantitative measure.

## Concurrent Execution in ICARUS

The latest extensions to ICARUS include two different methods for concurrent execution. One that uses a new field in skills allows explicit specifications of concurrency. Users can specify subgoals that require simultaneous consideration for execution in this field, and the system evaluates each and every one of these *parallel* subgoals in a single cycle. When this method is used, the user should make sure that there are no conflicts among the subgoals, since the system does not employ any facility to check if they are conflict-free.

The second method involves a more implicit way to represent concurrency, and it is also more complicated. In this case, the system does not have any special field in its skill structures but, instead, it assumes that all subgoals are candidates for concurrent execution, subject to the resources they require. The architecture tracks resources that are used for each primitive skill, and prevents concurrent execution of subgoals that lead to primitive skills with resource conflicts. We describe the two methods in more detail in the following sections, and explain how they lead to concurrent execution.

### Explicit Concurrency

In the developer's point of view, the most straightforward way to implement an extension to cognitive architectures might be creating a new field that will be interpreted in a certain way. The explicit specification of concurrency in ICARUS is a good example of that, and here we added a new field to its skill structure. When users intend to have certain subgoals executed in parallel, they can simply put the subgoals in this new field, and the system will consider them for simultaneous execution in a single cycle. In this section, we discuss the extended representation and its interpretation.

**Representation** We have seen skills for serial execution in Table 2. When the concurrent execution of subgoals is desired, one can use the new field, `:parallel` instead of `:subgoals` in skills. The syntax for this field is exactly the same as that of the `:subgoals` field, but the semantics are different. While the ordering of the elements in `:subgoals` reflects the priority, or simply the order in which they are to be executed, ordering in the `:parallel` fields do not have any meaning. The field simply lists parallel subgoals.

Table 3 shows the parallel version of one of the skills shown earlier. This modified skill, `ready-for-right-turn` means that the system should consider the subgoals, `in-rightmost-lane` and `at-turning-speed` simultaneously. This particular program is written so that the two par-

allel subgoals lead to different control actions at the leaf node of their skill paths, and therefore no conflicts occur during the execution of this skill.

Table 3: A sample ICARUS skill for parallel execution in the urban driving domain.

```
((ready-for-right-turn ?self)
 :percepts ((self ?self))
 :parallel ((in-rightmost-lane ?self ?l1 ?l2)
            (at-turning-speed ?self)))
```

**Interpretation** During the evaluation of its skills, the ICARUS architecture finds a path through its skill hiearchy that is executable. When it reaches at a skill with parallel subgoals, it continues evaluating each of them independently, as if each was a single subgoal at that point. Once all of the parallel subgoals are evaluated, it checks if each and every one of them either returns some actions or is satisfied in the current belief state. If this is true, the system simply collects the resulting actions.

For example, in Figure 2, the ICARUS agent has a top-level goal, `(on-street ME B)`. The system finds that it has a skill for the goal, with all the preconditions, `(intersection-ahead ME INT B)` and `(close-to-intersection ME INT)` met in the current state. The skill is a standard, serial skill, so the system selects the first unsatisfied subgoal, `(ready-for-right-turn ME)`. An available skill for this goal is the parallel skill shown in Table 3 that does not have any preconditions, and therefore the system considers its subgoals `(in-rightmost-lane ME L1 L2)` and `(at-turning-speed ME)` at the same time. All preconditions of the two skills for these goals are satisfied, so the system finds two executable skill paths, 0–1–2 and 0–1–3, for the top-level goal. It then executes implied actions from both paths, `*steer` and `*gas`.

## Concurrency through Resource Management

Another method for concurrency in ICARUS takes a more implicit approach. Execution in physical worlds mostly requires some type of resource, and the concurrent execution requires resources to support all the actions involved. Therefore, we can use resource as a measure to check any interference or conflicts between the candidates for concurrent execution. By tracking the resources required for each skill, the ICARUS architecture can automatically resolve any conflicts, giving skills with higher priority the opportunity to fire before any other lower priority skills that share some or all of the resources.

**Representation** This approach requires no changes to the syntax of the existing knowledge structures. However, the system now interprets all elements in `:subgoals` fields as candidates for concurrent execution, as well as the top-level goals an ICARUS agent might have. Users deposit subgoals in the existing field as usual, and the system finds opportunities for concurrency among them. Whenever the resources allow, it will take the opportunities found for execution.

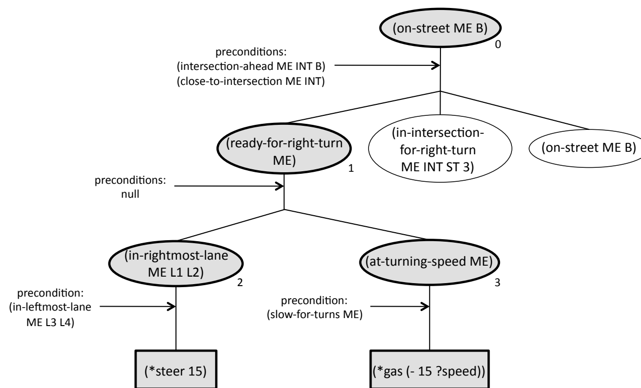



Figure 2: Multiple skillpaths that are executable in parallel, in the middle of a run in the urban driving domain. The agent's car is currently in a leftmost lane and moving slower than the desired turning speed. Ellipses denote goals, and rectangles represent actions. Numbers shown at the bottom-right corners of the goals are used to show skill paths in the main text.

**Interpretation** In this approach, ICARUS uses a new evaluation method during the interpretation of its skill knowledge. Here, the user pre-specifies the required resources for each action, and the system tracks the resources used for each primitive skill it evaluates. Instead of stopping the evaluation once an executable skill path is found, it marks resources for the primitive skill on the first executable path as *assigned*, and continues its search for the next path. It inspects the subsequent executable paths, and decide whether to allow their execution or not by checking the required resources for them. The ones that require any resource that is already assigned to prior paths are rejected. The system continues its search until either all available resources are assigned, or it reaches the rightmost branch of its skill hierarchy.

For example, the system finds the same skill paths, 0–1–2 and 0–1–3 shown in Figure 2, as in the first method. In this case, however, the system finds them not because the two subgoals at the third level, `in-right-most-lane` and `at-turning-speed` are marked as parallel, but because the actions they lead to, `*steer` and `*gas`, require two different resources. They require hands to control the steering wheel and the right foot for the gas pedal, respectively. The system first finds the path, 0–1–2 as it would in the standard system. But before searching through the hierarchy any further, it assigns the required resource, *hands*, to the selected `*steer` action. While continuing its search, the system finds another executable path, 0–1–3. It then checks if any of the resources this path requires are already assigned. When it finds the resource, *right foot*, is still available, it assigns that for the `*gas`

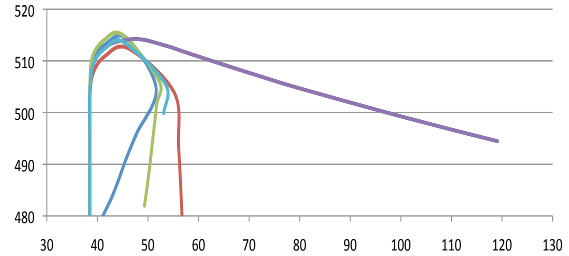
action. Once these two skill paths are found, the system stops its search through the skill hierarchy, since all the available resources are assigned.

## Experimental Observations

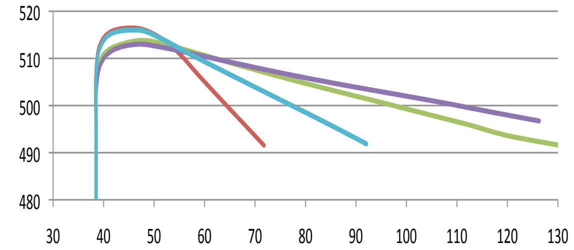
We hypothesize that the concurrent execution results in both qualitatively and quantitatively better behaviors. We also suspect that the system using the second method, namely, the one that uses the resource management scheme for concurrent execution will perform as well as the system that uses manually specified `:parallel` fields. However, our research is still in its early stages, and the urban driving domain is not yet ready for serious quantitative experiments. Therefore, in this section, we compare the serial system and the concurrent system by observing the quality of their behaviors in the urdan driving domain.

In this experiment, the agent starts at (38.5, 48.5), which is located in the rightmost lane on *A* street. Its goal is to be on *second* street, and for that, it needs to make a right turn at the upcoming intersection. In Figure 3, there is a building corner at the coordinate (50, 490), and therefore traversing in the rectangular region, $\{(x,y) \mid x > 50, y < 490\}$ is not allowed. With the serial execution system, the agent shows some insufficient behaviors as shown in (a), where it could not execute the turn properly and dangerously moving toward the building on four out of five times. The concurrent execution system using the resource management scheme, however, completes the right turn properly on four out of five times, as shown in (b).

Considering the fact that the agents in both cases do not have any low-level feedback controllers, the driving paths in (b) show significant improvements. We believe that the serial execution system fails in many cases because it could not accelerate by pushing the gas pedal during the steering action that reduces the speed of the car. In the concurrent execution system, however, both the steering action and the acceleration can happen simultaneously, resulting in turns that follow smoother arcs.




(a) Right turns by the serial execution system




(b) Right turns by the concurrent execution system that uses resource management

Figure 3: Some traces of the ICARUS agent's actual driving paths for a right turn, using the serial execution system in (a) and using the concurrent execution system in (b). The horizontal and vertical axes denote $x$- and $y$-coordinates, respectively. Each plot shows the results from five different trials.

## Related and Future Work

There have been various efforts to support concurrent execution and resource management in related fields. Freed's APEX (1998) manages resources and resolve conflicts using explicit descriptions in its procedural knowledge. With temporal constraints and priorities specified in its knowledge structure, the architecture provides a powerful execution mechanism for multiple tasks.

PRS (Georgeff & Ingrand, 1989; Myers, 1996) also provide a reactive execution capability that support multitasking. It uses special constructs in its goal structures, and controls termination and continuation of processes explicitly. Perhaps the closest work to PRS will be Firby's RAP system (1994). This system manages multiple continous processes that interact with each other, but in an implicit way. It does not use any explicit constructs that interrupt and resume processes.

More recently, Salvucci and Taatgen (2008) provided a broader view of the concurrent multitasking. They introduced the notion of 'threads' that resembles those in operating systems. As in our work on ICARUS, the work has its basis on a cognitive architecture, ACT-R (Anderson, 1993), providing better representation of multitasking behavior.

Compared to these systems, ICARUS controls concurrent subgoals more loosely, and it only restricts the execution of later tasks that require already assigned resources. However, the system imposes priorities to its goals, preferring the ones to the left side of its skill hierarchy. Since ICARUS' evaluation process happens from left to right of the hierarchy, it always executes higher-priority tasks first even under resource constraints. Interruption and continuation of tasks happen in an implicit way, again tied to the goal priorities. The system's reactive nature forces it to pause any ongoing tasks when higher-priority goals become false in its surroundings, and continue the paused tasks once these goals are achieved.

We are working to make this process more explicit, however. Through a mechanism for reactive goal nomination in the architecture, we have internally demonstrated that ICARUS can select and instantiate relevant top-level goals for the current situation. We also want this mechanism to handle priorities among the goals during this process. But instead of the priority computation in a cost-based approach (e.g.,

Freed, 2000), we plan to associate certain concept instances to each goal that will trigger nomination or abandonment of the goal.

The current version of ICARUS stores the concurrency it finds through resource management in a separate memory. Although we have implemented an initial version of the system that takes advantage of this knowledge, it cannot use the information unless an exactly identical situation arises in the future. We believe ICARUS should learn new skills or revise the existing ones based on the concurrency it finds. This way, the system will be able to have more general knowledge that can be useful in a later time.

Older versions of ICARUS could impose simple temporal constraints through disjunctive skills chained together through preconditions, but its representational power was limited. The current work on concurrent execution alleviates this limitation greatly, but still the system cannot express all possible temporal relations. An ongoing research involves an extension to the architecture that allows explicit specification of temporal constraints in both its concepts and skills.

## Conclusions

In this paper, we reported two extensions to the ICARUS architecture that support concurrent execution. The first method uses a new field in the system's procedural knowledge structures, which can be used to manually specify parallel subgoals. This method lacks of any facility to ensure that the subgoals are conflict-free, and it puts the burden on users. The second method solves this problem using resource management, and automatically finds parallel subgoals that can be executed concurrently. Combined with ICARUS' goal priorities, this method provides a promising way to model human behavior, especially in dynamic domains like the urban driving. But there are still more work to be done in this direction, and we hope to report results from our ongoing work in a near future.

## Acknowledgments

This research was funded in part by Grant HR0011-04-1-0008 from DARPA IPTO. Discussions with Pat Langley, Chunki Park, David Stracuzzi, and Dan Shapiro contributed to many ideas presented here.

## References

Anderson, J. R. (1993). *Rules of the mind*. Hillsdale, NJ: Lawrence Erlbaum.

Asgharbeygi, N., Nejati, N., Langley, P., & Arai, S. (2005). Guiding inference through relational reinforcement learning. In *Proceedings of the fifteenth international conference on inductive logic programming* (pp. 20–37). Bonn, Germany: Springer Verlag.

Choi, D., Kaufman, M., Langley, P., Nejati, N., & Shapiro, D. (2004). An architecture for persistent reactive behavior. In *Proceedings of the third international joint conference on autonomous agents and multi agent systems* (pp. 988–995). New York: ACM Press.

Choi, D., Morgan, M., Park, C., & Langley, P. (2007). A testbed for evaluation of architectures for physical agents. In *Proceedings of the aaai-2007 workshop on evaluating architectures for intelligence.* Vancouver, BC: AAAI Press.

Firby, R. J. (1994). Task networks for controlling continuous processes. In *Proceedings of the second international conference on ai planning systems* (pp. 49–54).

Freed, M. (1998). Managing multiple tasks in complex, dynamic environments. In *Proceedings of the fifteenth national conference on artificial intelligence* (pp. 921–927). Menlo Park, CA: AAAI Press.

Freed, M. (2000). Reactive prioritization. In *Proceedings of the second nasa workshop on planning and scheduling in space.* San Francisco, CA.

Georgeff, M. P., & Ingrand, F. F. (1989). Decision-making in an embedded reasoning system. In *Proceedings of the eleventh international joint conference on artificial intelligence* (pp. 972–978). Morgan Kaufmann.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Langley, P., & Choi, D. (2006a). Learning recursive control programs from problem solving. *Journal of Machine Learning Research*, *7*, 493–518.

Langley, P., & Choi, D. (2006b). A unified cognitive architecture for physical agents. In *Proceedings of the twenty-first national conference on artificial intelligence.* Boston: AAAI Press.

Langley, P., Choi, D., & Rogers, S. (in press). Acquisition of hierarchical reactive skills in a unified cognitive architecture. *Cognitive Systems Research*.

Myers, K. L. (1996). A procedural knowledge approach to task-level control. In *Proceedings of the third international conference on ai planning systems.*

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Salvucci, D. D., & Taatgen, N. A. (2008). Threaded cognition: An integrated theory of concurrent multitasking. *Psychological Review*, *115*, 101–130.