COORDINATED EXECUTION AND GOAL MANAGEMENT IN A REACTIVE COGNITIVE ARCHITECTURE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT
OF AERONAUTICS AND ASTRONAUTICS
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Dongkyu Choi

July 2010

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Stephen Rock, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Sanjay Lall

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Langley

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Coordinated execution and goal management are important human-level capabilities studied in cognitive science and psychology. Evidence suggests there are several different types of constraints that govern the coordinated execution of multiple, concurrent tasks. We took three of these types – shared objects, logical relations, and resource requirements – and developed an extension to a cognitive architecture, Icarus, that supports coordinated execution under such constraints. The extended architecture assumes that it should consider subgoals in its procedures concurrently unless the constraints prevent such execution. The existing framework supports constraints on shared objects and logical relations, but it requires a new facility to handle resource constraints. We specify resource requirements for each action that Icarus can use in the world, and the system assigns resources to executable skill paths during the skill retrieval process. The architecture prevents the execution of any skill path that requires already assigned resources and ensures that execution is coordinated according to the resource constraints.

We are also interested in the ability to manage goals that arise through changes in the environment. Such changes can lead people to pursue a new goal, abandon an old goal, and revise priorities among existing goals. This ability is especially important when executing multiple tasks at the same time, because concurrently pursued goals can interact with each other in both positive and negative ways. An agent should be able to give priority to more important ones if other goals are known to interfere. In

response, we made a second extension to the ICARUS architecture that lets the system manage its top-level goals in a reactive manner. The revised ICARUS distinguishes long-term and short-term goals, and it nominates instantiated versions of long-term goals when the state matches their corresponding relevance conditions. During the nomination process, the architecture computes the degree of match for the relevance conditions and uses it to modulate priority values of nominated goals. ICARUS revises the priority of nominated goals based on the modulated values.

To evaluate these extensions, we chose a variety of scenarios in an urban driving domain to demonstrate the new abilities. Due to the high-level nature of our research on cognitive architectures, the demonstrations focus on ability and parsimony of the extended ICARUS framework. We have also examined connections to earlier work on these topics and explored future directions for research.

Acknowledgments

This thesis would not have been possible without help from many people. My research advisor, Pat Langley, first introduced me to the fields of cognitive science and artificial intelligence and has greatly influenced my understanding of both since then. When I joined his laboratory at the Center for the Study of Language and Information back in 2003, Pat suggested that I work on the ICARUS architecture. From that point, my research has continued to fall within the context of ICARUS, leading to a variety of new components that fit under the hood of this interesting system. Pat is also an excellent writer who has spent a tremendous amount of time reviewing the drafts of this thesis.

My academic advisor and research co-advisor from my home department, Steve Rock, has influenced my approach to developing intelligent systems. Over the course of my study at Stanford, his classes laid the foundation for my background in control and heavily influenced my ideas about how to introduce concepts from control theory into ICARUS. Another member of my dissertation reading committee, Sanjay Lall, inspired my work on the matching of continuous concepts. One day, after studying his class material for modern control design, I suddenly realized how the feature could work and was able to sketch the basics in a relatively short time.

During my thesis defense, Nils Nilsson provided firm support for cognitive architectures, clarifying their advantages over statistical methods in artificial intelligence, but personally he inspired me much earlier. In my early days at Stanford, I attended

one of his talks, describing his groundbreaking work on Shakey, the robot, and his teleoreactive framework. Since then, his footsteps have guided my research and provided valuable lessons. In fact, the ICARUS architecture incorporates his ideas on teleoreactive systems into its core.

I am also grateful to Bernard Widrow, who kindly performed the duties of the committee chair for my thesis defense. His broad interest in adaptive systems and his open mind about related subjects gave me the courage to continue my work on ICARUS, which is an unusual topic for a graduate student in the Department of Aeronautics and Astronautics.

I should also express my gratitude to people in the Computational Learning Laboratory and the Institute for the Study of Learning and Expertise. Dan Shapiro has been present for the entire period of my study and has provided invaluable guidance and help. Two postdoctoral researchers, Will Bridewell and Tolga Könik, have given helpful practical advice on graduate study and have always been available for motivating discussions. I would also like to thank Will for very helpful comments on a draft of this thesis. My fellow doctoral students, Nima Asgharbeygi, Negin Nejati, and Chunki Park, have been there for me as friends and colleagues. We frequently engaged in discussions on a variety of subjects, regardless of their relation to my ICARUS work, and motivated each other during the long journey.

Most of the current work was completed while supported in part by the National Science Foundation, the Defense Advanced Research Projects Agency, and the Korea Institute of Science and Technology. During the last eleven months, I was supported from a project for the Office of Naval Research through the University of Illinois at Chicago. I would like to thank my supervisor there, Stellan Ohlsson, for supporting me in the process of finishing this thesis while working full-time. His vast knowledge of cognitive psychology and his work on constraint-based systems were also helpful. I

would also like to thank his wife, Elaine Ohlsson, for her comments on the coordinated execution chapter.

I want to express my gratitude to GarageGames for granting a free academic use of its TORQUE game engine. This product provides a reliable platform for fast and efficient development of simulation systems in three dimensions, and this helped me focus on content rather than technical details. I am also grateful to Ian Hardingham and Paul Taylor at mode7games. They added and improved functionalities in the urban driving simulator, satisfying our complicated requirements under limited time and budget.

To my friends, I cannot thank you enough for your encouragement and motivation: Jiyun, Youngjun, Euiho, Jaeheung, Jinwhan, Kihwan, Chunki, Minyong, Junkyu, Taehoon, and the rest of the KAASA members; Jiyoung and Jaeyoung from our laid back but fruitful book reading group; and my buddy since middle school, Youngung.

Last, but most important, I give thanks to my family. My wife, Sookyi, has provided endless love and support. Without her on my side, I would probably have not finished a single chapter. My new baby boy, Sunho, has been my greatest motivation to finish this thesis sooner rather than later. My parents and my brother back in Seoul have endured the long graduate years despite all the ups and downs, and I appreciate their love, support, and patience.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The human mind is an amazing machine that constantly processes information and controls our behavior at many levels. Research on cognitive architectures pursues the important dual goals of modeling the human mind and building artificial agents that exhibit human-level intelligence. To this end, these architectures are designed to provide broad accounts of human cognition, frequently characterized as the interplay of many distinct modules. Cognitive systems make assumptions about multiple aspects of mental computation, the details of which may vary significantly among different frameworks.

Nonetheless, all cognitive architectures subscribe to a handful of basic principles. First, rather than modeling isolated components of cognitive behaviors, such as perception, planning, or learning, they provide a unified theory of cognition (Newell, 1990). Second, cognitive architectures stay constant across different domains and provide an infrastructure for developing specific models or agents. Third, these architectures are also meant to exhibit the same qualitative behavior as humans, including the ability to show robust behavior when they face unexpected situations or errors. Lastly, many of them learn from their experience in an incremental manner that is interleaved with performance, which is another characteristic of human cognition.

1.1 The ICARUS Architecture

In this thesis, we build upon one such architecture, ICARUS (Langley & Choi, 2006b), which makes commitments to representations, memories, and processes that are similar to those in the Soar (Laird et al., 1986) and ACT-R (Anderson, 1993) architectures. The common assumptions include (a) ICARUS' distinction between short-term memories for dynamic information and long-term memories for stable knowledge; (b) memory elements that are cast as symbolic list structures; (c) cyclic operation that involves recognizing situations and acting in the world; and (d) the incremental learning of new structures. However, the architecture also makes some unique assumptions that distinguish it from its predecessors. ICARUS is specifically designed for agents that operate in physical domains, with all its knowledge structures being grounded in perception and action. Additionally, the architecture makes explicit distinctions between the concepts it uses to recognize situations and support inference and the skills to encode procedures, storing them in separate memories. Finally, ICARUS' long-term memories contain hierarchically organized contents that allow multiple levels of abstraction.

ICARUS operates in distinct cycles. At the beginning of each cycle, the system receives perceptual information from the environment. The percepts cause beliefs to be inferred based on the long-term conceptual knowledge. Once the architecture finishes deriving beliefs, it selects the first unsatisfied top-level goal and attempts to retrieve an executable path through its skill hierarchy for the current goal. If an applicable path exists, ICARUS takes it as the current intention, executes the selected path, and continues to the next cycle. The system tries to continue executing the intended procedure as long as it is executable, but, when the updated situation does not allow further continuation, ICARUS attempts to find a new skill path for the goal. When the system encounters an unfamiliar situation and it cannot find any

executable path for its goal, ICARUS invokes its problem solver. This mechanism uses a version of means-ends analysis to generate a solution to the new problem using existing knowledge as components. Whenever the system achieves its goal or any of the subgoals during the problem solving process, it uses the experience to acquire new skills.

Over the years, researchers have studied various aspects of cognition with the ICARUS architecture and applied it to many different domains. Choi et al. (2004) focused on ICARUS' persistent, but reactive, execution and studied the tradeoff between continuing ongoing procedures and reacting to the environmental changes. Choi and Langley (2005) and Langley and Choi (2006a) presented a version of means–ends problem solving and cumulative learning from solution traces in the Blocks World, FreeCell Solitaire, and a driving domain. Choi et al. (2007) studied the transfer of knowledge between similar but distinct scenarios in a first-person shooter game, *Urban Combat Testbed* (Youngblood et al., 2006), and Könik et al. (2009) worked on a similar topic using a platform for *General Game Playing* (Genesereth et al., 2005). Finally, Choi et al. (2009) used ICARUS to control a simulated humanoid robot for a modified Blocks World task. As seen, ICARUS provides a reliable framework that supports research on different aspects of cognition with many useful features.

1.2 Limitations of ICARUS

Although ICARUS incorporates powerful capabilities, experience with its use has revealed a number of limitations. One major deficit is that ICARUS can execute only one procedure at a time. In a given context, it finds a single executable path through its skill hierarchy and performs actions specified at the leaf node of the path. However, in some cases we want the system to execute multiple skills at the same time. For

example, for an ICARUS agent to hold an object with two manipulators, it should coordinate the manipulators simultaneously so that it does not drop the object. When the agent drives a car, it should coordinate its controls for the steering wheel and the gas/brake pedals to show reasonable behavior. But such concurrent multitasking requires not only the ability to execute multiple things at the same time but also the capability to coordinate the multiple actions under a variety of constraints.

Another drawback lies in ICARUS' handling of goals. Currently, the system takes its top-level goals as input from the system developer in their specific, instantiated forms. It reasons about goals and executes actions to achieve them, but it does not understand their origin or their ordering. ICARUS assumes that the goals are sorted in the order of their relative importance, and always achieves the earlier goals before acting on the later ones. This causes several problems. First, once the system starts execution with a given set of goals, it cannot deactivate or remove goals that become irrelevant later in the situation. Since the ordering, and hence the priority, of the goals is fixed, ICARUS cannot even work on later goals if an unsatisfied but irrelevant one exists earlier in the list of the top-level goals. Second, the architecture cannot react to changes in the world to adjust goal priorities. Many real-world situations require such changes. For instance, an ambulance driver normally observes all signals, but when an emergency strikes he must reorder his goals and react to traffic signals in different ways.

Finally, the ICARUS architecture can only match Boolean concepts. In many environments, this kind of matching suffices to recognize the situation properly. However, in other cases, the system should be able to detect more than true or false; rather, it needs concepts that return degrees of match. This becomes desirable when we want ICARUS to smoothly control the speed of a car while responding to subtly different situations. In such cases, a continuous (or discretized to a reasonably fine grain) recognition of its speed is crucial to compute the control input it should generate.

In this thesis, we respond to these limitations by extending the architecture along a number of fronts, as we summarize in the next section.

1.3 Main Contributions

We have made significant efforts to address the limitations in ICARUS explained in the previous section. The main scientific contributions of this thesis are:

- *Coordinated Execution.* We extended the ICARUS architecture for the ability to coordinate concurrent execution under different constraints. Although the research on concurrency and coordination has received much attention in computer science, it is less common in the literature on cognitive architectures. Only a few efforts have incorporated an architectural perspective (Freed, 1998; Georgeff & Ingrand, 1989; Myers, 1996; Firby, 1994; Salvucci & Taatgen, 2008). This thesis is the first effort to handle coordinated execution of concurrent multi-tasks in the ICARUS architecture. It is based on an analysis of constraints involved in such coordinated behavior, including shared objects, logical dependencies, and resource requirements. This work covers these constraints by extending the ICARUS representation and execution process. We demonstrate the performance of the extended system in urban driving scenarios, showing improved behavior with a smaller knowledge base that requires fewer concepts and skills than the original architecture.

- *Reactive Goal Management.* We developed a new capability for ICARUS that allows nomination, retraction, and prioritization of its top-level goals in reaction to the situation. The system incorporates long-term goals that resemble Ohlsson and Rees's (1991) conditional constraints, but also include attributes like default priority. ICARUS matches the relevance conditions of each long-term goal and

instantiates the goal predicate accordingly. It stores nominated goals in its short-term goal memory and sorts them in order of their relative priorities. The system calculates the priority value of each nominated goal based on its default priority and the continuous degree of match for its relevance conditions. We demonstrate this high-level behavior in scenarios from an urban driving domain, and show that the reactive management of top-level goals enables previously impossible behaviors.

- *Continuous Concept Match.* We extended ICARUS so that it partially matches concepts and returns a continuous value that indicates the degree of match. We used this feature in the context of goal prioritization, but there are many other potential applications of this capability. Concept definitions incorporate variables that should fall within specified intervals. Instead of returning false for a concept instance when a variable's value falls outside of the specified interval, the extended architecture lets the instance match partially in adjacent regions. ICARUS uses this degree of match to measure how completely the current state satisfies the particular instance. This measure has useful implications both in low-level control and in reactive goal management.

- *Urban Driving Domain.* We developed a simulation of urban driving to evaluate our extensions to the architecture. The domain involves a complex physical environment that provides challenges in both high-level cognition and low-level control. One can assign to an agent high-level missions such as patient transport in an ambulance and package delivery to different addresses. This domain provides a difficult and situationally rich testbed for research on cognitive architectures that integrate many components. Quantifying the effects of extending a cognitive architecture is often difficult and sometimes inappropriate. To evaluate our extensions to ICARUS, we take cues from Cassimatis et al. (2008)

and from Anderson and Lebiere (2003), who suggest a collection of qualitative measures. Following this tradition, we demonstrate the increased ability and parsimony of the extended system using scenarios in the urban driving domain.

In the next chapter, we introduce the urban driving domain we use throughout the thesis. Afterwards, we present a brief review of the ICARUS architecture, followed by a detailed description of our extensions for coordinated execution and reactive goal management. We report and discuss related work in the main chapters. In closing, we provide details about future avenues for research and offer some concluding remarks.

Chapter 2

An Urban Driving Domain

Cognitive architectures are integrated models of general intelligence, and they are often used to understand human-level intelligent behavior. As such, they encompass a large collection of functional components that process different forms of content. Several components cover high-level aspects of intelligence, while others deal with lower-level cognition and control. Therefore, evaluating such architectures naturally requires test domains that challenge their different aspects of cognition. Domains like the Blocks World, Tower of Hanoi, or FreeCell Solitaire provide challenges suitable to evaluate symbolic processing, but they abstract away aspects of physical control, assuming perfect and instantaneous manipulation of objects. In contrast, typical control domains like an inverted pendulum or robot arms pose continuous control problems in a physical environment, but they lack any high-level aspects like the reason such control is necessary. We want a complex physical domain that requires a system to handle both aspects, providing high-level challenges due to its complexity and creating opportunities for low-level, continuous, physical control.

Driving a car is one such task. This activity includes classic vehicle control problems like accelerating and decelerating comfortably, moving straight ahead, and turning smoothly through the simultaneous control of multiple effectors. Moreover, successful driving also requires the recognition of situations, decision making in reaction to one's surroundings, navigation to a target location, and so on. Although all driving tasks pose complex cognitive problems, urban driving offers a rich variety of scenarios that involve several different types of objects, including pedestrians, other cars, traffic signals and signs, and buildings with addresses. For this reason, we developed a three-dimensional simulated environment for urban driving for use in testing the ICARUS architecture. Some basic aspects of this simulation carry over from an older two-dimensional environment (Choi et al., 2004), but it features numerous improvements over its predecessor. In the following sections, we describe this simulated environment in detail.

2.1 Domain Simulation

We developed our simulation of urban driving from the ground up using the commercial game engine TORQUE (`http://www.garagegames.com/`). Figure 2.1 shows a screenshot of the simulated environment. This world models the downtown area of a city, which consists of square blocks populated with buildings and streets that run between them. Streets in this city include road segments, intersections, lane lines, crosswalks, and sidewalks. The streets are either horizontal or vertical as in a planned city, they lack dead-ends, and they can have multiple lanes. Each intersection may have traffic signals, stop signs, or none of these, and objects like vehicles and pedestrians may navigate the streets. Each block has several buildings that possess unique addresses on the particular street.

Cars in this city are governed by realistic dynamics that model mass, center of mass, aerodynamic drag, engine torque, brake torque, tire friction, lifetime, and so on. The vehicles have three control variables: the percentage that the gas and brake pedals are depressed and the angle of the steering wheel. The simulator directly controls most of the cars that drive around the city, accelerating them to a predefined speed, changing their lanes before making turns, and swerving them around slower cars. The simulated vehicles randomly turn at intersections and typically stop to avoid collisions. However, the cars violate these rules with certain probabilities, leading to collisions with each other or with other objects. When a car is stuck in a location due to such an accident, the simulator eventually respawns it at a random location.

Pedestrians, another type of dynamic object, are also controlled by the simulator. These roaming people initially appear on sidewalks, but they then move along the sidewalks and cross streets at designated crosswalks or near intersections. Pedestrians do not react to cars around them, leaving them to the mercy of drivers. As with the drone cars, the people can also violate rules depending on a specified probability of jaywalking. The greater this probability, the more likely that pedestrians will randomly cross streets and become involved in accidents. When a pedestrian is run over by a vehicle, the simulator removes it from the environment and spawns a new person at a random location to replace it.

2.2 Experimental Support

Like other domains used with Icarus, the urban driving domain supports two channels of communication. One transfers perceptual information from the domain to the architecture, whereas the other passes control actions from Icarus to the environment. Unlike some domains, especially those derived from existing games, the driving simulator is built from scratch with this communication interaction in mind, which

Figure 2.1: A screenshot from the urban driving domain.

gives us complete control over the agent's perceptions and abilities. These features facilitate the development and evaluation of Icarus driving agents.

When connected to the simulator, an Icarus agent drives a single car in the environment. The vehicle has the same dynamic properties as other cars in the city. From inside the car, the agent perceives various objects in agent-centered polar coordinates. The object attributes that Icarus perceives vary across different object types. Table 2.1 shows how the agent perceives particular objects in the simulation. For example, when the agent sees itself, it senses its name, speed, heading, steering wheel angle, percentage of throttle that is open, maximum throttle, percentage of brake applied, the street segment it is on, and the number of hits with pedestrians.

Similarly, when the agent perceives a lane line, it gets information on the name of the line, color, distance and angle from itself to the line, and the name of the segment to which the line belongs.

Table 2.1: Attributes of objects ICARUS perceives in the urban driving domain.

objects	percepts
itself	name, speed, heading, steering wheel angle, throttle, maximum throttle, brake, current street segment, number of pedestrian hits
drone cars	name, distance, angle, heading, speed, value
pedestrians	name, distance, angle, heading, speed, alive
street segments	name, street name, distance, angle
intersections	name, street name, cross street name, distance
lane lines	name, color, distance, angle, segment name
buildings	name, address, street name, distance to the closest corner, angle to the closest corner, distance to the second closest corner, angle to the second closest corner

In addition, the simulation offers actions that an ICARUS agent can perform in the world. Each action adjusts a set of control variables available in the simulation. Table 2.2 shows the actions provided by the domain. The first three of actions are necessary for ICARUS to drive, controlling the gas pedal, the brake pedal, and the steering wheel. Each action takes a single argument: the percentage of pedal application for the first two and the desired steering wheel angle for the last. Two additional actions are provided for convenience: one causes the car to coast (i.e., neither the gas pedal nor brake pedal is applied) and the other straightens the steering wheel. With these five actions, ICARUS controls its car in ways similar to a human driver.

The simulator models effects of these actions in realistic ways. The application of the gas and brake pedals are durative, as is the rotation of the steering wheel, potentially taking several cycles to achieve the desired application or angle. For instance, if the current angle of the steering wheel is 0° and the desired angle is 10°,

Table 2.2: Five actions ICARUS can perform in the urban driving domain.

actions	control variables		
	gas pedal	brake pedal	steering wheel
(*gas percent)	percent	0	N/A
(*brake percent)	0	percent	N/A
(*steer angle)	N/A	N/A	angle
(*cruise)	0	0	N/A
(*straighten)	N/A	N/A	0

the agent can reach the desired angle in one cycle. But if the current angle of the steering wheel is $-90°$ and the desired angle is $+90°$, then it might take two to three cycles to reach the desired angle, depending on the maximum rate of change allowed in the simulation. In addition, the pedals are spring-loaded, returning to zero after being pressed to some percentage. Therefore, the agent should continuously execute relevant pedal actions if it desires to maintain a constant depression.

In addition, the domain supports sophisticated parametric control over the simulation to facilitate its use as an evaluation tool. Through a user-friendly interface, experimenters can specify the number of horizontal and vertical streets, the length of each block, the number of drones and pedestrians, the probability of illegal moves for drone cars, the probability of jaywalks for pedestrians, the speed of simulation, the distance agents can see, the number of lanes on each street, and the type of traffic signals at the intersections. Developers can also choose to use a predefined placement of cars and pedestrians from a stored file or to let the simulation place them at random. This feature is useful for arranging particular starting situations to demonstrate relevant agent behavior. Figure 2.2 shows the user interface for setting the parameters in the city with slide bars, drop down menus, and check marks.

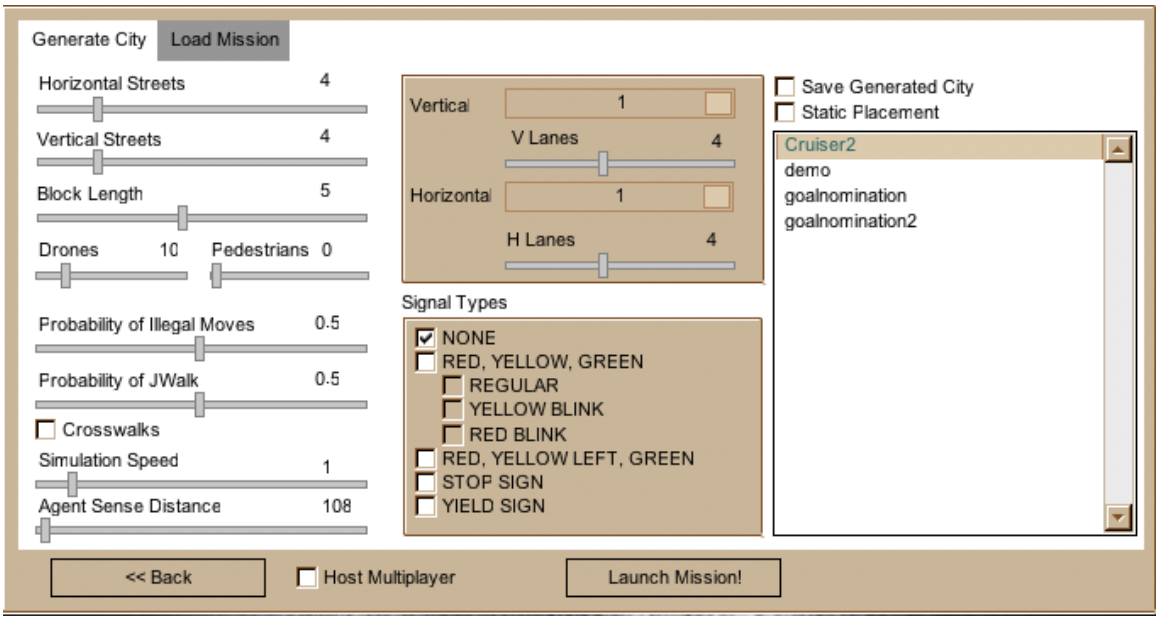

Figure 2.2: User interface for controlling city environment in the driving domain.

2.3 Advantages of the Domain

What makes this simulated environment interesting is that it supports a variety of interactions among objects. An agent driving a car in the city should watch for other vehicles, pedestrians, traffic signals, and stop signs. In some cases, it should also look at the addresses of buildings. The agent needs to take all these factors into account while driving around the city.

In addition, the agent may have higher-level motivations or goals for driving. For example, it might want to drive as fast as possible to decompress, it might want to deliver a package to some address, or it might need to take a patient to a hospital in an emergency. We can create numerous scenarios in the domain with a relative ease, since we use cars in so many different ways. It poses cognitive challenges, as it does in our everyday lives.

At the same time, the environment also requires driving agents to support low-level, continuous control. Using a set of controllers similar to those available in real

situations, an agent system should drive its car around the city without incident. Maneuvers like acceleration, deceleration, alignments, and turns all require feedback controllers of some sort to maintain good driving behavior.

This dual richness provides an interesting environment for testing different aspects of agents, while requiring only a handful of control inputs. Our experience in this domain helped us recognize important limitations in our architecture. Urban driving naturally requires coordinated action and provides a variety of interactions among objects, thereby motivating extensions to the framework that support concurrent execution and goal management. The domain also allows interesting scenarios in which we can demonstrate such extensions.

In summary, we believe that urban driving offers a familiar but rich dynamic environment that requires both high-level decision making and low-level control. For this reason, it provides an excellent testbed for unified cognitive architectures that allows the evaluation of capabilities at different levels within a single setting. This is why we rely exclusively on the TORQUE driving simulation to evaluate behaviors of agents embodied in ICARUS, which we review in the next chapter.

Chapter 3

Review of the ICARUS Architecture

ICARUS grew out of the cognitive architecture movement, and it shares basic features with other frameworks like Soar (Laird et al., 1986) and ACT-R (Anderson, 1993). Like these architectures, ICARUS makes commitments to a particular representation and interpretation of knowledge, along with memories that support them. One common feature is that the architecture distinguishes between short-term and long-term memories. The contents stored in these memories are symbolic list structures that can be composed dynamically during performance and learning. Second, cognitive processing occurs in cycles. On each cycle, the architecture accesses its long-term structures through pattern matching and instantiates them to perform actions relevant to the current state. Finally, ICARUS has the ability to learn incrementally by accumulating symbolic structures in its long-term memory.

However, ICARUS also has distinctive characteristics. The architecture is specifically designed for physical domains, and its mental structures are ultimately grounded in perception and action. ICARUS distinguishes conceptual and skill knowledge that it stores in different memories, although their elements refer to each other. The contents of ICARUS' long-term memories are organized in a hierarchical manner, allowing

different levels of abstraction. The system features reactive but goal-directed skill execution, but it falls back on problem solving if it hits an impasse, which in turn leads to learning.

Researchers have introduced a number of extensions to ICARUS over the years, including methods for learning from observations and learning from failures, but in this chapter we review the central aspects of the architecture, starting with the representational components and continuing with the processes that operate over them.

3.1 Representation and Memories

As noted earlier, ICARUS distinguishes between conceptual and skill knowledge, as well as between short-term and long-term entities. Figure 3.1 shows four different memories that result, organized along the two dimensions. A conceptual long-term memory stores descriptive knowledge structures that are similar to Horn clauses. These structures, which we call *concepts*, specify generic situations in the environment and fall into two categories. A *primitive concept* describes a class of situations in the world in terms of objects' attributes, their values, and relations among them. This type of concept can include constraints on these values in the form of numerical and logical restrictions. In comparison, *non-primitive concepts* describe more complex situations by specifying relations among other concepts.

ICARUS' skill memory stores knowledge structures similar to STRIPS operators. We call such structures *skills* and use them specify general procedures. Skills are indexed by the goals that they aim to achieve when executed to completion.[1] ICARUS distinguishes skills in much the same way as concepts. *Primitive skills* describe the effect of actions in the world under specified conditions stated as conceptual relations among objects. In comparison, *non-primitive skills* provide a subgoal decomposition, which leads indirectly to other skills, under similar relational conditions.

[1] In later sections, we note important implications of this feature for execution and learning.

The primitive and non-primitive structures in concepts and skills result in hierarchically organized knowledge bases that describe situations and activities at different levels of abstraction. ICARUS can express an abstract situation in terms of lower-level concepts and it can store a complex procedure with distinct steps in terms of subgoals that invoke lower-level skills.

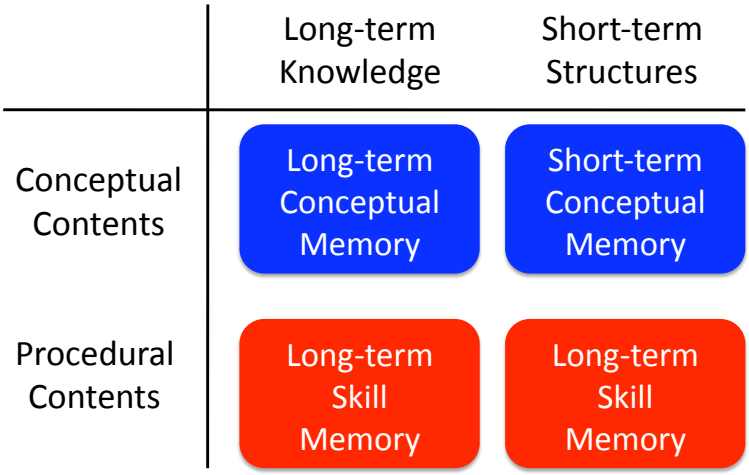

Figure 3.1: The four main memories in ICARUS.

Tables 3.1 and 3.2 show some examples of concepts and skills for the urban driving domain. The first two concepts, *yellow-line* and *at-turning-speed*, are primitive and have a *:percepts* field that lists observable objects and their attributes. These concepts also specify conditions in their *:tests* fields that enforce constraints on the variables involved. In comparison, the last concept, *ready-for-right-turn*, refers to two other predicates, *in-rightmost-lane* and *at-turning-speed*, which makes it a non-primitive concept. Similarly, the first skill in Table 3.2, *in-intersection-for-right-turn*, is primitive in that it mentions only actions the agent can apply directly in the world. However, the other two skills are non-primitive, since they specify subgoals that must be achieved by other skills. Furthermore, in ICARUS two or more skill clauses can

have the same head, supporting both disjunction and recursion as seen in the last example. In this manner, the architecture encodes complex descriptions of possible situations in the world and possible courses of action.

Table 3.1: 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 3.2: Sample ICARUS skills for the urban driving domain.

```
((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)))

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

((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)))
```

The architecture also has short-term memories related to concepts and skills. The short-term conceptual memory stores instantiated concepts for the current situation.

These concept instances, which we call *beliefs*, take the form of predicates and associated arguments that the agent believes to hold in the environment. Table 3.3 shows some sample contents of the short-term conceptual memory.

Table 3.3: Sample beliefs stored in ICARUS' short-term conceptual memory.

```
(aligned-and-centered-in-lane me yellow1_3866 white3_4085)
(aligned-in-lane me yellow1_3866 white3_4085)
(almost-aligned-in-lane me yellow1_3866 white3_4085)
(centered-in-lane me yellow1_3866 white3_4085)
(clear me c4713_1_1)
(closest-lane-on-right me white3_4085 sidewalk2_2774)
(facing-street me b)
(in-lane me yellow1_3866 white3_4085)
(in-leftmost-lane me yellow1_3866 white3_4085)
(in-segment me s2543)
(intersection-behind me sb1first first)
(lane yellow1_3866 white3_4085)
(lane white3_4085 sidewalk2_2774)
(lane-on-right me white3_4085 sidewalk2_2774)
(leftmost-lane yellow1_3866 white3_4085)
(not-emergency me)
(on-right-side-of-road me)
(on-street me b)
(rightmost-lane white3_4085 sidewalk2_2774)
(segment-behind me s1922)
(segment-behind me s1878)
(sidewalk sidewalk2_2774)
(slow-for-cruise me)
(slow-for-turns me)
(steering-straight me)
(yellow-line yellow1_3866)
```

In a separate short-term memory, ICARUS stores instantiated skills. These skill instances, which we call *intentions*, are in the form of special constructs that include a goal, a set of variable bindings, and a list of satisfied preconditions. Unlike the short-term concept memory, these are structured around *goal stacks*. At the first level of a goal stack is an agent's top-level goal. Each level in a stack includes a subgoal of the main goal and a skill instance or intention that the system retrieved for the subgoal. Agents with multiple top-level goals will have multiple stacks in this memory. In this manner, the memory serves as a storage for ICARUS' goals, intentions, and other execution-related information. Therefore, we sometimes refer to it as a *goal memory*.

For example, Table 3.4 shows a sample goal stack stored in the goal memory at a certain stage of execution in the urban driving domain. The stack contains a top-level goal or *objective*, *(on-street me b)*. The system's intention for this goal is to use a skill clause with id 18. The skill is non-primitive and it refers to subgoals that include *(ready-for-right-turn me)*. For this subgoal, ICARUS intends to execute a skill with id 17 that, in turn, leads to another subgoal, *(at-turning-speed me)*. The architecture has selected a primitive skill clause with id 4 to achieve goal, which includes an action *(*brake 20)* in its body.

Table 3.4: An example of a goal stack stored in ICARUS' goal memory for the top-level goal, *(on-street me b)*.

```
objective:  (on-street me b)
skill path:  ((on-street me b) (ready-for-right-turn me) (at-turning-speed me))
skills:  #18 for (on-street me b)
           bindings:  ((?int . s1671) (?street . third)
                       (?st . a) (?self . me) (?target . b))
           satisfied preconditions:  ((no-road-ahead me)
                                      (intersection-ahead me s1671 third)
                                      (close-to-intersection me s1671)))
       #17 for (ready-for-right-turn me)
           bindings:  ((?self .  me))
           satisfied preconditions:  nil
       #4 for (at-turning-speed me)
           bindings:  ((?speed) (?self .  me))
           satisfied preconditions:  ((fast-for-turns me))
           actions:  ((*brake 20))
```

3.2 Inference and Execution

As mentioned above, ICARUS operates in cycles. On each pass, the system perceives sensory information from the environment, infers beliefs based on the resulting percepts and available concepts, and retrieves a skill that is relevant to the inferred situation and the current goal. Figure 3.2 shows the overall operation of the architecture and the interactions among its memories through various processes. Shaded

rectangles denote memories and ovals represent the processes that operate over them. Arrows show the direction of information flow between the memories and processes.

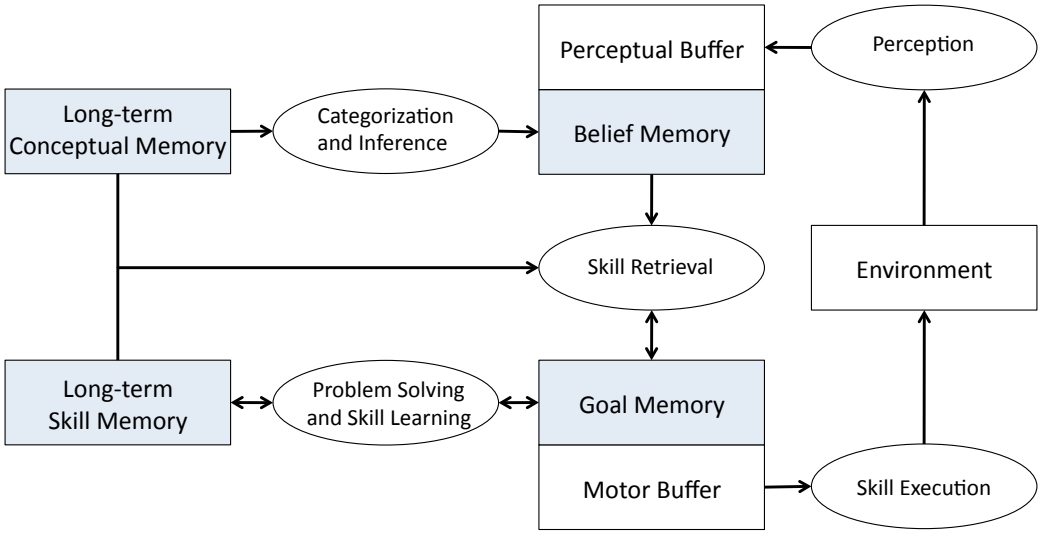

Figure 3.2: Interactions between ICARUS' memories through different processes.

When ICARUS receives percepts from the environment, it deposits them into its perceptual buffer. Each element in this buffer specifies the type and name of an object, along with a set of attribute-value pairs. For example, in the urban driving domain, ICARUS receives percepts of objects like *(self me speed 5.0 heading -90.0 wheel-angle 0 throttle 10.0)* and *(lane-line yellow1_3866 color yellow dist -16.5 angle 0.0 segment S2543)*.

The architecture attempts to match the concept definitions stored in its long-term conceptual memory against these perceived objects, instantiating the variables from definitions with object names or the values of their attributes. For instance, ICARUS would match the first concept shown in Table 3.1 against the lane-line object mentioned in the previous paragraph, binding the variable *?line* to *yellow1_3866* and checking that the *color* attribute is *yellow*. When such pattern matches are successful,

as in this example, ICARUS instantiates the head of the concept and deposits it into short-term conceptual memory. In this case, the system will add *(yellow-line yellow1_3866)* into the memory. As shown in Figure 3.3, ICARUS performs this process starting with the lowest level structures, the primitive concepts, and moves up the hierarchy to non-primitive concepts until it infers all the instances that hold in the situation.

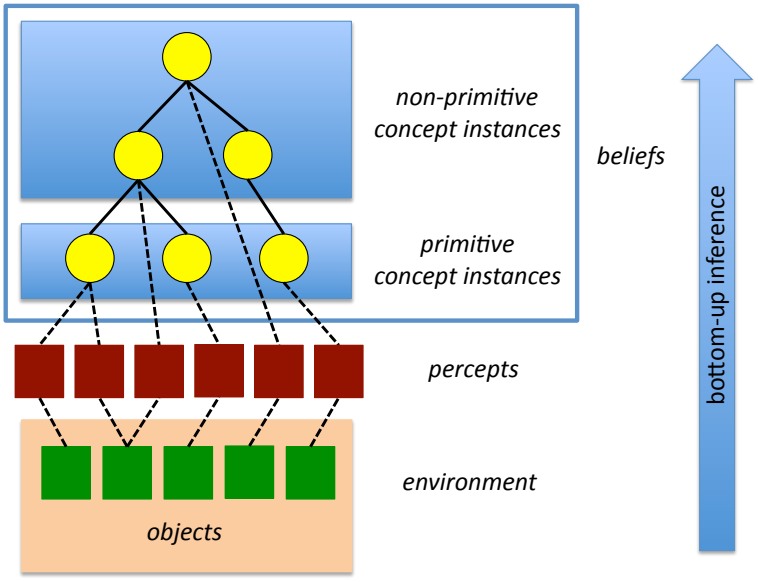

Figure 3.3: Bottom-up inference of beliefs in ICARUS.

To handle the combinatorial nature of this matching process, the inference module compiles and records dependencies among concept definitions on the initial cycle and associates inferred concept instances with the corresponding definitions. On subsequent cycles, the architecture checks whether perceptual and conceptual supports for a concept instance still hold and updates its belief state accordingly. Since the inference module updates its beliefs in this manner rather than starting from scratch

on every cycle, it achieves a significant speedup after the initial cycle.[2] The network of dependencies ICARUS maintains for this purpose is similar to that in the Rete algorithm (Forgy, 1982), while the matcher's operations resemble those of truth maintenance systems (Doyle, 1979).

Once the system infers the current belief state, it selects the first unsatisfied top-level goal to drive the execution process. Then ICARUS retrieves a skill from long-term memory that would achieve this goal and makes the instantiated skill its current intention. During this process, ICARUS performs a form of *flexible skill retrieval*. This involves looking for skills with the current goal in their heads but, if no such skills exist, also considering skills that achieve a goal that subsumes the current one. The system then evaluates the skills until it finds one that is *executable* in the current state, preferring more recently acquired skills. A non-primitive skill is executable when all its preconditions are met and there exists at least one executable skill that will achieve its first unsatisfied subgoal. Primitive skills have no subgoals, so they are executable if their preconditions are met. As shown in Figure 3.4, the system searches through its skill hierarchy in a top-down manner to find 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 that specifies actions the system should execute in the environment. ICARUS applies these actions to alter its surroundings. In turn, the system's perceptual input on the next cycle will change, and the system repeats the above processes based on the new sensory data.

[2] Asgharbeygi et al. (2005) offer another response to the problem of slowed inference, in which the system learns to prioritize different concept instances for inference. Their system first infers the instances that it has learned are more important in an attempt to provide possibly incomplete, but still useful, belief states even under time constraints.

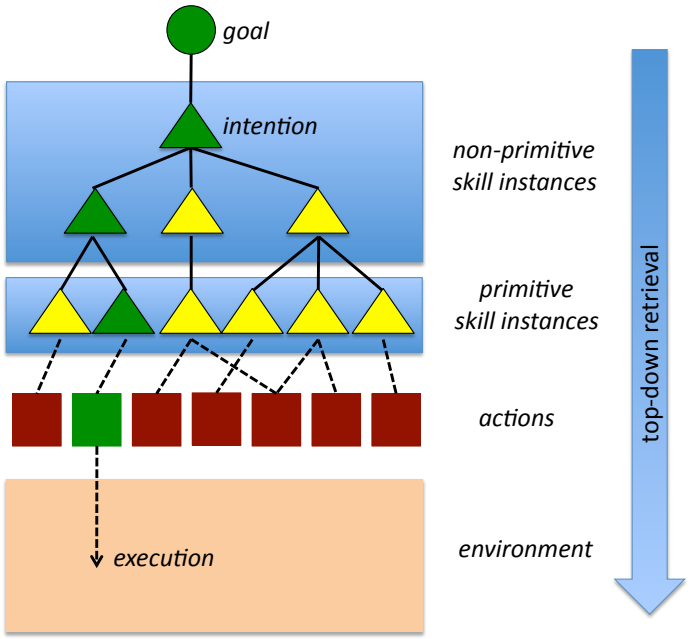

Figure 3.4: Top-down retrieval of skills in ICARUS.

3.3 Problem Solving and Learning

When ICARUS encounters an unfamiliar situation, it cannot find any executable path through its skill hierarchy. In such cases, the architecture attempts to achieve its goals using a version of means-ends problem solving. As ICARUS achieves each goal or subgoal, it can learn from the experience by creating new skills. Although we do not use this part of the architecture in this thesis, it is an important component of ICARUS and deserves some explanation.

To solve unfamiliar problems, the architecture chains backward from its goal using a concept or skill definition, with skills taking precedence over concepts. Figure 3.5 shows how these problem-solving chains work. In a chain that involves a skill definition, the system uses a skill that is known to achieve the current goal but whose precondition is not met by the beliefs. The problem solver assumes that each skill has

a single precondition, which may be a higher-level concept that implies more than one subcondition. ICARUS pushes the unsatisfied precondition onto its goal stack for further problem solving. In a chain that uses a concept definition, the system decomposes the goal into its subgoals. ICARUS chooses one of the unsatisfied subgoals and pushes it onto the goal stack, as with skill chains. Once it pushes a subgoal onto the stack, the system applies its problem solving steps recursively, until it reaches a subgoal with an associated skill that it can execute immediately. If the system hits a dead end during this process, it pops the last level of the goal stack and backtracks to try other alternatives.

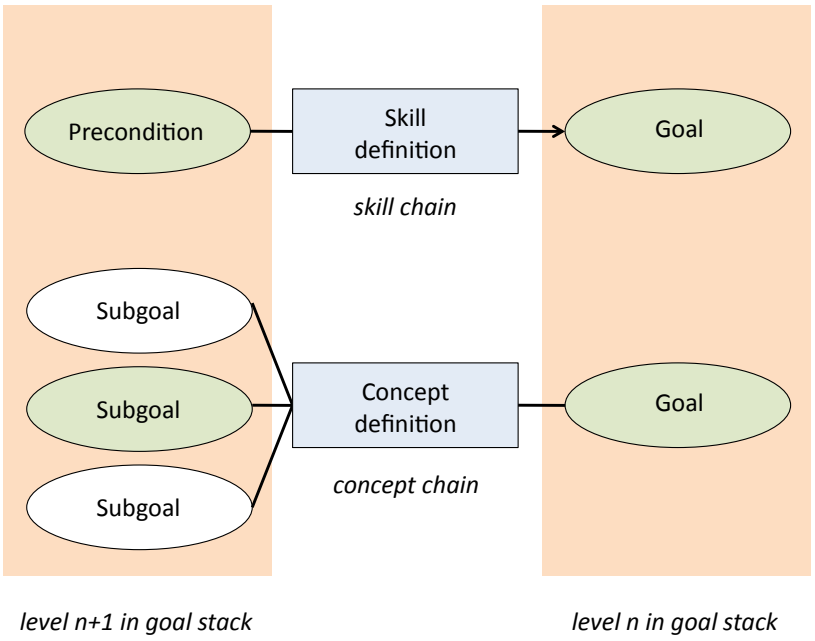

Figure 3.5: Concept and skill chains in problem solving.

Whenever ICARUS achieves a subgoal, it is presented with a learning opportunity. It can acquire a new skill based on the problem-solving trace generated. The new skills are non-primitive ones that use the lower-level subgoals as components. The system composes new skills differently, depending on the problem-solving chains used.

3.4 An ICARUS Driving Agent

Elsewhere, we have described ICARUS agents for a number of different domains (Choi & Langley, 2005; Langley & Choi, 2006a; Choi et al., 2007; Langley et al., 2009; Könik et al., 2009), but in this thesis we focus on an agent that operates in the urban driving environment described earlier. To determine whether ICARUS supports interesting high-level behavior in this domain, we first needed to create a program that exhibits basic driving capabilities. These include vehicle acceleration and deceleration, cruising at constant speed, aligning to lane lines, lane changes, and turns. To this end, we wrote an ICARUS program with 70 concepts and 32 skills that covers these basic abilities.[3] Among them, 18 concepts and 22 skills are primitive. The concept hierarchy is five levels deep and the skill hierarchy is three levels deep, the latter with some recursive structures. With this program, ICARUS can maintain its cruising or turning speed, stay on the right side of the street, keep aligned and centered in a lane, change lanes to the left or right, make a right turn, and stop its vehicle.

To illustrate how the agent operates, consider the task of reaching street *B*, the upcoming cross stree, when starting from a location on street *A*. We would give the top-level goal, *(on-street me B)*, for which the system retrieves a non-primitive skill that has a satisfied precondition, *(intersection-ahead me int1 B)*, and three subgoals, *(ready-for-right-turn me)*, *(in-intersection-for-right-turn me int1 A B)*, and *(on-street me B)*. The ICARUS agent selects the first unsatisfied subgoal, *(ready-for-right-turn me)*, and retrieves the only skill that achieves it. This skill has a null precondition and two subgoals, *(in-rightmost-lane me line1 line2)* and *(at-turning-speed me)*. The system again selects the first unsatisfied subgoal, *(in-rightmost-lane me line1 line2)*, for which it has an executable skill that invokes an action, *(*steer 15)*. The agent achieves this subgoal after executing the primitive skill for several cycles and shifts to

[3] Appendix A provides concepts and skills for the complete program.

the second subgoal. When the subgoal is achieved, the higher-level subgoal, *(ready-for-right-turn me)*, also becomes true, and ICARUS continues to work on the later subgoals at this level until it achieves its top-level goal to be on street *B*.

The quality of driving behavior varies slightly on machines with different speeds, but, on a typical dual-core computer, this minimal agent usually manages to execute the basic maneuvers more or less competently. However, due to variations in the simulation and in the ICARUS–simulator interface, the agent does not always perform such maneuvers successfully, sometimes colliding with buildings or driving off streets. We might offset these problems by adding more knowledge, say by making finer distinctions about the vehicle's situation. However, this requires additional effort on the programmer's part, and increasing the number of skills can lead to slower responses. As we discuss in the ensuing chapters, the limitations observed in the basic agent's behavior suggest the need for substantial extensions to ICARUS that overcome these problems.

Chapter 4

Coordinated Execution

4.1 Motivation and Background

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. When they are driving, they steer their cars while pushing the gas pedal. Sometimes these concurrent activities can be serialized: 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 move properly if the driver cannot steer and push pedals at the same time.

Our research focuses on modeling human behavior of this sort within a cognitive architecture (Newell, 1990). As an infrastructure for modeling general intelligence, such an architecture must support simultaneous, concurrent execution. We study this problem in the context of Icarus, which we have just reviewed in the previous chapter. We have shown that this architecture can support intelligent agents in a number of domains, including Blocks World, FreeCell (Langley et al., 2009), and General Game Playing (Könik et al., 2009). But when used in more dynamic domains like the urban driving domain we reviewed in Chapter 2, it becomes clear that serial

execution significantly limits the system's ability. For example, an ICARUS agent that delivers packages in a city should perform maneuvers like turning left or right. To execute a turn properly, it should continuously adjust both its speed and its steering angle. However, a serial system can perform only one of these activities at a time, and it often fails to steer while focusing attention on speed or vice versa. Instances of this sort motivated us to extend the ICARUS architecture to support concurrency.

There are at least two distinct facets of concurrency: recognizing concurrent behavior and executing concurrent actions. The former involves detection, analysis, and explanation of multiple things happening simultaneously, possibly over contents of an episodic memory (Tulving, 1972). This requires a mechanism to keep track of what is true and what is not over a period of time. In the ICARUS framework, Stracuzzi et al. (2009) proposed a representation that describes exact durations over which predicates hold in the environment and provide a basis for recognizing temporal relations among multiple predicates.

However, we need a different type of mechanism for the execution of concurrent actions. Rather than focusing on the exact timing of multiple things that happen, we should address the problem of coordinating concurrent execution subject to various types of constraints. In this chapter, we discuss new mechanisms that we have incorporated into the ICARUS architecture. However, before we begin, we should examine several issues related to the coordination of concurrent execution, including constraints related to shared objects, logical dependencies, and resource requirements.

First, we recognize the important role of objects themselves in constraining and coordinating concurrent execution. The physical world around us requires coordination among objects. For example, a basketball player might hold the ball with his both hands before throwing it to the goal. A golfer would hold her club with her hands in a way that best suits her swing. We can find many cases that involve coordinating two or more parts of our body around an object or objects.

As in sports, however, coordinations involve more than just objects. For instance, a tennis player should throw a ball in the air and swing her racket when the ball is at a certain height to serve it. To hit the ball accurately, she should control her body parts to work around the objects involved in a way that aligns the center of the racket and the ball. We can express constraints like this as logical dependencies and impose the alignment as a precondition on the skill for a service. Within the ICARUS framework, A being a precondition of B requires the temporal order A then B, not B then A or both A and B at the same time. When a high-level goal can be decomposed into subgoals, then there might be dependencies among them or among their own subgoals. Logical dependencies require certain temporal layouts restricting how the system should execute such subgoals in time.

Meanwhile, resource constraints like having only one manipulator on a robot also affect how the system executes. For example, if the robot has a block in its manipulator hand, it cannot grasp another until it releases the existing block in its hand. Or if a driver is using his both hands to turn the steering wheel, he cannot reach out to grab something in the glove box unless he releases a hand from the wheel. Finally, there are purely temporal constraints that specify certain amounts of timed delays or consider a specific points in time. For example, when we cook, we might have something simmer in the pot for 20 minutes while working on another dish that requires frying. When we make an appointment to be somewhere at a certain time, say, one o'clock in the afternoon, we mean a specific point in time, and we should coordinate actions according to that.

As seen so far, many aspects of the world (including the agent itself) affect coordinated behavior. Rather than attempting to develop a general theory that covers them here, we focus on the implementation of the first three constraints within the architectural framework. In the following sections, we first analyze briefly the different constraints related to coordination and provide an overview of the extended

Icarus. Then we explain how the new system handles different types of coordination and provide the details of the extensions required for that purpose. Afterwards, we demonstrate the new capability in the urban driving domain. We conclude by reviewing related work and summarizing the main contributions.

4.2 Analysis of Coordinated Execution

Before we describe how Icarus handles coordinated execution, a brief analysis of coordination in general will be helpful to understand the problem better. We are especially concerned with the coordination for concurrency in the context of execution, which requires a mechanism that is generally less complicated than the recognition of concurrency. This is because coordinated execution per various constraints leads to concurrent behavior without explicitly laying out the temporal configurations of each action. For recognition, Stracuzzi et al. (2009) proposed the use of two different time stamps that represent the starting point of the execution for a goal and the time when the goal is achieved. But we do not necessarily need such an explicit representation of time points to coordinate the execution of multiple actions, except for cases in which we must specify an absolute point in time. We study the reasons why something can (or cannot) be done at the same time and how these factors affect execution, rather than focusing on the mechanical issue of timing different actions. Furthermore, we may be able to set the starting point of a skill, but it does not make sense to control the time point for completion in this context, since we have no control over the speed of progress in skill execution. With these points in mind, we investigate different aspects of coordination mentioned earlier, namely object, logical, and resource constraints.

Perhaps the most basic type of constraint for coordination is that of shared objects. Many coordinative behaviors related to manipulation are subject to constraints of

shared objects. In sports, we can see many such examples, like handling a ball or a racket with both hands. Although control mechanisms for our body parts seem quite modular, they are all connected at the highest level by a central nervous system, and we can coordinate the multiple parts to manipulate an object. Information about the object is shared across different parts involved, so that they all act toward our goal for the object. An architecture for physical domains should have a facility to handle these shared objects that impose constraints on the control of different body parts of the agent involved.

Logical constraints have a different character that is often hidden beneath the goals we have for different objects. For example, if you are going on a road trip, you might have goals like *luggage packed*, *luggage loaded in the car*, and *car at destination*. You can immediately see that you should achieve the goals in that order, not in any other way. You know this because loading luggage in the car assumes that the luggage is packed and you need the luggage in the car at the destination. This is obvious to humans, but it is not so apparent to an artificial agent with the three goals. Some goals need to precede other goals because they are preconditions of the other goals. In a simple Blocks World task to build a tower with block A on B and block B on C, the ultimate goal is often expressed as the conjunction of *(on A B)* and *(on B C)*, but the agent should achieve them in a particular order, namely, *(on B C)* first and then *(on A B)*, which is not explicitly described. An agent architecture should have a facility to handle such constraints during execution.

Another type of constraint is related to resources. Our notion of resource is not limited to physical types such as fuel and manpower. Rather, it covers a broader range of items that the agent needs to achieve some goal, including perceptual, computational, and physical elements. For example, when we are trying to find an address on a street, we use a "gaze" resource to perceive the address on buildings, computational

resources to match observed addresses against the target address, and a mobility resource (e.g., feet) to move. If we are not familiar with the place and require a map for navigation, we will need to use the gaze resource to read the map, and we might alternate between looking at building addresses and the map. These resource constraints affect what we can do at the same time and what we cannot, and its impact on execution is significant. This is all the more important for agents operating in physical domains where various kinds of physical resource are important. To be complete, an architecture needs facilities that handle all these constraints for coordination. Next, we discuss how we extended ICARUS to support them.

4.3 Overview of the Extended Architecture

The original ICARUS architecture has potential to support the constraints we have seen involved in coordinated execution. In fact, it can already deal with shared objects and logical constraints. ICARUS' variable matching capability provides a mechanism for handling shared objects, and its ability to specify preconditions and goals of skills suffices to enforce logical constraints. Figure 4.1 shows how the original architecture supports these constraints. For shared objects (see (a) in the figure), the system matches a variable (e.g., *?var*) against an object (e.g., A) in the environment at a certain level of its skill hierarchy. Then it carries the binding over to the lower levels, so that the two (or more) skill paths that are distant from each other act on the same object. ICARUS also supports logical constraints within the existing framework. As shown in (b) in the figure, two (or more) distant skill paths may have a logical relationship through a condition (e.g., G). The condition can serve as a goal on one path, while it acts as a precondition on another, forcing the first path execute before the second one and preventing them to fire at the same time.

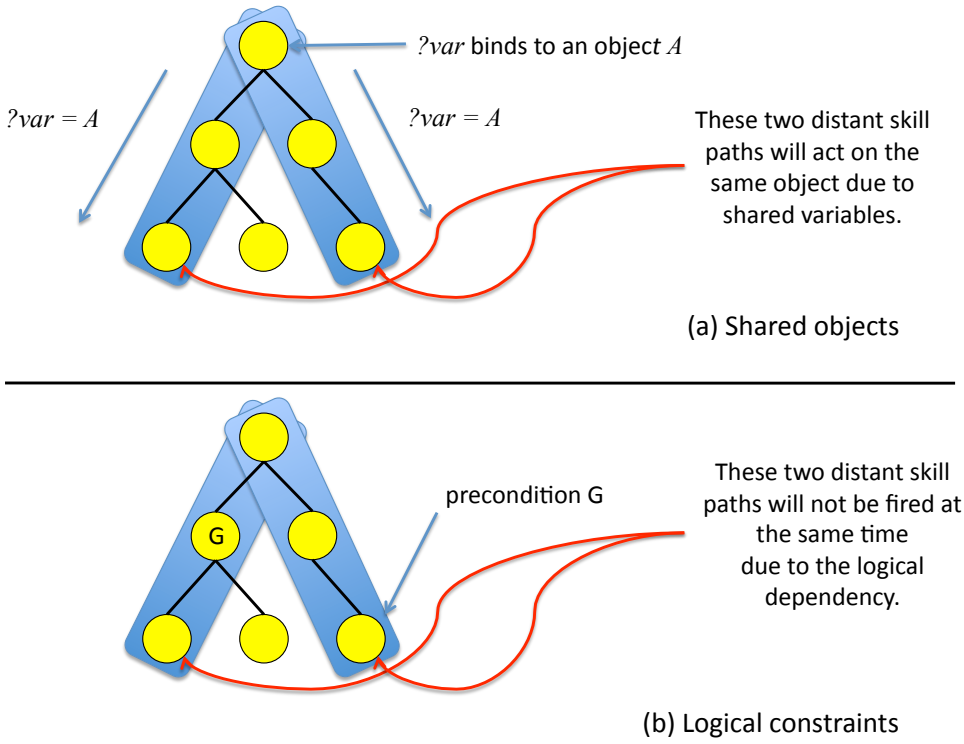

Figure 4.1: The original ICARUS architecture includes variable matching and explicit expression of preconditions and goals that support shared objects and logical constraints with only minor changes.

Although the original architecture has these facilities, we have never used them in the context of coordination. This is because the architecture could not retrieve multiple skill paths, thereby disallowing any coordinated behavior. Therefore, our initial effort to bring coordinated execution into the ICARUS framework focused on a modified interpretation of subgoals in ICARUS' skill knowledge and a new mechanism that allows retrieval of multiple skill paths for coordination. This extension unlocks the power of existing facilities like variable matching and logical specifications in skills, and it lets the architecture deal with such constraints related to shared objects and logical dependencies.

On the other hand, ICARUS needs an additional extension to deal with resource constraints. This should involve assigning available resources and checking resource conflicts during the retrieval of multiple skill path. Programmers define resource requirements of actions ICARUS can use in its primitive skills. Upon finding an executable skill path that leads to certain actions, the system checks the resource requirements for the actions and assigns available resources to the path. Should there be any requirement for resource that is not available at that moment, ICARUS rejects the skill path for execution. In the following two sections, we provide more details of these extensions.

4.4 Retrieval of Multiple Skill Paths

In the extended system, we make a slight modification from the original ICARUS representation for skills, namely, the *:subgoals* field that specifies ordered subgoals. We relax the ordering constraint in this field, thereby allowing retrieval of multiple skill paths for coordinated execution. The subgoals specified in this field become candidates for coordinated execution, and the architecture attempts to find any concurrently executable subgoals. However, this does not mean that all subgoals are treated equal. The system still imposes priorities on them, with subgoals that come earlier in the field taking priority over ones that come later.

Table 4.1 shows one of the skills we considered, *ready-for-right-turn*. This has two subgoals, *in-rightmost-lane* and *at-turning-speed*, which the original ICARUS would consider one after the other. However, we can see that these two subgoals are independent, since the former deals with steering wheel and the latter uses gas and brake pedals, which are controlled by hands and feet, respectively. By relaxing the ordering constraint on these subgoals, the architecture can have the opportunity to consider executing for the two subgoals at the same time if constraints allow such execution.

Table 4.1: A sample ICARUS skill in the urban driving domain with the potential for concurrent execution.

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

During the skill selection process, ICARUS attempts to find a path through its skill hierarchy that is executable. When it reaches a non-primitive skill with subgoals, it takes the first subgoal and continues evaluating until it reaches the bottom of the skill hierarchy. Once it finds an executable path, rather than stopping further evaluations as the original architecture would, the new system continues to the next executable path. As long as all constraints are satisfied, it sweeps through its skill hierarchy from left to right and finds all skill paths for concurrent execution.

For example, in Figure 4.2, 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 belief state. The skill is non-primitive, so the system selects the first unsatisfied subgoal, *(ready-for-right-turn me)*. An available skill for this goal has two subgoals that are concurrently executable (in this case, satisfying a resource constraint). Therefore, the system considers both subgoals, *(in-rightmost-lane me L1 L2)* and *(at-turning-speed me)*. The preconditions of the two skills considered for these goals, *(in-leftmost-lane me L3 L4)* and *(slow-for-turns me)*, are satisfied, so the system finds two executable skill paths (marked as 0–1–2 and 0–1–3 in the figure), from the top-level goal. It then executes direct actions from the both paths, **steer* and **gas*.

With this ability to retrieve multiple executable paths through its skill hierarchy, ICARUS is one step closer to coordinating multiple actions. What it also needs is the

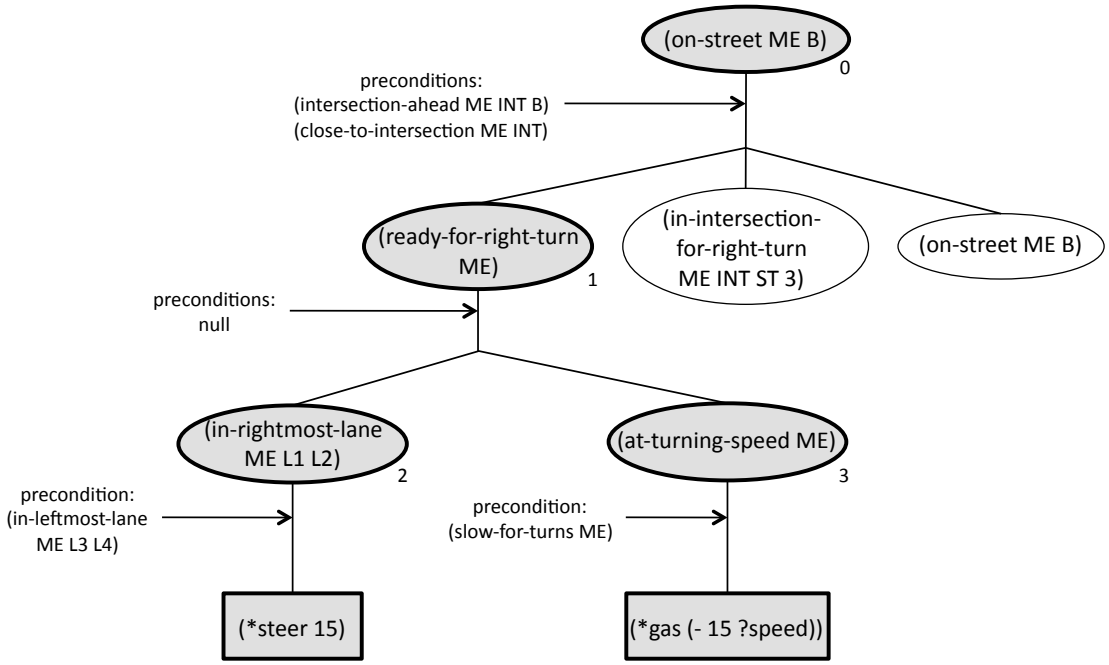

Figure 4.2: Multiple skill paths that are executable in parallel during a run in the urban driving domain. The agent's car is currently in the leftmost lane and moving slower than the desired turning speed. Ellipses denote goals and rectangles represent actions. Numbers shown at the bottom-right corners of some goals show skill paths in the main text.

capability to enforce different constraints on these actions to achieve coordinated behavior. In the next section, we explain how the extended Icarus deals with different types of constraints and coordinates actions according to them.

4.5 Constrained Coordination

Adding the ability to retrieve multiple skill paths to Icarus allows coordination involving shared objects and logical constraints without any further extensions. This

is due to existing abilities like variable matching and explicit description of preconditions and goals. However, another type of coordination related to resource constraints requires an additional extension to manage resources during skill execution. In this section, we cover the details of how ICARUS handles these coordinated behaviors.

4.5.1 Shared Object Coordination

Matching variables to objects in the environment and their attributes is a fundamental capability in ICARUS. This lets the architecture react to the environment while maintaining its goal-directed behavior. During skill selection, the system starts with a top-level goal and evaluates skills for it. ICARUS matches variables at higher levels and passes them down through different branches of the hierarchy. By doing so, it ensures that the shared objects are used consistently at lower levels of the hierarchy, even when the branches are long and they are connected only at the highest levels.

This top-down propagation of variable bindings is crucial to coordinating behaviors at different branches of the hierarchy. Without such checks on consistency, multiple actions scattered at various places in the hierarchy might locally bind to different variables, making it impossible to coordinate them. A good example is the lane lines in the urban driving domain. When a driving agent decides to change its lane to the rightmost one on the current street, it perceives a pair of lane lines that uniquely identify the rightmost lane, like *line1* and *line2*, and binds the corresponding variables to these objects. The two skills involved in the lane change behavior, *(in-lane ?self ?line1 ?line2)* and *(aligned-and-centered-in-lane ?self ?line1 ?line2)*, must work on the same lane lines. The system ensures that all its efforts are coordinated by passing the bindings for *line1* and *line2* down to the lower level of its skill hierarchy, and it changes the agent's lane to the right, then aligns and centers the agent's car in the rightmost lane.

4.5.2 Logical Coordination

For logical coordination, the ability to specify preconditions and goals of skills explicitly is crucial. Unlike procedural rules used in many cognitive architectures, ICARUS skills are indexed by the goals they achieve in their heads, making the architecture explicitly goal-directed. Preconditions of skills occur in a separate field in skill structures, and they are combined when skill paths are dynamically generated. When the system considers multiple executable paths through its skill hierarchy, it checks all the preconditions on the paths against the current belief state, and it verifies any logical dependencies specified as preconditions or goals, even when the paths are not close to each other within the hierarchy.

In this manner, ICARUS ensures that multiple skill paths it retrieves always satisfy any logical constraints specified in the skill hierarchy, so that the system does not perform procedures out of the logical order. For example, the second level subgoals shown in Figure 4.2, *ready-for-right-turn*, *in-intersection-for-right-turn*, and *on-street* have logical dependencies that require execution in that order. During skill evaluation, the system might consider a path that achieves the first subgoal and a path that leads to the third subgoal for concurrent execution. Without the proper logical dependencies specified, it would find the two paths executable at the same time, resulting in a mixed behavior that is out of logical order. The preconditions associated with each skill ensure that ICARUS will not execute multiple skill paths with any logical inconsistency.

4.5.3 Resource Coordination

The coordination of resources differs from the other two types of coordination described above, in that it requires an additional extension to ICARUS. Execution in physical worlds typically requires some type of resource, such as effectors that can do

only one thing at a time, and coordinating multiple actions requires their management. The new system uses resource as a measure to check for interference or conflicts among candidate skills being considered for coordinated execution. By tracking the resources that each skill requires, the extended architecture can automatically resolve such conflicts, while still giving priority to skill paths on the left side of the hierarchy that logically precede ones on the right side that share resources. This approach requires no changes to the syntax of the existing knowledge structures. During the evaluation of skill paths, the system takes the opportunities found for coordinated execution whenever it has resources available for them.

This depends on the developer to specify the resources required for each action, and the system tracks the resources used for each primitive skill it evaluates. Instead of stopping the evaluation once it finds an executable skill path, it marks resources for the primitive skill on the first executable path as *assigned* and continues its search for additional paths. The architecture inspects other applicable paths and decides whether to allow their execution by checking their resources. ICARUS rejects paths that require any resource that is already assigned to prior paths. The system continues this process either until it has assigned all available resources, or until it has considered all paths through its skill hierarchy.

For example, let us revisit the case in Figure 4.2. We give two available resources, "hands" and "right foot," to the ICARUS agent. As in the original architecture, ICARUS finds the skill path, 0–1–2 first, but the new system checks resources required for this path before considering additional paths. At the end of the skill path is a primitive skill, *in-right-most-lane*, that uses a direct action in the world, **steer*. The system knows that the action requires a resource, "hands," and that it is available. So ICARUS assigns the resource to this skill path and continues its search through the skill hierarchy. Then the system finds another path, 0–1–3, that uses the action **gas*. This action requires a different resource, "right foot," that is still available. After

assigning the resource to the second skill path, ICARUS detects that it has no other resources available for further assignment. As a result, the system stops its search for additional paths and performs all the actions found, after which it starts a new cycle.

4.6 Architectural Implications

We have discussed two important extensions to the ICARUS architecture. One of them enables the system to retrieve multiple paths through its skill hierarchy during execution, and the other allows resource-based coordination of concurrent skill paths. Neither of them require any drastic changes to the existing formalism, but their implications are significant. The ability to retrieve multiple skill paths unlocks ICARUS' hidden capacity to coordinate concurrent execution. With this extension alone, the architecture can already control the execution of multiple skill paths with respect to shared objects and logical constraints. ICARUS' existing capabilities, namely variable matching and explicit description of preconditions and goals, play a significant role here, by providing two types of constraints the architecture can use when coordinating concurrent execution.

The second extension for resource management adds another important constraint for ICARUS' use. During the skill selection process, which now allows the retrieval of multiple paths, the new system checks resource requirements of each skill path it finds and assigns those resources to the path if they are available. ICARUS considers executing the skill paths it finds later in the process only if they do not require any resources that are already assigned. Despite its notable impact on the architecture, this extension does not require any changes to the representation of concepts and skills. It only requires straightforward specifications of each action's resource requirements, which domain experts can do in advance. This means that the programmers do not have any additional work to use this new capability if the domain is well understood.

Another implication, which seems more important from an architectural perspective, is the impact on programming style. Although the extensions do not require any changes in how we write ICARUS programs, it promotes a new programming style that lets the developer specify maintenance objectives as top-level goals of the agent. For instance, suppose that we have a maintenance goal to keep the vehicle at a certain cruising speed. The original architecture forced us to put a subgoal, like *at-cruising-speed*, everywhere throughout the skill hierarchy, so that the agent could consider increasing or decreasing its speed at any point. This required redundant programming and it was prone to error. In the extended architecture, we would instead put this goal before other goals at the top level, giving higher priority than the goals related to other maneuvers. We could not do this before, since the original architecture would get stuck in such a generally applicable, higher-priority goal and have limited access to the skills for other maneuvers. The new system considers more than just the first executable path and has a constant access to the overall skill hierarchy subject to certain constraints. The resulting skill hierarchy is much simpler, while it maintains comparable or improved behavior.

We believe that such improvements are mainly due to the fact that the system has improved control through an effective increase in the control frequency. To show smooth behavior comparable to that of humans, the ICARUS architecture should run at cycle frequencies on the order of 10 to 100 Hz, but, in practice, factors like conceptual inference and skill selection slow things down significantly. Through concurrent execution, the system can deal with multiple issues on each cycle without the additional burden of repeated inferences, effectively increasing the frequency of the control loop. As we will see in the next section, this affects the behavior of ICARUS agents substantially.

We also argue that this extension makes the ICARUS architecture more psychologically plausible. When people encounter situations that would require concurrent

multi-tasking, they attempt to understand what can be done in parallel and what cannot. The extended architecture carries out a similar process through the dynamic conflict resolution of multiple tasks according to shared objects, logical, and resource constraints. In the next section, we provide more detailed comparisons of the two architectures on this dimension.

4.7 Demonstration of New Capabilities

We hypothesize that the concurrent execution results in both qualitatively and quantitatively better behavior. But this kind of improvement at the architectural level does not lend itself to traditional quantitative evaluation, due to its general impact on the architecture. As discussed previously, the extensions promote a new programming style that makes direct comparison between the original and extended architectures even more difficult. Still, some type of evaluation is necessary to verify the improvements such extensions bring, and researchers have proposed several approaches. Cassimatis et al. (2008) have suggested ability, breadth, and parsimony as measures for computational accounts of higher-order cognition like ICARUS. Anderson and Lebiere (2003) have described the 'Newell test' for a theory of general cognition and suggested associated qualitative characteristics for evaluation. In this section, we demonstrate the advantages of the extended framework by comparing it to the original architecture in two urban driving scenarios, with respect to the qualitative measures presented in these works. Especially, we focus on improvements in control ability and program parsimony.

4.7.1 Scenario 1: Expressway Cruiser

The simpler of the two scenarios we cover here involves cruising on a stretch of street in the urban driving domain. This scenario involves no traffic signals, but

the agent must still perform various maneuvers like speeding up, coasting, changing lanes, and stopping. There are drone cars and pedestrians that occasionally hinder the agent's cruising and force it to take evasive moves. Although ICARUS programs for this scenario are relatively simple to allow quick demonstrations, it lets us see clear differences between the original and extended architectures.

In the initial ICARUS, we would program an agent for this scenario in the following manner. We provide two top-level goals like *(all-clear me)* and *(cruising-in-lane me ?line1 ?line2)*. To reflect the fact that avoiding other cars and pedestrians is more important than cruising at a designated speed, we would put the first goal before the second one so that the system gives the former priority. In the second top-level goal, we would deliberately include the two variables, *?line1* and *?line2*, that designate the lane to cruise in, so that the agent would have the liberty to change lanes when its current lane is blocked by obstacles. Skills required for speeding up, coasting, changing lanes, and stopping would be included under the skill for the top-level goal. Tables 4.2 and 4.3 show some sample concepts and skills written for the original architecture, respectively.

Concepts shown in these tables are what we use to describe the first top-level goal of the agent,[1] *all-clear*, which implies that there are neither other vehicles or pedestrians blocking the agent's current lane. The second and the third concepts also act as preconditions of the skills for the top-level goal, shown as the first three skills in Table 4.3. To avoid other cars in front, the agent uses the second and the third skill to change its lane to the left or right.

The rest of the skills specify methods for changing lanes, ensuring that the agent maintains its speed and steers in the relevant direction at the same time. The original

[1]Note that the last concept, *in-lane*, describes a more general situation than others and that it is also used for the second top-level goal.

Table 4.2: Sample ICARUS concepts for the original architecture for the Expressway Cruiser scenario in the urban driving domain.

```
((all-clear ?self)
 :percepts  ((self ?self))
 :relations ((not (pedestrian-ahead ?self ?obj))
             (not (vehicle-ahead ?self ?obj))))

((pedestrian-ahead ?self ?ped)
 :percepts  ((self ?self)
             (pedestrian ?ped dist ?dist angle ?angle alive ?alive))
 :tests     ((= ?alive 1)
             (< ?dist 20)
             (< ?angle 30)
             (> ?angle -30)))

((vehicle-ahead ?self ?car)
 :percepts  ((self ?self)
             (car ?car dist ?dist angle ?angle))
 :tests     ((< ?dist 70)
             (< ?angle 30)
             (> ?angle -30)))

((in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self segment ?sg)
             (lane-line ?line1 segment ?sg)
             (lane-line ?line2 segment ?sg))
 :relations ((lane ?line1 ?line2)
             (line-on-left ?self ?line1)
             (line-on-right ?self ?line2)))
```

architecture requires these disjunctive skills to achieve a reasonably well-executed lane change. We decomposed the procedure into four different cases:

- a lane exists on the left and the agent is not veering to the left;
- there is a lane on the right and the agent is not veering to the right;
- a lane exists on the left and the agent is steered to the left; and
- there is a lane on the right and the agent is steered to the right.

We need tedious disjunctions like these because the system allows only one execution per cycle. To avoid losing the agent's speed when steering left or right, we need the last two skills to adjust its speed during the move. But even with these additional skills, the original architecture often exhibits a 'zigzag' motion due to the repeated switches between steering and acceleration.

Table 4.3: Sample ICARUS skills for the original architecture for the Expressway Cruiser scenario in the urban driving domain.

```
((all-clear ?self)
 :percepts ((self ?self))
 :start    ((pedestrian-ahead ?self ?ped))
 :actions  ((*brake 1000)))

((all-clear ?self)
 :percepts ((self ?self))
 :start    ((vehicle-ahead ?self ?car)
            (closest-lane-on-left ?self ?line1 ?line2))
 :subgoals ((in-lane ?self ?line1 ?line2)))

((all-clear ?self)
 :percepts ((self ?self))
 :start    ((vehicle-ahead ?self ?car)
            (closest-lane-on-right ?self ?line1 ?line2))
 :subgoals ((in-lane ?self ?line1 ?line2)))

((in-lane ?self ?line1 ?line2)
 :percepts ((self ?self))
 :start    ((lane-on-left ?self ?line1 ?line2))
 :requires ((not (steering-to-left ?self)))
 :actions  ((*steer -35)))

((in-lane ?self ?line1 ?line2)
 :percepts ((self ?self))
 :start    ((lane-on-right ?self ?line1 ?line2))
 :requires ((not (steering-to-right ?self)))
 :actions  ((*steer 35)))

((in-lane ?self ?line1 ?line2)
 :percepts ((self ?self))
 :start    ((lane-on-left ?self ?line1 ?line2)
            (steering-to-left ?self))
 :subgoals ((at-cruising-speed ?self)))

((in-lane ?self ?line1 ?line2)
 :percepts ((self ?self))
 :start    ((lane-on-right ?self ?line1 ?line2)
            (steering-to-right ?self))
 :subgoals ((at-cruising-speed ?self)))
```

The complication is not limited to this particular example. Rather, it is a general problem we have experienced with the original architecture, especially at lower levels of the control hierarchy, where independent control of multiple variables is often required. Whenever this occurs, the system must consider several subgoals at the same time, and it requires similar case-by-case disjunctions of skills that cover all possible combinations.

In contrast, the extended system requires no such complicated disjunctions of skills to handle multiple subgoals. In the Expressway Cruiser scenario, the extended system maintains speed independently during various maneuvers. Therefore, it shows better

performance than the original system without the need for the last two skills shown in Table 4.3. Instead, the system relies on the skills for its second top-level goal, *cruising-in-lane*, which, without any changes from the ones for the original system, already maintains speed of the agent's car.

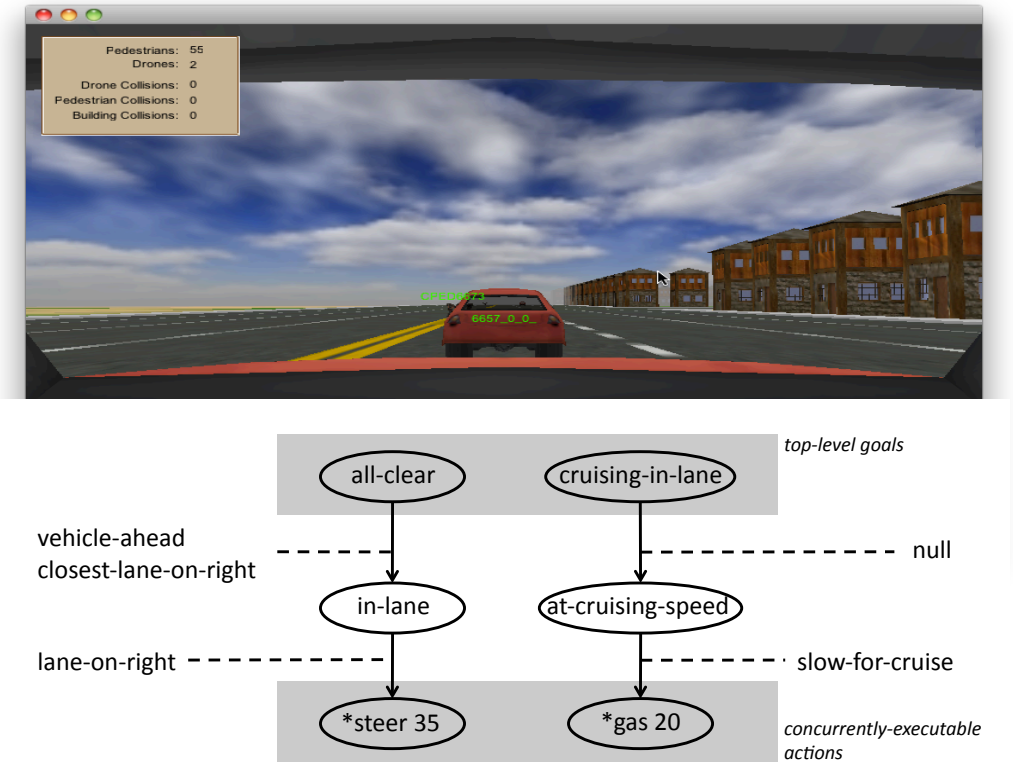

Figure 4.3: Concurrent execution paths found by the extended ICARUS for the Expressway Cruiser scenario.

As an example, let us consider a typical run in this scenario. The ICARUS agent starts in the leftmost lane on street *B*. There are no obstacles in front of it, at least initially, so the first goal of the agent, *(all-clear me)*, is automatically satisfied without any action from the agent. So it focuses on the second goal, *(cruising-in-lane me ?line1 ?line2)*, and accelerates to reach its cruising speed for ten cycles. On cycle 11, however, it finds a car, *c6120*, blocking its way, and the first goal is no longer

satisfied. The system therefore considers this goal and executes a lane change to the right with its *in-lane* skill. This takes nine cycles to complete, during which the agent concurrently accelerates to continue working on its second goal. On cycle 20, the agent reaches the target lane and starts aligning itself to that lane. At this point, the first goal is again achieved and the system can focus on the second goal. Soon, however, the agent finds another car blocking the lane and decides to execute a lane change to the left. From cycle 23, it starts to steer to the left, while occasionally pushing the gas pedal to maintain its speed for the second goal. The agent finishes this maneuver by cycle 38 and starts cruising along the street again.

Figure 4.3 shows the multiple skill paths the system finds during this run when there is a vehicle in front of the agent's car. The first top-level goal of the system, *all-clear*, leads to the intention to change its lane to the right, which, in turn, invokes an action, *(*steer 35)*. Meanwhile, the system finds another skill path starting from its second top-level goal, *cruising-in-lane*, that brings the intention, *at-cruising-speed*. At the end of this path is an action, *(*gas 20)*, that speeds up the vehicle. If given the same skill set, the original architecture would execute only the first path, and attempt to steer to the right without increasing its speed. Of course, this would not work well, since the car would be stuck at the current location steering its wheels to the right without enough speed. The original system has little or no chance to succeed in such scenarios if left without the additional skills described earlier.

4.7.2 Scenario 2: Local Cruiser

Our second scenario involves similar cruising behavior, but this time occurs on a local road with traffic signals that challenge the system with a more complicated situation. Each intersection has a tricolor signal that changes between red and green with intermediate yellow states. The driving agent must now not only watch for other cars and pedestrians, but also take the signals into account. For this scenario, we

provided the concepts and skills shown in Tables 4.4 and 4.5, in addition to the ones used for the previous scenario.

The knowledge responsible for behaviors like obeying signals is quite abstract. The agent uses the basic driving knowledge we encountered in the previous chapter and in the first scenario, but it uses them differently by invoking them selectively at the top level. The major improvements from the concurrent execution capability occur at the lower level, and this provides the same advantages as in the first scenario. In this scenario, however, the advantage is more apparent, since the agent must stop and go more frequently than before, exposing it to more cases that require accelerating while steering. The scenario also adds different combinations, such as decelerating while aligning to a lane to stop at an intersection with a red light.

Table 4.4: Additional ICARUS concepts for Local Cruiser scenario. Concepts for basic driving are omitted for the sake of simplicity.

```
((all-okay-to-go ?self)
 :percepts  ((self ?self))
 :relations ((okay-to-go ?self ?signal)))

((all-okay-to-go ?self)
 :percepts  ((self ?self))
 :relations ((not (signal-ahead ?self ?signal))))

((signal-ahead ?self ?signal)
 :percepts  ((self ?self)
             (signal ?signal color ?color
                     exitpath ?street angle ?angle))
 :relations ((on-street ?self ?street))
 :tests      ((< ?angle 0)))

((okay-to-go ?self ?signal)
 :percepts  ((self ?self)
                (signal ?signal color ?color))
 :relations ((signal-ahead ?self ?signal))
 :tests      ((equal ?color 'green)))
```

Let us analyze the behavior our extended architecture shows in this scenario in more detail. Each run is slightly different due to the timing of signal changes, but a typical one goes as follows. The agent starts at one end of street B with three different

Table 4.5: An additional ICARUS skill for Local Cruiser scenario. Other skills remain the same as in the first scenario.

```
((all-okay-to-go ?self)
 :percepts ((self ?self)
            (signal ?signal))
 :starts   ((signal-ahead ?self ?signal))
 :actions  ((*brake 25)))
```

goals at the top level, including *(all-okay-to-go me)*, *(all-clear me)*, and *(cruising-in-lane me ?line1 ?line2)*. It does not see any obstacles ahead and it does not see a signal, so the first two goals are already satisfied and the agent focuses on the third goal. It accelerates to reach cruising speed for nine cycles, thereby achieving this goal. On cycle 10, however, it finds a car, *c61067*, blocking its way and the second goal becomes unsatisfied. Now the system considers both the second and the third goals, performing a lane change to the right and accelerating at the same time to compensate for the slowing caused by the steering action. Maneuvering around the car takes 12 cycles, but the agent soon finds another one ahead on the new lane, *c61074*, on cycle 29. This time, it avoids the car by shifting into the left lane, again changing steering and accelerating concurrently.

On cycle 30, even before the agent finishes aligning itself to the new lane, it sees a traffic signal at the upcoming intersection that is turning from green to red. This causes the concept instance *(all-okay-to-go me)* to disappear, which leads the system to consider both the first and last goal. To achieve the first one, it executes the skill shown in Table 4.5, and it continues aligning itself to the lane for the third goal. These two happen at the same time, with the car slowing down while steering to the right to align itself correctly. The light takes more than 15 cycles to change again, and, on cycle 47, the agent starts moving again upon seeing the green light. It continues accelerating until reaching its cruising speed, then continues down the street while occasionally adjusting its alignment and speed.

4.8 Related Work on Coordinated Execution

Many previous studies have influenced our research on coordination of concurrent execution. Psychologists have investigated extensively how humans deal with concurrency. As Meyer and Kieras (1997) summarize in their excellent review of the literature, researchers have proposed a variety of models for human multi-task performance. The *single-channel hypothesis* (Telford, 1931; Craik, 1948; Vince, 1948; Welford, 1952) assumes a channel of central mechanisms that governs mental processes. This channel can process and respond to one stimulus at a time, causing delayed responses in multi-task cases. Before the current work, ICARUS operated exactly in this fashion, assuming a single channel of processes for inference, skill selection, and execution.

However, findings that contradicted the single-channel hypothesis led to *bottleneck models* (Broadbent, 1958; Smith, 1967; Welford, 1967; Keele, 1973; Pashler, 1984). These maintained there is a bottleneck among the cognitive processes of perception, decision making, and manipulation, although researchers did not agree on exactly where the bottleneck exists. The three processes map precisely onto ICARUS' perception and belief inference, skill selection, and execution. Unlike bottleneck models, the architecture performs the inference of beliefs based on all the available perceptual data before it considers the given tasks. The two later processes, skill selection and execution, can happen in parallel on a single cycle for multiple goals, as long as resources allow. Hence the architecture's bottleneck for multi-task performance lies in its execution mechanism.

Other researchers proposed a general-purpose central processor with limited capacity (Kahneman, 1973; Norman & Bobrow, 1975). They assumed that there is a single limited cognitive resource that can be divided and assigned to specific processes. The limit on the resource was not fixed, but multiple tasks inevitably make fewer

resources available to each, delaying their responses. Other scientists proposed multiple resource models that assume various disjoint sets of resources (Navon & Gopher, 1979; Wickens, 1984). These were capable of explaining many experimental results, and they aligned with neurological findings to some degree, but the unrestricted addition of different resources to the models received strong criticism. ICARUS' notion of cognitive and physical resources is consistent with these models. Regardless of the nature of these resources, the architecture can assign them to executable paths it finds in the skill hierarchy. But the resource management mechanism in ICARUS currently does not allow division of a resource among multiple tasks.

Another important branch of work in this direction is that of *contention scheduling*. Norman and Shallice (1986) proposed a general theory that accounts for several phenomena in the control of action. Its core mechanism, contention scheduling, and the associated *supervisory attentional system*, influence the activation and inhibition of supporting and conflicting schemas, avoiding conflicts during performance. The system has vertical and horizontal 'threads,' in which factors like attention and motivational variables affect the activation of each schema, with activation alone determining schema selection. Whether two schemas support or conflict with each other depends on whether they use common processing structures. In contrast, the ICARUS architecture assumes each action has predefined resources requirements associated with it and handles the conflict resolution among tasks using required resources for actions involved in them. This approach prevents conflicting tasks from executing at the same time, while giving priority to more important goals.

More recent work has explained multi-task performance with explicit computer models. Cognitive architectures provide excellent frameworks for this purpose. One example is EPIC (Meyer & Kieras, 1997), which includes units for perceptual, motor, and cognitive processing along with declarative, procedural, and working memories. They have specified its perceptual and motor processors in detail, and they have

shown their framework is consistent with empirical data. EPIC's cognitive processor is based on a production system that allows parallel execution of actions by relying entirely on the conditions in rules and elements in working memory. This is similar in spirit to ICARUS' use of preconditions and goals to coordinate concurrent execution according to shared objects and logical constraints. But unlike EPIC's encoding of the executive process for multiple tasks as production rules, ICARUS handles multiple tasks dynamically using the coordination capability embedded within the architecture. This lets it to find opportunities for concurrent execution on the fly, although the current system does not learn from its experience in this regard.

The APEX architecture (Freed, 1998) is another example. It manages resources and resolves conflicts using explicit descriptions in its procedural knowledge. This framework represents procedures as a list of primitive actions, non-primitive procedures, a special termination marker, or any combination of them. The system assumes all elements of this list are concurrently executable unless they specify preconditions or priorities that forbid it. APEX allows each procedure to have its own profile that can include resource requirements. Essentially, ICARUS treats its subgoals in the skills this way, although we associate resource requirements with actions, not with skills. By computing the resource requirements for each skill from those of the actions involved, ICARUS does not need explicit coordination in skills. This not only reduces programming effort but also provides a more general way of encoding resources.

PRS (Georgeff & Ingrand, 1989; Myers, 1996) also has a reactive execution capability that supports multitasking. Like ICARUS and APEX, PRS specifies coordination information explicitly, but it uses special constructs in its goal structures, rather than in its procedures. The system controls termination and continuation of processes using these constructs. In contrast, Firby's RAP system (1994) manages multiple continuous processes that interact with each other in an implicit fashion

without any explicit constructs that interrupt and resume processes, although otherwise it is one of the closest frameworks to PRS. In this sense, ICARUS is closer to the RAP system, in that its skills do not have any such explicit descriptions.

More recently, Salvucci and Taatgen (2008) proposed a general theory of concurrent multitasking using autonomous threads that interact within the context of the ACT-R architecture (Anderson, 1993). They introduced a notion of threaded cognition that resembles threaded processes in operating systems. While a serial procedural resource coordinates threads of processes, other resources for perception and motor control execute them. In contrast, ICARUS' coordinative capability is embedded in the architecture, not taking the form of 'coordination skills.' Our framework assumes a central control mechanism that coordinates concurrent execution, rather than using explicit rules or skills for that purpose.

In summary, the ICARUS approach to concurrent execution combines two traditions: condition-based distributed control and a central processing unit that oversees management of resources. The shared objects and logical constraints are expressed as conditions (both preconditions and goal predicate) in its skills, but the resource-based ability is embedded within the architectural framework rather than implemented as another layer of rules. Unlike other research that focuses on modeling human performance with predefined structures, ICARUS supports dynamic conflict resolution among multiple tasks and their execution in the physical world.

4.9 Conclusions

In this chapter, we reported two extensions to the ICARUS architecture that support concurrent execution. The first involves the relaxation of the ordering constraint on subgoals in ICARUS' skills and a modification to the execution module to allow retrieval of multiple skill paths on a single cycle. Without any additional work, this

extension enables the architecture to find and coordinate concurrently executable skill paths subject to shared objects and logical constraints. The second extension adds support for another constraint that governs the coordination of concurrent execution. The new system manages resources by assigning them to executable skill paths and prevents the concurrent execution of paths that would use any resources that are already assigned to another path.

We demonstrated the advantages of the extensions using two scenarios in the urban driving domain. We found that they make the program simpler while improving driving behavior, especially at lower levels of abstraction. This is because the extensions for concurrent execution have the effect of increasing control frequency (or bandwidth) by enabling the system to generate more control inputs on each cycle. Having more control bandwidth decreases the need for fine-grained procedures, and this in turn reduces the number of disjunctive skills without sacrificing control performance. In the next chapter, we discuss another major extension to the ICARUS architecture for the reactive management of top-level goals.

Chapter 5

Reactive Goal Management

5.1 Motivation and Background

Goals play an important role in human cognition. Once established, they guide people's behavior by restricting the space of possible actions. On one hand, people seem to generate new goals in reaction to events in the world around them. Instead of thinking about all possible situations at any given moment, they react to current events and attempt to respond appropriately. People either explicitly or implicitly select a goal that suits the current events. On the other hand, people have ideas on what they want to do or what they should do, and these also give rise to goals. Such inner states of the mind seem to motivate a variety of goals, which in turn result in different actions.

How goals originate and how people manage them are closely related. In the literatures of artificial intelligence and control theory, however, studies on either topic are rare. Most researchers focus on what should happen once people have goals and they often overlook the few examples in the literature on these topics. But when designing a system to support general intelligence, it is crucial to include mechanisms that support generation and management of goals. Traditionally, however, cognitive

architectures like ICARUS (Langley & Choi, 2006b), Soar (Laird et al., 1986), and ACT-R (Anderson, 1993) assume that goals are given by the programmer. Nonetheless, the original ICARUS architecture has explicit descriptions of its goals and skills have close ties to the goals they aim to achieve, which provide an important foundation for new mechanisms that nominate and prioritize goals. These processes react to the current situation and provide improved ways to control the agent's behavior, modeling goal management in humans. Such mechanisms would not only grant the system more autonomy, but also lay a groundwork for an account of motivation.

Psychology differs from artificial intelligence or control theory in that it has given considerable attention to this topic. Typical studies examine the broader topic of emotion and its role in controlling behavior. For example, Sloman (1987, 2002) suggested that any system with priority in beliefs and actions will naturally have emotions. He argued that goals often conflict with each other, and systems must have a mechanism to resolve such conflicts. He also proposed that *motivators* can serve this purpose by generating and managing goals.

Researchers generally accept the idea that both the environment and internal states affect motivational processes (Miller et al., 1960; Norman & Shallice, 1986; Bargh, 1990; Moskowitz & Gesundheit, 2009). We can easily find evidence for this in our daily lives. For example, assume that a person sees someone else smiling at him. This might cause different reactions depending on his mood. Both the external fact (i.e., someone smiling at him) and the internal state (i.e., his mood) affect his selection of goals and actions (e.g., being nice by showing a friendly gesture or becoming wary and avoiding the person). This implies that the selection of goals in reaction to an event varies among individuals, and even within a single entity, depending on the agent's physiological, emotional, and other factors. As such, the combination of the environment and internal states is an important source of autonomy and adaptiveness, providing both motivations and restrictions for an agent's behavior. In this

regard, psychologists have suggested various models of interaction among emotion, motivation, and goal management. Most accounts resemble Gray and Braver's (2002) framework, in which an agent's environment triggers affective states, such as approach and withdrawal, which, in turn, introduce, maintain, or retract goals in its working memory. These goals eventually generate responses in the world, thereby forming the agent's behavior.

Regardless of being internal or external, ICARUS expresses the state of the 'world' as its internal representation of beliefs. These are sufficient to cover all aspects of the context that can trigger the goal selection. This means that we can program different 'rules' that relate beliefs to goals, and then goals to behaviors. We can describe these beliefs and goals in generalized forms, resulting in general rules that govern the agent's goal selection based on descriptions of the world. But the state not only triggers goals in agents; it can also cause them to deactivate or retract goals. Once a particular situation disappears or a different one surfaces, a person may stop what he has been doing, return to prior work, or start working on something completely different. For example, after an emergency maneuver to avoid an accident, a person might continue driving home or stop on the shoulder to calm himself before continuing. A complete model of goal management should have a mechanism that covers the retraction of goals as well as their nomination.

Apart from theoretical concerns, there are other reasons why we desire a mechanism for dynamic goal specification. In ICARUS, we can specify multiple top-level goals with fixed priorities. This lets the architecture pursue the most important goal during serial execution or consider more important goals earlier during concurrent execution, as described in the previous chapter. Although this produces reasonable behavior in many cases, it is unrealistic to assume that a fixed priority among a static set of top-level goals will suffice in all situations. The developer might be able to program ICARUS' goals in ways that they are mostly suitable for 'normal' circumstances,

but static goal specifications that do not deviate over time limit the system's ability to react to changing situations. Once the system starts running, there would be no way to alter the goal priorities, which would be fixed until it terminates.

In the following sections, we provide an overview of the extensions to ICARUS that resolve this problem, after which we describe each of them in greater detail. We also discuss the architectural implications of these extensions and demonstrate the new capabilities in four scenarios. We then discuss related work and make some closing remarks.

5.2 Overview of the Extended Architecture

We have extended ICARUS to accommodate new capabilities for goal management. The extended system features a general mechanism that can nominate, retract, and prioritize goals for embodied agents. It brings a substantial portion of motivational process to light, providing a way to dynamically generate agents' top-level goals, which users previously had to provide manually. Building on the fact that ICARUS' beliefs and intentions have long-term versions – concepts and skills – we have created a new long-term memory for generalized goals to parallel them. Of course, these generic goals must come from somewhere, but for now we will assume that they are given by the system developer.

Given a set of generalized goals, the architecture instantiates them based on the current environment and deposits the instantiated goals in short-term goal memory. The new ICARUS cannot have any goals other than ones derived from the generic goals stored in long-term goal memory. This assumption produces a more balanced architecture, in that it treats goals in the same way as beliefs and intentions. At the same time, it makes a connection to the psychology literature, which views goals as cognitive structures that one must retrieve from long-term memory (Bargh, 1990).

The extended system recognizes events that occur in the environment, which in turn leads it to nominate and prioritize corresponding goals to guide execution. Here the environment not only includes the external state of the world, but also the internal state of the agent. For this reason, we will refer to it as the *state* rather than as the *environment*. Potentially, anything in the state can affect the nominated goals and the priorities among them. Although the nomination of goals does not require any changes to ICARUS' inference process, the prioritization of goals requires a means to change priority values, and the extended system uses continuous matching as the source of such dynamic changes. The degree of match for state conditions modulates the default priority value of each goal, changing this value as a function of the state. If the situation changes and an event that triggered a goal disappears, the system retracts this goal and works on others that are currently relevant.

Figure 5.1 shows the operation of the extended architecture with goal management. The original ICARUS followed inference immediately with selection of skills to achieve top-level goals. The extended system, however, inserts processes for goal nomination and prioritization between inference and skill selection. Note that the architecture does not have a special mechanism for retracting previously nominated goals. Rather, on every cycle, the system invokes the goal nomination and prioritization afresh based on the current state, and any previously nominated goals that are no longer relevant in the state are effectively retracted.

The process happens in two steps, one for the nomination of goals and the other for the prioritization of these goals. The first step involves finding general goals associated with triggering conditions that hold in the current state. Each match of a goal trigger produces an instantiated goal, so the system can nominate multiple instantiations of a single long-term goal. The instantiated goals are those relevant to the agent's current situation, sorted according to their default priorities as specified in long-term memory.

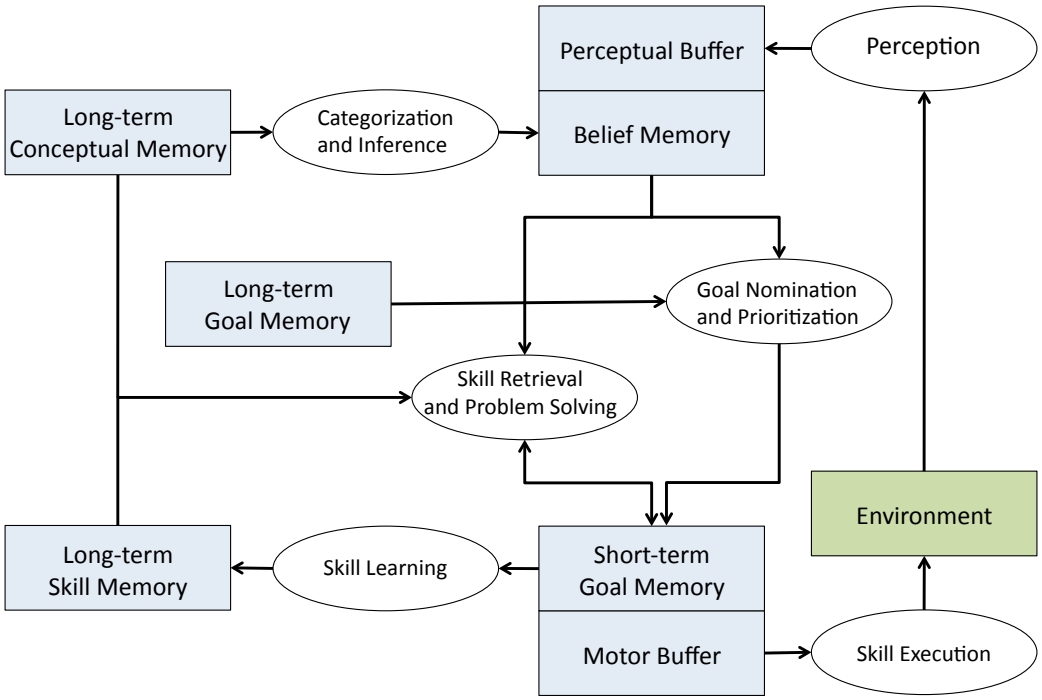

Figure 5.1: Operation of the extended ICARUS architecture with goal nomination and prioritization.

During the second step, the architecture prioritizes the nominated goals, again based on the current situation. The degree of match for goal triggers may also modulate these priorities. ICARUS can express a degree of match for concepts with a number between zero and one. A zero means that the concept instance is false, while a one means that it is completely true. It is important not to confuse this number with the probability of the instance being true, since there is no uncertainty involved. The degree of match describes, for example, the intensity of a red object's color, but not the probability that the object is red. ICARUS uses the degree of match for triggers associated with general goals – the *degree of goal relevance* – to modulate their default priorities. In short, the relevance affects the instantaneous importance of goals. In

the next two sections, we describe this two-step process in detail, starting with the new representations used for each step and then explaining how they operate.

5.3 Reactive Goal Nomination

As discussed in Chapter 3, ICARUS uses a separate goal memory to store the agent's current goals. The previous version stored a fixed set of top-level goals in this memory. In contrast, the extended architecture lets the top-level goals change dynamically during the course of execution. The new system includes a goal nomination module that generates top-level goals on each cognitive cycle. It bases these on the generalized goals stored in a new long-term memory and deposits instances in a short-term goal memory.

ICARUS' long-term goal memory can specify both domain-independent and domain-specific rules for the nomination of goals. These rules collectively constitute the ICARUS agent's motives. We believe that this arrangement simulates human behavior, which appears to instantiate general templates for goals in a given situation. For example, someone might have a general rule to avoid hitting pedestrians by slowing down or swerving around them. This rule might only apply when a pedestrian appears in the front of the vehicle, at which time the driver instantiates the goal to avoid that particular person.

On each cycle, ICARUS nominates a set of top-level goals. The system checks the current state to see if any matches exist for the conditions that trigger them. If it finds matches, it instantiates the corresponding goals accordingly. This means that the process of goal nomination reacts to the current state. However, this also has an important implication for goal retraction. Since the nomination occurs on every cycle, any existing goal that is no longer nominated in the subsequent cycle disappears from memory. We discuss details of these representations and processes below.

5.3.1 Representation and Memories

As noted, the extended ICARUS has a new long-term memory for goals. This memory stores generalized goals associated with corresponding *goal triggers*. These triggers are stated as conditions that are matched against belief memory to determine the goals' relevance to the current situation. As seen earlier, ICARUS encodes concepts as Boolean predicates, which implies that the goal triggers are either matched or unmatched, and, in turn, the associated goals are either nominated or not. For the purpose of goal nomination, this type of concept handling is sufficient, but we will see later that it can cause problems. Table 5.1 shows some sample trigger-goal pairs from ICARUS' long-term goal memory.

Table 5.1: Example of goal nomination triggers and their corresponding generalized goals stored in ICARUS' long-term goal memory.

```
((stopped-and-clear me ?ped)
 :nominate ((pedestrian-ahead me ?ped))

((okay-to-go me ?signal)
 :nominate ((signal-ahead me ?signal)
            (not-emergency me))

((clear me ?car)
 :nominate ((vehicle-ahead me ?car))

((cruising-in-lane me ?line1 ?line2)
 :nominate nil)
```

ICARUS orders these generalized, conditional goals according to their relative priorities, as specified by the programmer. This fixed priority among goals correspond to people's common sense about which ones are more important. For example, we know that people generally care more about saving their lives than avoiding robbery. In case of an accident, they might want to save a child before saving an adult. There are many examples of this sort, and many of us agree on such relative priorities. We consider the fixed ordering in long-term goal memory as a default priority structure for agents that are similar to those in humans.

Furthermore, due to the relationship between long-term and short-term goals, the priority structure in long-term goals carries over to their instantiated counterparts. For instance, if long-term memory encodes the priority that saving a person precedes saving a possession, then the instantiated goal of saving John has priority over that of saving a gold bar. Of course, there might be more than one instantiation for a general goal, in which case the ordering among these multiple instantiations is arbitrary. Table 5.2 shows examples of instantiated goals stored in the short-term memory that correspond to the general goals from Table 5.1. We will next describe how the system nominates and instantiates these goals.

Table 5.2: Nominated goals stored in ICARUS' short-term goal memory. This example has instantiated goals from three long-term goals in Table 5.1. Some long-term goals may have no instantiations, while others may have multiple instantiations.

```
[(goal :chaintype  skill
       :goaltype   primary
       :objective  (stopped-and-clear me p7102)
       :intention  ((stopped-and-clear me p7102) id:  1
                    bindings:  ((?ped .  p7102) (?self .  me))))]

[(goal :chaintype  skill
       :goaltype   primary
       :objective  (clear me c5871)
       :intention  ((clear me c5871) id:  17
                    bindings:  ((?obj .  c5871) (?self .  me))))]

[(goal :chaintype  skill
       :goaltype   primary
       :objective  (cruising-in-lane me ?line1 ?line2)
       :intention  ((cruising-in-lane me ?line1 ?line2) id:  19
                    bindings:  ((?self .  me))))]
```

5.3.2 The Nomination Process

On each cycle, when the extended architecture finds a match for any trigger stored in long-term goal memory, it instantiates the corresponding goal accordingly and stores the instantiated goal in short-term goal memory. When nomination is complete, the system has a series of top-level goals to guide agent behavior. Figure 5.2 depicts this process with an example from the urban driving domain.

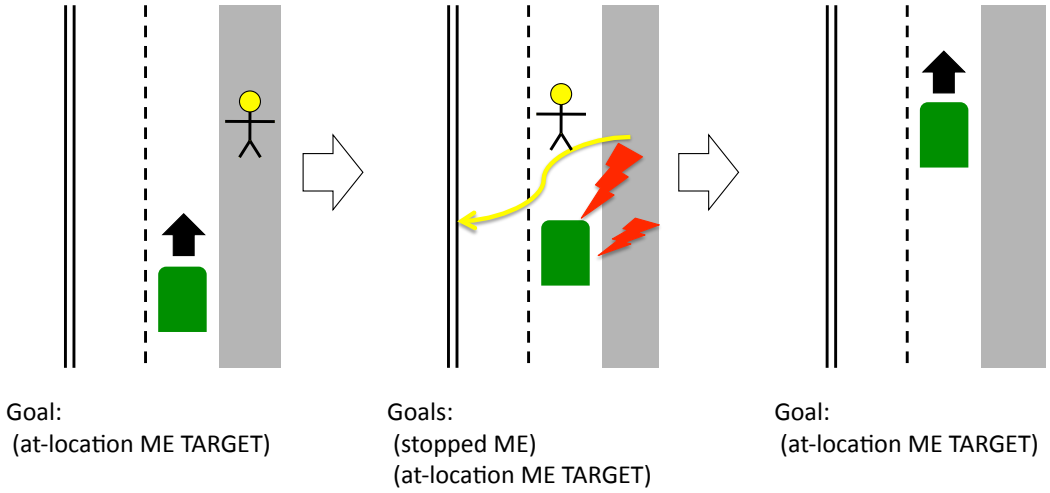

Figure 5.2: An example of goal nomination process in the urban driving domain.

The nomination process starts after the architecture infers its belief state based on its current perceptions of the environment. The system considers each <trigger, goal> pair stored in long-term goal memory and attempts to match the triggers against the current belief state. Whenever this attempt is successful, ICARUS instantiates the corresponding goal with the variable bindings it has found from the match.

The situation shown in Table 5.2 and Figure 5.2 involves a pedestrian, *p7102*, and a vehicle, *c5871*, that are just ahead of the agent's car. They cause the concepts *pedestrian-ahead* and *vehicle-ahead* to match and create the beliefs *(pedestrian-ahead me p7102)* and *(vehicle-ahead me c5871)*. Two goals, *stopped-and-clear* and *clear*, have these concept instances as their nomination conditions, respectively, so the goals are triggered. The third generic goal, *cruising-in-lane*, has a null nomination condition, causing it to be nominated in any situation.

Once the short-term goal memory has been populated in this manner, the system begins the execution process. As described earlier, ICARUS handles these top-level

goals differently depending on its execution mode. In the serial execution mode used by previous versions of the system, ICARUS works on the first unsatisfied top-level goal, finding a skill path that will achieve the goal. In the concurrent execution mode, however, it considers all the top-level goals specified in order of their priority, executing as many as the constraints allow. The nomination of goals on subsequent cycles is completely independent of the current set of goals. ICARUS performs this nomination from scratch on each cycle, and goals that are no longer relevant disappear from short-term memory.

5.4 Reactive Goal Prioritization

The extended ICARUS nominates goals that are relevant to the current situation and dynamically changes top-level goals on every cognitive cycle. Although it is a more complete model of cognition with practical advantages, the nomination process does not support dynamically changing priorities among the goals, in that it simply deposits them in the order of their instantiation. As described, instantiated goals in the short-term goal memory preserve the priorities of their generalized counterparts in long-term memory, sorted from more important goals to less important ones.

However, people also prioritize their goals based on the situation. As an example, we sometimes see in films a president who debates whether to save a small group of people in one area or to save millions by sacrificing the small group. People know that human life is very important, but situations of this sort would certainly cause a debate and a possible change of priorities. A more practical example might be a driver who bumps into a parked car to avoid hitting a child who abruptly runs into the street. People will normally not want to hit someone else's car, but the more urgent situation that involves a child can force the driver reprioritize goals.

To model such prioritization of goals in ICARUS, we needed some measure of the relative importance of each goal. As noted earlier, goal nomination uses the ordering of goals in long-term memory to initialize priorities. However, this is static and brittle, because no priority structure works in every possible situation. We should have more sophisticated ways to calculate the relative importance of goals depending on the situation. Hence, we introduce the notion of priority values associated with goals, which the system can change according to the situation. As we saw in the examples above, the severity of conditions affect the priority of the goal associated with them. People will not usually rummage through garbage for food, but then they might do so if they are extremely hungry. This illustrates the impact of state conditions on the priority of goals.

In ICARUS, we assign default values to long-term goals as a common sense priority and let the system modulate these in reaction to changes in the environment, based on the *degree* to which the goal triggers match in the current state. For this, we need to diverge from the Boolean inference assumed in the original ICARUS to produce intermediate values for each match. To this end, we introduce a continuous match of concepts that outputs a value between zero and one for each belief. This degree of match modulates the priority value associated with the corresponding goal, resulting in a revised priority for the current situation. We discuss this reactive prioritization of goals in more detail below.

5.4.1 Representation of Priorities

Common sense among people seems to give them rough ideas of good and bad. ICARUS simulates this with its default goal priorities, which the original architecture represents as an ordered list of its top-level goals. However, our experience in domains like urban driving suggests that the system should also change the priorities dynamically. For instance, consider a paramedic driving an ambulance. She would

observe the traffic rules just as regular drivers when there is no emergency or the situation is not critical. But when there is an emergency she might drive on the left side of the road to avoid traffic and run red lights to reach the destination quickly. The severity of the situation would affect this behavior, so the driver might make more emergency maneuvers when the patient's condition is critical than she would when it is not.

Situations of this sort require a change of priorities among goals, which benefits from having explicit values associated with goals rather than encoding them implicitly in terms of goal orderings. For this reason, we have assigned a default priority value to each long-term goal. However, we intend these only to show relative importance of the agent's goals, rather than to indicate their absolute values.

Table 5.3 shows an example of a general goal with an associated default priority. We have introduced a new field, *:priority*, in ICARUS' goal structures that stores a scalar value to represent the goal's relative priority. In this case, the first goal, *stopped-and-clear*, has the highest default value. The second goal, *okay-to-go*, has higher default value than the third, *clear*, which in turn has higher priority than the last goal, *cruising-in-lane*. As a result, the system recognizes the relative importance of avoiding pedestrians, obeying traffic signals, swerving around slower cars, and moving forward in this order.

5.4.2 The Prioritization Process

As shown in Section 5.3, goal nomination works with the existing Boolean inference about the environment. This is due to the discrete nature of the process, which decides whether or not the system should have certain goals subject to their relevance conditions. However, to support reactive prioritization, we need a metric that changes based on the situation, which in turn can influence the goal priorities. We focus on the fact that people's reaction changes according to the severity of conditions (e.g.,

Table 5.3: Examples of goal nomination triggers and their corresponding generalized goals stored in long-term goal memory, along with their respective priority values.

```
((stopped-and-clear me ?ped)
 :nominate ((pedestrian-ahead me ?ped))
 :priority 10)

((okay-to-go me ?signal)
 :nominate ((signal-ahead me ?signal)
            (not-emergency me))
 :priority 5)

((clear me ?car)
 :nominate ((vehicle-ahead me ?car))
 :priority 3)

((cruising-in-lane me ?line1 ?line2)
 :nominate nil)
 :priority 1)
```

the extent to which the firetruck driver might ignore the traffic rules depends on the seriousness of the emergency), and modulate the constant priority values based on the degree to which the goal triggers match in the current state. We can compute the *degree of match* for any concept instance, including goal triggers. This will be a scalar number between zero, which means that the instance does not hold in the state, and one, which means that the instance is true. For example, people might describe a color as 'light blue,' which would mean that it is blue, but its 'blueness' is smaller than usual. Using the degree of match, ICARUS might infer that the color is about 60% blue. It is important to distinguish the degree of match from the probability of match, which describes the likelihood that the color is blue.

Using the degree of match for goal-triggering concept instances, the system modulates the corresponding goal's priority value by multiplying the two numbers. Thus the revised priority of a goal for the current situation will fall between zero and the default priority of the goal. This modulation can change the ordering of nominated goals based on the agent's situation. For example, consider the ambulance example above with the goals shown in Table 5.3. For the sake of simplicity, assume that

there are no pedestrians or other cars in sight. Then the first and the third long-term goals are irrelevant, and we can focus on the remaining long-term goals, *okay-to-go* and *cruising-in-lane*. Note that one of the relevance conditions for the first goal is *(not-emergency me)*. The goal nomination system will always place priority to the first goal over the second one.

In contrast, reactive prioritization will let the system change this priority structure based on the degree of relevance for the *(not-emergency me)* condition, simulating human behavior in this example. When the situation is critical, the degree of match for this condition will be a number close to zero, while it will be closer to one when the situation is not so serious. The system will multiply this number by the default priority value of the corresponding goal, *okay-to-go*, and the resulting value will be close to zero in a critical situation and close to the default value in a casual situation. This causes the ordering of the goals, *okay-to-go* and *cruising-in-lane*, to be reversed in a serious emergency. In the next section, we describe the process of computing degrees of match, which we call *continuous concept matching*, in more detail. We will also provide examples of its operation.

5.4.3 Continuous Concept Matching

Agents that operate in some environment react to what is happening around them, and ICARUS computes a set of beliefs about its environment for this purpose. As seen in Chapter 3, the architecture infers its beliefs based on its concepts and the information it receives from the environment. In the original ICARUS, the inference process involves matching variables against objects in the world and their attribute values. There are also certain conditions on these variables. Inference leads to a Boolean value that indicates whether or not a concept instance is true. This *symbolic* pattern matching has served ICARUS well in a variety of domains, but we need a more continuous approach to conceptual inference for reactive goal prioritization.

We can find a source of continuity in concepts that incorporate numeric tests in their definitions. By applying a monotonic curve at the boundaries of these tests, we can get some continuous measures of how close we are to satisfying the tests. We will refer to the variables in these tests as *pivots* to emphasize that they serve as references for the computation of continuous degree of match for the concepts. This approach is very similar to Black's (1937) notion of *vagueness* and Zadeh's (1965) *fuzzy sets*, in which a function defines degree of membership as a number between zero and one. We can view the each numeric test in ICARUS' concepts as a fuzzy set defined on the pivot variable as a 'universal set,' with the 'core' of the fuzzy set as the range of the variable that satisfies the numeric test.

As an example, let us use the familiar case of a right turn in the urban driving domain. Depending on the angle of the turn, there will be an ideal range of the speed and the steering wheel angle of the car. For a typical 90 degree turn, we have found that a speed between 15 and 20 and a steering angle of 10 degrees produce reasonable behavior. Table 5.4 shows the original (a) and the new (b) versions of the concepts, *at-turning-speed* and *at-steering-angle-for-right-turn*, that describe the situation that the agent is ready for a right turn in terms of speed and steering wheel angle, respectively. The first two concepts are for the original architecture, while the rest use the extended representation for the continuous matching.

Assuming that we have an agent named 'me,' the traditional inference process would output *(at-turning-speed me)* when the agent's speed is between 15 and 20, but nowhere else. For the second concept, it would say *(at-steering-angle-for-right-turn me)* is true only when the steering wheel angle is 10. When the speed and the angle fall out of the regions, even by a miniscule amount, the system does not infer any of these instances, and ICARUS cannot tell how bad the situation is for a right turn. However, if it uses the continuous matching capability with a new field, *:pivot*,

Table 5.4: ICARUS concepts for right turns in the urban driving domain for (a) the original architecture and (b) the extended architecture.

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

((at-steering-angle-for-right-turn ?self)
 :percepts  ((self ?self steering ?angle))
 :tests     ((= ?angle 10)))
```

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

((at-steering-angle-for-right-turn ?self)
 :percepts  ((self ?self steering ?angle))
 :tests     ((= ?angle 10))
 :pivot     (?angle))
```

for each of these concepts, the degree of match will tell the system how close the current situation is to satisfying these conditions.

When inferring instances for a primitive concept, the extended ICARUS still matches patterns against percepts in the same way as before, but it applies a monotonic curve around the threshold values for the pivot variable in the numeric tests. For example, in Table 5.4, the third concept has a pivot variable, *?speed*, and ICARUS constructs a curve like those in the first row in Figure 5.3. For the last concept, the numeric test involves an equality of a pivot variable, *?angle*, to a single number, 10, and the system superimposes two monotonic curves around the number. This results in a bell shape like the ones in the second row in the figure. Although the piecewise linear curves in the right column are very simplistic, they serve our purpose of modeling a decreasing degree of match from the test regions. Ultimately, each test in primitive concepts should have different curves based on their sensitivity to changes in their pivot variables, but, for this thesis, we assume they are fixed across all the tests.

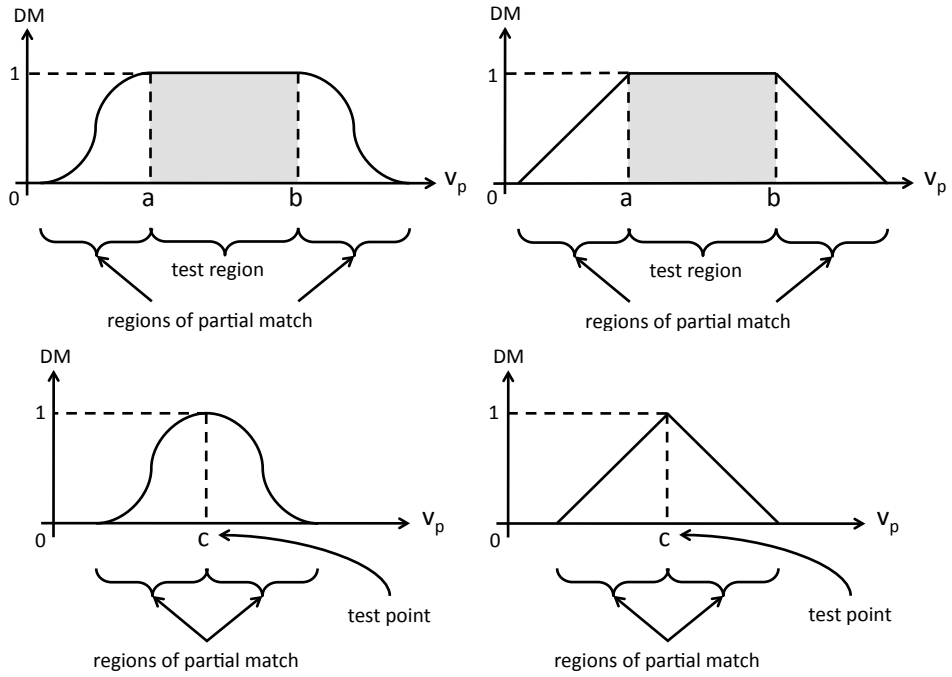

Figure 5.3: Monotonic functions applied to numeric tests against pivot variable v_p.

Now that we have a way to extract continuous values from numeric tests, we can propagate them through ICARUS' concept hierarchy. Some primitive concepts have more than one numeric test, so it is possible for a concept to have two or more pivot variables. This leads to the need for combining multiple degrees of match along these variables. Furthermore, higher-level concepts consist of other concepts, which may include multiple primitive concepts that have their own degrees of match, so we need a mechanism to combine two or more degrees of match. For this, we treat each degree of match as an axis in a multi-dimensional space and compute the combined degree of match as the vector sum specified by individual degrees of match on different Cartesian axes, as shown in Figure 5.4. Through this process, the continuous degree of match propagates upward through the concept hierarchy, enabling higher-level concepts to serve as triggering conditions for goal prioritization.

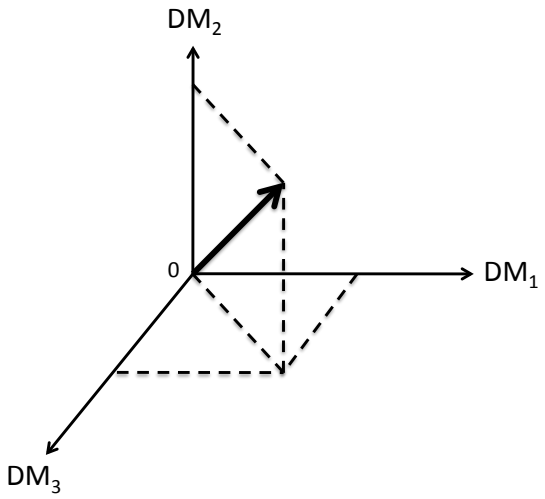

Figure 5.4: Combining multiple degrees of match in a multi-dimensional space.

5.5 Architectural Implications

Overall, the extended ICARUS architecture combines the components we have discussed so far, including:

- a new long-term memory for generalized goals;
- a modified goal structure that stores nomination conditions and a priority value for each goal;
- a new mechanism that handles the nomination of goals based on the conditions associated with them;
- a new process that allows continuous, partial matches based on the proximity of the test values in concepts; and
- a new mechanism that changes priorities among nominated goals using both the priority values and the degree of match.

These extensions give ICARUS the ability to nominate and prioritize its top-level goals based on the current belief state. This remedies several problems with the original system, as explained at the beginning of this chapter. The most important improvements include goal-oriented control at the top level of skill execution and added flexibility in the concurrent execution of skills.

As we demonstrate in the following section, these advantages by themselves comprise a significant advance from the previous versions of the system. However, the implications do not stop there, as they lay the foundation for increased psychological plausibility. Early in this chapter, we hinted that, in the psychology literature, the nomination and retraction of goals are often tied to a larger motivational system. The nomination mechanism should let us model the effect of internal drives and prioritization should let us simulate conflict resolution among candidate goals.

5.6 Demonstration of New Capabilities

The extensions we have described increase the power of the ICARUS architecture. In terms of the 'Newell test' for theories of cognition (Anderson & Lebiere, 2003), these extensions improve ICARUS on two fronts. First, the extended architecture exhibits more effective adaptive behavior, nominating and prioritizing goals in reaction to the current state. Second, it behaves more robustly in the face of unexpected situations, enabling agents that operate successfully in dynamic environments.

Testing these claims, however, does not lend itself to standard techniques, because cognitive capabilities like goal nomination and prioritization occur at a very high-level. We want to show performance improvements achieved in the extended system, but doing so using quantitative measures would be very difficult. Instead, we can demonstrate the qualitative behavior of the extended system and show that it improves functionality while aligning more closely with our intuitions about human

cognition. We might also compare the extended system's behavior to that of other architectures, but it is difficult to give credit to or blame a particular mechanism in these architectures based on quantitative results. In response, Cassimatis et al. (2008) have suggested that models of higher-order cognition should be evaluated on three dimensions: their ability compared to humans; the breadth of situations they cover; and the parsimony of their mechanisms. We believe the extended system improves significantly over the original architecture on all three accounts, but, in this thesis, we focus on issues of ability (i.e., functionality) and parsimony.

In the following sections, we first consider the goal nomination capability, examining the architecture's behavior in several scenarios with and without the extension. This comparison shows the advantages of this capability in terms of programability and human-like behavior. Often the version that lacks goal nomination cannot demonstrate the desired behavior at all, while the extended system can do so easily. After this, we demonstrate the benefits of goal prioritization in a similar fashion.

5.6.1 Nomination of Goals

Intuitively, thinking about more than a handful of goals at the same time does not seem human-like. In fact, people may be aware of many different things to do, but they do not think about them all the time. They focus on goals that are relevant to the current situation, not the ones unrelated to it. The extended Icarus models this by allowing many goals in its long-term memory, but holding only a few instantiated goals with matched nomination conditions in short-term memory. This not only prevents the short-term goal memory from becoming unnecessarily complicated, but also minimizes the risk of unwanted interactions between important goals and unimportant ones.

Scenario 1: Cruiser I

Imagine someone in a sports car cruising down the street. The driver notices a car slowing down, and she swerves around it by changing lanes. Later, some pedestrians suddenly run into the street. In response, she stops to avoid hitting them and continue onward down the road. Situations of this sort should be familiar to any urban driver.

In the previous version of ICARUS, we would produce such responses by giving the agent two goals, *(stopped-and-all-clear me)* and *(cruising-in-lane me ?line1 ?line2)*, in this order. The resulting system would give higher priority to the first goal, so it would focus attention on maintaining a safe distance from pedestrians before worrying about maintaining its cruising speed. However, this approach has a number of drawbacks. Not only the system has the first goal whether or not it is relevant, but the goal also does not mention any specific pedestrian, so it must pick a pedestrian dynamically within the skills for this goal. As a result, the agent can deal with only one pedestrian at a time. We might program things so that the closest pedestrian receives attention, but then the agent would have no way to consider any other pedestrians. Furthermore, as shown in Tables 5.5 and 5.6, the system requires a significant amount of conceptual knowledge to describe the particular goal state, *(stopped-and-all-clear me)*, as well as skill knowledge for achieving it. This affects the programability of the agent adversely.

In contrast, the goal nomination capability of the extended architecture lets us program three long-term goals, such as *(stopped-and-clear me ?ped)* with the nomination condition *(pedestrian-ahead me ?ped)*, *(clear me ?car)* with the nomination condition *(vehicle-ahead me ?car)*, and *(cruising-in-lane me ?line1 ?line2)* with a null condition. We can also assign priorities 10, 5, and 1 to these top-level goals, respectively. Tables 5.7 and 5.8 show ICARUS concepts and skills for the extended system that take this approach. One advantage of this strategy is that the agent will have only the relevant set of goals at any given moment. More important, the ICARUS agent can consider each goal instance separately. For example, if three pedestrians

Table 5.5: ICARUS concepts for the Cruiser I scenario using the original architecture. Concepts for basic driving are omitted for the sake of simplicity.

```
((stopped-and-all-clear ?self)
 :percepts  ((self ?self))
 :relations ((stopped ?self)
             (all-clear ?self)))

((all-clear ?self)
 :percepts  ((self ?self))
 :relations ((not (pedestrian-ahead ?self ?obj))))

((all-clear ?self)
 :percepts  ((self ?self))
 :relations ((not (vehicle-ahead ?self ?obj))))

((pedestrian-ahead ?self ?ped)
 :percepts  ((self ?self)
             (pedestrian ?ped dist ?dist angle ?angle alive ?alive))
 :tests     ((= ?alive 1)
             (< ?dist 20)
             (< ?angle 30)
             (> ?angle -30)))

 ((vehicle-ahead ?self ?car)
 :percepts  ((self ?self)
             (car ?car dist ?dist angle ?angle))
 :tests     ((< ?dist 50)
             (< ?angle 30)
             (> ?angle -30)))
```

Table 5.6: ICARUS skills for the Cruiser I scenario using the original architecture. Skills for basic driving are omitted for simplicity's sake.

```
((stopped-and-all-clear ?self)
 :percepts ((self ?self))
 :subgoals ((all-clear ?self)))

((all-clear ?self)
 :percepts ((self ?self))
 :start    ((pedestrian-ahead ?self ?car))
 :actions  ((*brake 1000)))

((all-clear ?self)
 :percepts ((self ?self))
 :start    ((vehicle-ahead ?self ?car)
            (in-leftmost-lane ?self ?line1 ?line2))
 :subgoals ((in-rightmost-lane ?self ?line3 ?line4)))

((all-clear ?self)
 :percepts ((self ?self))
 :start    ((vehicle-ahead ?self ?car)
            (in-rightmost-lane ?self ?line1 ?line2))
 :subgoals ((in-leftmost-lane ?self ?line3 ?line4)))
```

are jaywalking in front of the agent's car, then three instances of the generalized goal *(stopped-and-clear me ?ped)* are deposited into short-term goal memory, so the system can consider all of them in the order of their priorities. This lets it take an action for

the highest priority goal and continue to the subsequent ones if resources are available. Moreover, the system does not require a complicated goal concept. Instead of using multiple disjunctive definitions of a goal concept to cover complex cases, individual goals for different pedestrians are instantiated from a single generalized goal description, and deposited into the current goal memory.

Table 5.7: ICARUS concepts for the Cruiser I scenario using the extended architecture. Concepts for basic driving are omitted for the sake of simplicity.

```
((stopped-and-clear ?self ?obj)
 :percepts  ((self ?self))
 :relations ((stopped ?self)
             (clear ?self ?obj)))

((clear ?self ?obj)
 :percepts  ((self ?self)
             (pedestrian ?obj))
 :relations ((not (pedestrian-ahead ?self ?obj))))

((clear ?self ?obj)
 :percepts  ((self ?self)
             (car ?obj))
 :relations ((not (vehicle-ahead ?self ?obj))))

((pedestrian-ahead ?self ?ped)
 :percepts  ((self ?self)
             (pedestrian ?ped dist ?dist angle ?angle alive ?alive))
 :tests     ((= ?alive 1)
             (< ?dist 20)
             (< ?angle 30)
             (> ?angle -30)))

((vehicle-ahead ?self ?car)
 :percepts  ((self ?self)
             (car ?car dist ?dist angle ?angle)
             (lane-line ?line1 dist ?dist1 angle ?angle)
             (lane-line ?line2 dist ?dist2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((< ?cdist 70)
             (> ?cangle -90)
             (< ?cangle 90)
             (> (* ?cdist (sin (/ (* (- ?cangle ?angle) pi) 90))) ?dist1)
             (< (* ?cdist (sin (/ (* (- ?cangle ?angle) pi) 90))) ?dist2)))
```

Let us analyze a typical run with the extended system. The agent starts in the leftmost lane of a street segment. There are several other cars in that stretch of the street and the first one, *c6120*, is far ahead of the agent in the same lane. For the first 10 cycles, the agent has a single goal, *(cruising-in-lane me ?line1 ?line2)*, that is nominated and retained from the beginning. On cycle 11, as the agent gets closer to the car, *c6120*, it detects that the car is blocking its way when *(vehicle-ahead*

Table 5.8: ICARUS skills for the Cruiser I scenario using the extended architecture. Skills for basic driving are omitted for simplicity's sake.

```
((stopped-and-clear ?self ?obj)
 :percepts ((self ?self))
 :actions  ((*brake 1000)))

((clear ?self ?obj)
 :percepts ((self ?self))
 :start    ((in-leftmost-lane ?self ?line1 ?line2))
 :subgoals ((in-rightmost-lane ?self ?line3 ?line4)))

((clear ?self ?obj)
 :percepts ((self ?self))
 :start    ((in-rightmost-lane ?self ?line1 ?line2))
 :subgoals ((in-leftmost-lane ?self ?line3 ?line4)))
```

me c6120) becomes true. In response, the system nominates *(clear me c6120)* as a goal. There is no pedestrian in sight, so there are no other goals. On the next cycle, ICARUS retrieves a skill for the first goal with the same name, *clear*, which leads to the action *(*steer 35)*. While the agent is changing its lane to the right, it notices (on cycle 13) that its speed is below the predefined cruising speed and the second goal, *cruising-in-lane*, is unsatisfied. This leads the agent to execute *(*gas 20)* concurrently with *(*steer 35)* to adjust its speed. The system continues steering to the right while it performs the speed adjustments as needed until cycle 21, when it notices that it is in the target lane and starts aligning itself. By this time, the agent has successfully avoided the blocking vehicle and the concept instance *(vehicle-ahead me c6120)* is no longer true. As a result, the goal *(clear me c6120)* that was triggered by this concept instance disappears.

The agent cruises along the street until cycle 42, staying in its current lane. On cycle 43, however, it finds another car blocking its lane and nominates the corresponding goal *(clear me c6127)*. Upon seeing that the left lane is clear, the agent decides to change its lanes, since it is programmed to prefer passing on the left whenever possible. The system performs maneuvers similar to those above and changes to the first lane from the center. By cycle 58, it finishes the maneuver and starts cruising

down the street again. Finally, on cycle 105, it detects yet another car, nominates a goal to clear it, and changes lanes to the right, following the same strategy.

Scenario 2: Ambulance I

To make the task more complicated, let us think about driving an emergency vehicle, say an ambulance. The vehicle's driver would normally observe all the traffic signs and signals, driving safely. However, when an emergency strikes, he would turn on the siren and lights, and try to get to the destination as fast as he can, ignoring signals. We can model this kind of behavior using the previous version of the ICARUS architecture, by giving the system two goals, *(all-okay-to-go me)* and *(at-address me 100)*, in that order. Again, the system gives higher priority to the first goal, which checks if the agent can move forward subject to the situation (i.e., whether there is an emergency or not) and traffic signals. As in the previous scenario, we must program a complicated set of concepts to describe this goal condition properly, although the skills stay relatively simple. Tables 5.9 and 5.10 show the concepts and skills for the original architecture.

In the extended system, however, the program becomes simpler and more straightforward. We do not need a special goal that covers both emergencies and traffic signals. Instead, we can have a goal like *(okay-to-go ?self ?signal)* that forces the agent to observe signals except when there is an emergency. We now have two top-level goals, *(okay-to-go me ?signal)* with nomination conditions *(signal-ahead me ?signal)* and *(not (emergency ?self))*, and the same *(at-address me 100)* as in the original system. This seems more reasonable, since human drivers do not pursue a special goal with associated skills in an emergency. Rather, the emergency forces us to ignore some things while driving, which is exactly what the new system does.

Now, let us look at the extended system's behavior in more detail during a typical run in which the hospital is at the address 101 at the end of the current street. The

Table 5.9: ICARUS concepts for the Ambulance I scenario using the original architecture. Concepts for basic driving are omitted for simplicity.

```
((all-okay-to-go ?self)
 :percepts  ((self ?self))
 :relations ((not (emergency ?self))
             (okay-to-go ?self ?signal)))

((all-okay-to-go ?self)
 :percepts  ((self ?self))
 :relations ((not (emergency ?self))
             (not (signal-ahead ?self ?signal))))

((all-okay-to-go ?self)
 :percepts  ((self ?self))
 :relations ((emergency ?self)))

((emergency ?self)
 :percepts  ((self ?self status ?status))
 :tests     ((equal ?status 'emergency)))

((signal-ahead ?self ?signal)
 :percepts  ((self ?self)
             (signal ?signal color ?color
                              exitpath ?street angle ?angle))
 :relations ((on-street ?self ?street))
 :tests     ((< ?angle 0)))

((okay-to-go ?self ?signal)
 :percepts  ((self ?self)
             (signal ?signal color ?color))
 :relations ((signal-ahead ?self ?signal))
 :tests     ((equal ?color 'green)))
```

Table 5.10: ICARUS skills for the Ambulance I scenario using the original architecture. Skills for basic driving are omitted for simplicity.

```
((all-okay-to-go ?self)
 :percepts ((self ?self)
            (signal ?signal))
 :starts   ((signal-ahead ?self ?signal))
 :actions  ((*brake 25)))

((at-address ?self ?address)
 :percepts ((self ?self)
 :subgoals ((cruising-in-lane ?self ?line1 ?line1)))
```

simulation triggers an emergency situation with a probability of one percent at any point. The agent starts in the first lane on a stretch of *B street*. There are no obstacles in sight, so it starts accelerating to achieve its default goal, *(at-address me 101)*. By cycle 9, it reaches cruising speed. On cycle 10, however, it finds a car ahead, and *(clear me c61067)* is nominated as its higher priority goal. The agent decides to change its lane to the right and steers right until cycle 17 while maintaining its speed

Table 5.11: ICARUS concepts for the Ambulance I scenario using the extended architecture. Note that this is a significantly smaller subset of the concepts than in the original system.

```
((signal-ahead ?self ?signal)
 :percepts  ((self ?self)
             (signal ?signal color ?color exitpath ?street angle ?angle))
 :relations ((on-street ?self ?street))
 :tests     ((< ?angle 0)))

((emergency ?self)
 :percepts  ((self ?self status ?status))
 :tests     ((equal ?status 'emergency)))

((okay-to-go ?self ?signal)
 :percepts  ((self ?self)
             (signal ?signal color ?color exitpath ?street))
 :relations ((on-street ?self ?street))
 :tests     ((equal ?color 'green)))
```

Table 5.12: ICARUS skills for the Ambulance I scenario using the extended architecture. Skills for basic driving are omitted for simplicity.

```
((okay-to-go ?self ?signal)
 :percepts ((self ?self)
            (signal ?signal exitpath ?street))
 :starts   ((on-street ?self ?street))
 :actions  ((*brake 25)))

((at-address ?self ?address)
 :percepts ((self ?self)
 :subgoals ((cruising-in-lane ?self ?line1 ?line1)))
```

by occasionally pushing the gas pedal. On cycle 18, it notices that it is in the lane it desired and starts aligning itself in that lane. The agent cruises in the lane for a while, then finds another car, *c61074*, in front of it on cycle 29. This time it decides to avoid the car by changing its lane to the left, finishing the lane change by cycle 36. It cruises for a while again, but then it sees a traffic signal *c61042* in the red-yellow state on cycle 49. The agent starts braking to stop at the intersection, but its state changes to an emergency situation on cycle 51. At this point, its goal to observe the signal, expressed as *(okay-to-go me c61042)*, is retracted by the system. The agent starts to accelerate to get to the hospital, ignoring traffic signals. On cycle 79, it finds yet another car ahead and follows the same procedure as before to change lanes to the right, then cruises along the street until it reaches the hospital at address 101.

The two sets of programs shown so far, one for the original architecture and the other for the extended architecture, produce equivalent behaviors at the high level. However, they differ at lower levels in their basic driving maneuvers, and the extended system shows much smoother driving. Moreover, the goal nomination capability leads to a much simpler program that seems more intuitive and reasonable.

5.6.2 Prioritization of Goals

Complementing the goal nomination capability, goal prioritization serves as an added layer that provides the system with further adaptiveness and reactivity. People constantly change the priority of tasks according to their relative importance in a situation. This extension lets ICARUS react to the situation around it and adapt to changes that require prioritization of goals. Each generalized goal has a default value that represents its priority compared to other goals, and we believe this corresponds to the general consensus among people about relative importance under normal conditions. But extreme circumstances sometimes force people to reorder their goals, leading to decisions that are more appropriate for those situations.

Scenario 3: Cruiser II

To demonstrate that the extended system can simulate this kind of behavior, we revisit the first scenario, which assumed a driver cruising down a street in which other cars may block lanes and pedestrians sometimes jaywalk. Now imagine several jaywalkers of different ages and physical conditions who are crossing the street at the same time. Some may just stand frozen in the middle of the road from the fear that you might hit them, while others may move faster to avoid being hit. There may also be a parked car at the side of the road or a trucker unloading packages. In such complicated situations, the driver must consider many different factors; he cannot

focus on avoiding a single pedestrian. However, he may also be unable to avoid every possible collision; sometimes accidents are inevitable. In such cases, one must decide which alternative is the least serious outcome.

The previous version of the architecture does not support this kind of behavior, as it lacks any mechanism that allows dynamic comparison of alternatives. It has a fixed set of goals and a pool of skills that are relevant to them, and it simply cannot make a reactive decision about which goal is most important. With the goal prioritization, however, the architecture can reconsider the goal ordering given at the outset and change it dynamically. In such cases, the extended system can decide to bump into a parked car to avoid hitting a pedestrian or to hit an adult rather than hitting a group of children. Thus, it can minimize damage when it cannot avoid it altogether, as can happen in real life.[1]

For this scenario, let us take a simple but realistic example to test the extended architecture's behavior. Here, we disable the brakes in the agent's car and force it into a situation where an accident is inevitable. There will be three cars of different monetary values, and the agent must decide which car it should avoid hitting (or which car it should hit, for that matter). The agent has two long-term goals that are similar to those in Scenario 1, *(clear me ?car)*, with the nomination condition *(vehicle-ahead me ?car)*, and *(cruising-in-lane me ?line1 ?line2)*, with no nomination conditions. However, the concept definition of the nomination condition *vehicle-ahead*, shown in Table 5.13, differs from before in that it now has a pivot variable for continuous matching. There are no changes necessary to ICARUS' skills.

In a typical run, the agent starts at one end of a street segment with three lanes in each direction. The car is in the first lane initially and there are no obstacles immediately ahead. The agent accelerates to reach its cruising speed until cycle 10.

[1]For details on an accident in which a truck driver chose to hit a bus rather than a more expensive car when his brakes malfunctioned, see the news article at `http://www.danwei.org/front_page_of_the_day/one_reason_why_should_one_buy.php`.

Table 5.13: ICARUS concepts for the Cruiser II scenario using the extended architecture. The agent's skills and other concepts are the same as in Scenario I.

```
((vehicle-ahead ?self ?car)
 :percepts  ((self ?self)
             (car ?car dist ?dist angle ?angle value ?value)
             (lane-line ?line1 dist ?dist1 angle ?angle)
             (lane-line ?line2 dist ?dist2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((< ?value 50)
             (< ?cdist 70)
             (> ?cangle -90)
             (< ?cangle 90)
             (> (* ?cdist (sin (/ (* (- ?cangle ?angle) pi) 90))) ?dist1)
             (< (* ?cdist (sin (/ (* (- ?cangle ?angle) pi) 90))) ?dist2))
 :pivot     (?value))
```

At this point, it sees a car ahead of it and nominates the goal *(clear me c9252)*. To achieve this goal, it retrieves a skill for changing lanes to the right. On the next cycle, however, it sees another car in the right lane, and on the subsequent cycle it sees yet another car in the rightmost lane. By this time, the agent has three different instantiations of *clear*, one for each of the three cars. The ordering among these goals is determined by the monetary values of the three cars in sight. In this case, the car in front of the agent is the most expensive car, with the value 50. The other two cars are valued at 30 and 10, respectively. The default priority value for the goal *clear* is set to 5, but, due to the modulations from the three cars' values, the goal *(clear me c9252)* has priority 5 ($= 5 \times 1$), *(clear me c9259)* has priority 3 ($= 5 \times 3/5$), and *(clear me c9266)* has priority 1 ($= 5 \times 1/5$). Therefore, the first goal becomes the most important goal for the agent, so it maneuvers to avoid hitting the car *c9252* and collides instead with *c9259*. This scenario is less complicated than some real-world situations, but it demonstrates the benefits of ICARUS' new capability. The extended architecture not only can nominate relevant goals in the current situation; it can also change priorities dynamically among these goals, even ones that originate from the same long-term goal.

Scenario 4: Ambulance II

We can also expand the story in the second scenario above. When we walk down a street, we sometimes notice that an ambulance is driving normally, waiting for pedestrians to pass, observing the speed limit, and stopping for red lights. Yet other times we see an ambulance speeding by almost as though controlled by a reckless driver, blinking its lights and blaring its siren. We may guess that they are responding to problems of different severities that affect the drivers' behaviors.

Modeling this difference in the version of ICARUS with only nomination capability is very difficult, requiring the programmer to write long-term goals with different priorities for all possible cases. Even then, the system would require so many matches against these goal triggers that the nomination process would run very slowly. However, the extended system supports this behavior easily by using generalized goals and their continuous goal triggers. Table 5.14 shows a concept that we modified for this scenario to support continuous matching.

Table 5.14: An ICARUS concept modified for the Ambulance II scenario using the extended architecture. The agent's skills and other concepts are the same as those in Scenario 2.

```
((emergency ?self)
 :percepts ((self ?self status ?status level ?level))
 :tests ((equal ?status 'emergency)
         (= ?level 10))
 :pivot (?value))

((not-emergency ?self)
 :percepts  ((self ?self))
 :relations ((not (emergency ?self))))
```

To let the agent reach the hospital with the proper urgency, we modify one of the goals for Scenario 2, *(okay-to-go me ?signal)* with priority 2, to have the nomination conditions *(signal-ahead me ?signal)* and *(not-emergency me)*. This goal forces the agent to observe traffic signals. The change is due to the fact that our system currently

does not handle negations inside the nomination conditions, but the meaning of the goal is essentially the same as before. All other goals stay exactly the same.

Now let us consider how the system behaves during a typical run. As before, the agent starts out by accelerating itself to reach its cruising speed. On cycle 7, it finds a car blocking its path and starts steering to the right to avoid the car. With occasional accelerations to maintain speed, it continues steering to the right. On cycle 13, the agent notices that it is in the target lane and starts to cruise. It soon finds another car that it avoids in a similar manner, this time on the left, and finishes this maneuver by cycle 21. By cycle 30, the agent has cleared yet another car and arrived at an intersection. Because the traffic signal is red, it brakes to stop. During the wait, the emergency level changes to 8, which, in turn, changes the degree of match for the concept instance *(emergency me)* to 0.8. The negation of this instance, *(not-emergency me)*, therefore matches with degree 0.2. This is a nomination condition for one of the current goals, *(okay-to-go me c27224)*. Hence the system modulates the priority value of the goal to be $2 \times 0.2 = 0.4$. This causes the goal to be less important than the default, *(cruising-in-lane me ?line1 ?line2)*, which has priority one. Therefore, the system stops observing traffic signals and continues driving even through red lights. Later, on cycle 95, when it reaches the next intersection, the emergency level drops back to 3 and the modulated priority value for *(not-emergency me)* becomes 0.7. This again places the goal to obey traffic signals before the default goal of driving ahead, and the system starts observing signals again.

5.7 Related Work on Goal Management

We first approached the topic of goal management from an architectural perspective, but our work has been heavily influenced by related research in psychology, where one can find considerable work on motivation and goal selection. Many psychologists

have recognized that the internal and external states of an agent influence motivation (Miller et al., 1960; Norman & Shallice, 1986; Bargh, 1990; Moskowitz & Gesundheit, 2009). For example, Norman and Shallice's detailed model for control of behavior includes the environmental stimuli, motivational factors, and an attentional system to govern the activation of goals and the selection of action schemas. A *supervisory attentional system* activates goals under influence from motivational factors and the environmental stimuli control behavior. In this model, as Bargh (1990) summarizes, there is no explicit connection between environmental stimuli and the activation of goals. In contrast, ICARUS has an explicit link between the environment (represented as an agent's internal beliefs) and the selection of goals. Nomination rules stored in long-term goal memory controls this process by matching the relevance conditions associated with goals against current beliefs.

Simon (1967) proposed goal-terminating and interruption mechanisms that enable an essentially serial information processor to deal with unpredictable situations in real time. His termination mechanism stops further actions when a goal is achieved, which ICARUS incorporates as one of its basic features. But Simon's interruption mechanism assumes a motivational system similar in spirit to the one we have developed. He argues that an urgent need should interrupt ongoing actions and force an agent to focus its attention to the immediate problem. The extended ICARUS can achieve this functionality in either of two ways. First, if the interruption is commonly observed, a developer can program long-term goals with the interrupting situation as a negated nomination condition. Alternatively, one can have a separate goal to deal with such a situation and let the goal prioritization mechanism change the goal orders to focus on the urgent problem first. Of course, ICARUS' concurrent execution capability still lets it pursue the previous goals subject to coordination constraints.

More recently, Sloman (1987, 2002) proposed 'motivators' to resolve conflicts among goals. In contrast to ICARUS, which uses degree of relevance to prioritize

goals, he proposed three different measures – insistence, urgency, and intensity – that should affect how a system manages goals. ICARUS' notion of relevance does not map directly to these measures. Rather, the architecture computes the degree of conceptual match and the concepts can encode any of them. ICARUS provides a developer the ability to use any concepts for the relevance conditions on goals, and the developer must decide what type of measure to use.

Gray and Braver (2002) approached this topic in the broader context of emotion, which they claimed can prioritize conflicting alternatives and trade-offs. They further argued the need for integration of emotion and cognitive control, assuming this aids adaptation to the environment. They also outlined (pp. 298) a set of emotion-related processing stages that relate a situation to behavior through approach and withdrawal responses that lead in turn to corresponding goals. We should note that researchers in this field sometimes use the term *emotion* very loosely. However, regardless of the details, there is general agreement that a motivational system generates goals for agents and that the environment influences this process. Our work on ICARUS follows this trend, but it adapts the explicit notion of long-term goals with associated relevance conditions.

Jennings and Cohn (2006) have reviewed other related work in psychology that is similar in spirit to those above. However, there is surprisingly little work that directly addresses motivations and goals in the context of cognitive architectures. One such architecture is CLARION (Sun, 2007), which incorporates implicitly represented drives and explicitly specified goals. Two subsystems in the architecture interact to nominate goals. The *motivational subsystem* maintains an implicit, value-based network that relates the state of the world and the strength of an agent's drives. The *meta-cognitive subsystem* uses a multiple vote approach to determine the current goal. Each internal drive proposes multiple goals in the order of their assigned numeric

preference. The subsystem chooses the goal that receives most votes across all the drives and passes it to the execution module. In contrast, ICARUS does not divide this process into two submodules. The architecture maintains long-term goals as direct links from beliefs to goals. Unlike CLARION, which has a single goal that is active at a given moment, it also supports the nomination of multiple top-level goals.

Another architecture that incorporates an explicit goal nomination mechanism is Broersen et al.'s (2002) BOID. This is based on BDP logic (Thomason, 2000), which explains goals as a result of interactions between beliefs and desires, but the architecture includes obligations and intentions. An agent in this framework computes beliefs from observations of the world, while desires and obligations that are consistent with these beliefs trigger goals. The system treats previously generated goals as intentions that it uses to generate successive goals. Interactions among these four *conditional mental attitudes* may produce more than one candidate goal set, and the architecture supports different 'agent types' that resolve conflicts in different ways. For example, a realistic agent tends to choose a goal derived from beliefs over one from obligations, while a dogmatic agent does the opposite. Although ICARUS handles beliefs and intentions in a similar fashion, it includes neither obligations nor desires. However, we can program many of these aspects using ICARUS' goal management mechanism, since the architecture does not have any restrictions on the relevance conditions for its long-term goals. As long as we can represent obligatory or desire-driven rules as conditionalized long-term goals, ICARUS can achieve similar effects.

Some other architectures provide limited capabilities for goal management, with Gordon and Logan's GRUE (2005) being a good example. This closely resembles ICARUS in that it extends the teleoreactive framework (Nilsson, 1994) in a number of directions. Gordon and Logan include a goal arbitration mechanism that uses

resources, but GRUE cannot nominate or retract goals, which ICARUS achieves using its goal management mechanism. In contrast, Soar (Laird et al., 1986) has the ability to nominate its top-level operators as its action-like goals or intentions. But the architecture proposes a single intention at a time, removing both the need for prioritization and the potential advantages of interactions among goals.

This topic of goal management has also been discussed in the context of planning under nondeterministic conditions. A good example is Molineaux et al.'s ARTUE system (2010), which integrates a hierarchical task network planner (Nau et al., 2003) with four additional components: discrepency detection between expected and observed states; explanation of the unexpected events; goal generation; and goal prioritization. Upon detection of symbolic or numeric discrepencies, the system attempts to find hidden factors that influence the state and explain the detected discrepencies by abduction. Then ARTUE generates goals using background knowledge in the form of *principles*, which consist of a set of participants, a condition, a fixed intensity level, and a goal form. This representation is similar to ICARUS' long-term goals, which include relevance conditions, a default priority value, and a generalized goal. During the goal management process, the ARTUE system chooses a single goal with the highest intensity, while ICARUS modulates the default priority values of goals with the degree of relevance and reorders goals using the resulting values.

The field of robotics also includes related research. For example, Hanheide et al. (2010) propose a goal generation and management framework for the PECAS robot architecture (Hawes et al., 2009). Following Beaudoin and Sloman (1993), this framework encodes an agent's drives as goal generators, which react to the internal and external states. An *attention filter* blocks some of the generated goals based on their importance and urgency, thus protecting the higher-level management process. This latter process either activates or suspends the filtered goals. The authors does not

describe details of these processes, but they appear to use heuristics like information gain to determine the importance of candidate goals.

Motivation and self-control are also topics of interest in multi-agent systems literature. For instance, Luck and d'Inverno (1998) investigate the motivated generation of goals as a prerequisite for transfer or adoption of goals by multiple entities. The authors extend their previous work on agent autonomy by adding mechanisms for generation and adoption of goals. The system adopts a four-tiered hierarchical view of entities, objects, agents, and autonomous agents.

From a representational standpoint, the long-term goals in ICARUS resemble the constraints that Ohlsson and Rees (1991) use in their HS system. This encodes constraints as pairs of relevance and satisfaction conditions, which it uses to detect violated states and to revise rules to prevent further failures. Their notion of constraints is more general than our notion of conditionalized goals, in that they use constraints to evaluate or judge the state of the environment.

Although the extended ICARUS architecture does not yet offer a complete framework for general intelligence, the work reported in this chapter is an important step toward an architectural account of goal management. In addition to providing mechanisms for goal nomination and prioritization, it provides a unified approach to representing and interpreting goal knowledge in relation to other types of cognitive structures. Furthermore, it serves as a general mechanism that should let us investigate the effects of emotion, desires, and other factors that influence goals. As seen in the demonstrations above, the extended architecture can model the delicate interactions among instantiated goals that are needed for fine-grained judgements. ICARUS also has potential to model individual differences that occur in such prioritizations. In summary, our main contribution lies in proposing a unified framework that incorporates capabilities for goal nomination and prioritization into a general cognitive

architecture. This provides a solid foundation for future research, which we discuss in the next chapter.

5.8 Conclusions

In this chapter, we introduced an extension to ICARUS that supports goal management. In addition to its architectural significance, the approach has close connections to previous work in psychology. The extended framework nominates goals based on the current belief state, and it reacts to changes to pursue only relevant goals. Using two examples from the urban driving domain, we demonstrated that the extension allows simpler programs and better behavior than the original architecture.

Another important extension involves the ability to prioritize nominated goals. Through a mechanism that computes continuous degrees of match for nomination conditions, the system modulates the default priority values assigned to goals and uses the resulting values to reprioritize them. The extended framework can handle dynamic tasks like comparing multiple instances of the same goal or choosing the most important among a set of goals. Using two examples from the urban driving domain, we demonstrated a new ability that the original architecture lacks.

We argue that the contribution of these extensions goes beyond the relative advantages we have shown. They also provide a unified approach to representing and interpreting goal knowledge within the context of ICARUS' existing long-term and short-term cognitive structures. Our comparisons to related work also suggests that these extensions hold promise for modeling emotion, desires, obligations, and other factors that affect the behavior of intelligent agents.

Chapter 6

Future Work

Despite the novel contributions of this thesis, ICARUS is still not a complete framework for modeling human cognition. Specifically, there is ample room for further improvements in both coordinated execution and goal management. In this chapter, we describe these open problems and propose solutions to them.

6.1 Coordination under Temporal Constraints

We have already noted ICARUS' inability to handle explicit temporal constraints in Chapter 4. Ongoing research in the area of temporal inference enables the architecture to recognize time-related events, which provides a partial solution to temporal coordination in skills. Stracuzzi et al. (2009) propose an approach that associates concept instances with two time stamps. The *start time stamp* records the cycle when a concept instance initially becomes true, and the *end time stamp* records its last valid cycle. Using these time stamps, ICARUS can record the episodic history of predicates and reason about their temporal interactions. Furthermore, due to ICARUS' assumption that the heads of skills are the concept instances they achieve, the system can immediately include time-stamped predicates in its skills.

Despite this ability to represent and reason about temporal beliefs, ICARUS lacks a mechanism to process skills encoded with time-stamped goals and subgoals. The architecture can neither describe temporal constraints nor impose such constraints on time stamps. To support these abilities, ICARUS would require at least two extensions: (1) a new field in skills that specifies relations among time stamps associated with goals and subgoals, and (2) an extension to the execution mechanism that enforces these temporal relations to ensure coordinated behavior.

The first extension could be implemented by adding a new field like *:constraints* to ICARUS' skill structures. This field would include arithmetic relations to describe temporal constraints among time stamps that are mentioned in the head and *:subgoals* field of the skill. The second extension would require extensive modifications to the execution mechanism. When considering a skill for execution, ICARUS currently prioritizes earlier subgoals in the skill over later ones, but the revised version should instead sort the subgoals according to the new temporal constraints. For example, consider a skill with two subgoals, A with time stamps *?t1* and *?t2* and B with time stamps *?t3* and *?t4*. If we impose a temporal constraint, $?t1 > ?t4$, then the system should achieve the second subgoal, B, before continuing to the first one, A.

Once the system determines the temporal layout of subgoals, it can start execution for the ones that are currently relevant. However, note that these extensions do not enable ICARUS to ensure goal achievement at a particular point in time. Since the architecture's skills neither guarantee achievement of their goals nor specify when they will complete, the system cannot control when the execution for a certain subgoal will finish. This means that using absolute time for the end time stamps would not make sense in this framework.

6.2 Psychological Modeling of Cognitive Resources

If we consider the current mechanism for resource constrained execution in ICARUS, we find more opportunities for research. The psychology literature frequently refers to perceptual or cognitive bottlenecks, but ICARUS mostly ignores these limitations. The architecture assumes that all beliefs are inferred in parallel, and that the inference process completes before execution begins. These are not only psychologically implausible, but the strict ordering of inference and execution also prevents the system from simulating some well-established bottlenecks.

As seen in Chapter 4, ICARUS currently supports constrained parallelism in execution. Shared objects, logical, and resource constraints apply only during this process. But there is no reason that we cannot extend the parallelism to the architecture's internal processes and introduce a similar capacity for coordination. We can extend the notion of a resource to the architectural level and specify resources such as the central cognitive processor, the perception modules, and the motor controller. Then, using a mechanism similar to resource-constrained execution, we can simulate different cognitive bottlenecks.

For this extension, we would modify ICARUS so that different goals can call upon their own processes. Then we can assign resources to these internal processes and let the system manage them. If we assign the resources "perception" and "inference" to the processes that separately populate the perceptual buffer and infer beliefs based on the contents, then tasks requiring the same resources cannot execute concurrently, thereby simulating a perception or inference bottleneck. We can simulate different bottlenecks in a similar fashion by assigning other resources to processes.

However, this approach is not sufficient for modeling divisible resources. It would not have the ability to share a resource among multiple tasks and subsequently would lack a mechanism that simulates impeded processing due to shared resources. We

believe this capacity is more advanced than the one described above, making it a topic for longer-term research.

6.3 Learning from Coordinated Execution

Another drawback of the current architecture is that it fails to learn from its experience with coordinated execution. For example, once ICARUS finds concurrently executable skill paths according to their resource requirements, it is reasonable to store them in order to save the time and effort in the future. We have implemented an initial version of a learning mechanism that caches the multiple skill paths that ICARUS has found, but, without generalization, the cached knowledge would be nearly useless unless the system encounters a situation that is exactly the same as the one in which it was learned.

In response, we explored generalizing the skills involved in coordinated execution. During this process, we noticed that many concurrently executable skills are independent of each other, with no common bindings. This is because the skills have different top-level goals, which usually describe orthogonal aspects of the goal condition. Taatgen (2005) describes a skill acquisition mechanism in ACT-R that learns from concurrent execution. This type of learning is useful when the rule interpreter can handle only a single rule at a time, and therefore requires a compiled rule to coordinate execution for multiple tasks. The coordinated execution in ICARUS provides a powerful means to consider multiple skills on a single cycle, effectively eliminating the need for such compilation. Instead, the architecture can execute concurrent paths dynamically subject to different constraints.

However, if we introduce limits on cognitive resources, as discussed in the previous section, then ICARUS would also benefit from compilation of concurrent tasks to reduce the use of cognitive processing resources. For this, we can borrow ideas from

compilation of production rules in ACT-R. If we add a new field in ICARUS' skill structure to specify concurrent subgoals and store multiple subgoals the architecture finds during the coordination process, the architecture will be able to use such skills to reduce its use of cognitive resources. Combined with a method for learning a new skill that is a composition of skills that ICARUS executes together, this should enable more rapid processing on concurrent tasks.

6.4 Concept-dependent Partial Matching

Another important aspect of the current system is the continuous matching of concepts. ICARUS applies monotonic curves at the boundary of numeric tests within concepts to compute the degree of match. In this thesis, we used the same curve across all concepts. Nevertheless, we can imagine situations where a single, system-wide curve is inappropriate and where different concepts require distinct slopes at the edges of their numerical regions.

For example, we might have two concepts, *aligned-with-lane* and *heading-north*, in the urban driving domain. Since the accuracy we need to check whether we are driving straight in a lane is probably higher than that needed when measuring the heading, we want to match the former more strictly than the latter, with less tolerance permitted. In such cases, we should apply two different curves to guide the continuous matching of concepts, which would impose two distinct levels of tolerance.

Supporting the use of multiple curves does not require major changes to the architecture, but we need more than a selection of curves. For this feature to be useful, ICARUS should update the shape of its curves based on experience. However, the learning process must be closely related to the outcome of a goal achievement. This requirement makes the extension more complicated than simple value learning.

To address this problem, ICARUS will need to adjust the shape of the curves based on the results of execution – a process that is influenced by several factors. We can incorporate ideas from work on fuzzy logic controllers for a solution. For example, Berenji (1992) proposed a learning mechanism that updates monotonic membership functions of fuzzy sets using a method similar to temporal difference learning. ICARUS' monotonic functions for partial matching maps directly onto these membership functions for fuzzy sets, so this approach to learning should be applicable.

6.5 Uncertainty of Beliefs

We also recognize the need for further research on uncertainty in ICARUS. Here we plan to focus on the issue of inaccurate percepts caused by noisy sensor outputs. The partial matching capability introduced in Chapter 5 provides one response to this problem, in that the mechanism lets ICARUS adapt to sensor errors by inferring beliefs outside the boundaries of concepts' numeric tests. For example, consider a concept that checks whether the distance to a vehicle is less than five. If ICARUS perceives a vehicle at distance six when, in fact, the vehicle is at five, partial matching would still return a high degree of match for the concept instance, despite the fact that the numeric test is not satisfied. This feature lets the system tolerate inaccurate sensor values, much as fuzzy logic controllers.

However, partial matching cannot handle missing percepts due to sensor limitations, such as limited visual fields and occlusion. One response to such issues would involve building a probabilistic version of the ICARUS inference process that lets the system recover from missing sensor outputs. There is a long history of work that addresses this type of challenge, with Markov logic networks (Richardson & Domingos, 2006) being a recent, popular example. These networks combine first-order logic and probabilistic models into a single representation. Each clause in the knowledge

base constrains the truth values of the variables and has an associated weight that reflects the strength of the constraint. Given a set of observations, the system carries out a hill-climbing search through a space of possible worlds, returning the one it considers most probable. This world will contain beliefs that are consistent with the observations, given the available rules, including ones that cannot be derived deductively.

ICARUS' inference mechanism is quite modular, and replacing it with a Markov logic interpreter would be a straightforward task. Once integrated, it would address the problems of missing percepts. The increased stability of perception and inference would let ICARUS agents behave more reliably and realistically. For example, consider an agent driving its car down the street. It might see only the front half of a car coming around a corner. Although this partial sensing does not give the agent a complete percept of the car, a Markov logic interpreter would suggest the car is very probable, and the agent can make proper decisions in response.

6.6 Learning Goal Priority Values

The final target for further research concerns the current mechanism for goal management. As explained in Chapter 5, the extended architecture assumes constants for the default priority values of long-term goals. Although we argued that these constants represent most people's notion of ethics, this does not mean that they cannot change. It seems plausible that values should drift slowly over time, as a person might change his ethical position over the years. We believe that experience can motivate this change, especially the experience of failures.

One approach to updating default priorities would introduce utility values associated with concepts and use them as default priorities for goals. Previous work by Asgharbeygi et al. (2006) has explored learning such utility values within a variant of

ICARUS in the context of multi-player games with delayed rewards. These domains yield only a small number of conceptual predicates that have innate non-zero values. The system uses a version of temporal difference learning over its relational concept structures to propagate reward and update expected value functions for other predicates that may lead to concepts with innate values. Such value functions represent the predicates' importance in the domain, letting them serve as reasonable default values for goal prioritization.

However, as we discussed in Section 5.7, utility to the agent might not be the only factor for deciding goal priorities. Sloman (1987) proposed insistence and urgency as factors that influence goal priorities, and we can incorporate this idea to ICARUS. For this, it will be beneficial to have a separate priority learning mechanism. One approach would use a simple reward mechanism in which repeated failure to achieve a goal decreases the insistence value associated with that goal, and in which failure caused by delayed execution increases the urgency value associated with it.

Chapter 7

Conclusions

In this thesis, we introduced two distinct but related high-level capabilities into ICARUS, a reactive cognitive architecture. The framework shares basic features with other architectures of its kind, like Soar (Laird et al., 1986) and ACT-R (Anderson, 1993). It makes commitments to representations, memories, and processes that underlie cognition. ICARUS distinguishes long-term knowledge and short-term structures, it has separate memories for conceptual and skill knowledge, and it supports bottom-up inference of beliefs and top-down execution of skills. The architecture uses means-ends analysis to solve problems that it cannot immediately achieve through execution, and it learns new skills from successful traces of such problem solving.

However, the standard version of ICARUS lacks some key abilities that appear necessary to an account of general intelligence. One such facility involves coordinating concurrent execution of skills under constraints. For this, an architecture must represent and process constraints that govern concurrent execution in relation to shared objects, logical relations, and resource requirements. To support this capability, we developed a mechanism for multiple skill retrieval and extended ICARUS to manage resources to coordinate concurrent skill execution.

The extended system relaxes the earlier architecture's strict ordering on subgoals and allows their concurrent execution subject to the three types of constraints. The original architecture could handle the first two types within its existing skill structures, but resource constraints require additional machinery that assigns and tracks the resource requirements of multiple skill paths considered for concurrent execution. During the skill retrieval process, the previous ICARUS retrieves the first executable skill path it finds, but the extended architecture uses predefined resource requirements for each action and assigns available resources to the executable skill path found. Then it continues its search for more executable paths, retrieving candidates that do not require any resources already assigned. In this way, the architecture finds all concurrently executable skill paths the resource constraints allow and executes them in the environment.

Evaluating integrated frameworks like ICARUS poses problems for traditional quantitative performance comparisons. The architecture's behavior arises from interactions among its modules, and it is hard to isolate the incremental improvements due to each extension. Furthermore, the new capability for coordinated execution produces qualitative improvements, like greater ease of programming, that are not easily measured. Instead, following the tradition outlined in Cassimatis et al. (2008), we have demonstrated the superior ability and parsimony of the extended architecture using examples in an urban driving domain.

Another high-level capability we developed is reactive goal management. We originally introduced this feature for practical reasons, but the extension also makes close connections to the psychological literature. With this new mechanism, the agent's inferences about the environment trigger it to nominate or retract top-level goals, as well as to prioritize them. The ability to manage goals in this way lets the architecture react to its environment very flexibly, guiding the execution of skills in an adaptive manner.

For the nomination and retraction of goals, ICARUS uses a constraint-like representation that consists of relevance conditions and generalized goals. The architecture instantiates the long-term structures upon encountering triggering beliefs that match the corresponding relevance conditions. The mechanism not only generates goals in a way that is psychologically plausible, but also maintains architectural consistency with other ICARUS commitments. The distinction between long-term and short-term goals resembles that between concepts and beliefs, as well as that between skills and intentions. This consistent treatment provides a more unified account of knowledge in the architecture.

Once it has nominated top-level goals in this reactive manner, the extended ICARUS prioritizes its nominated goals according to the degree of match for their corresponding relevance conditions. The process assumes some continuous measure for the triggering beliefs of each long-term goal, and we developed a partial matching capability that uses numeric tests associated with concepts as the source of continuity. Instead of treating a concept instance as false when a variable falls out of the region for corresponding numeric tests, the system applies a monotonic curve that allows partial matches in regions near to the numeric boundaries. The system uses this continuous degree of match to modulate the priority of each short-term goal. In this manner, ICARUS prioritizes its nominated top-level goals in reaction to changes in the environment on every cycle. We have demonstrated this capability, as before, on examples from the driving domain.

Our demonstrations of the extended architecture on urban driving reveal that its new abilities enable behaviors that were formerly intractable. The extensions also fit into the architectural framework in a well-organized fashion that provides a parsimonious account of human capabilities. However, the extensions also suggest directions for additional work. In the context of coordinated execution, we plan to incorporate temporal constraints to provide a more complete model of coordination.

From a psychological perspective, we should attempt to model cognitive bottlenecks with resource-based methods similar to those in our current system. We believe this adaptation will be a major modification in the architecture, although it should be straightforward conceptually, if we establish closer connections to the psychology literature. We also see benefits of learning from traces of coordinated execution, which seems to require deeper analysis of concurrency itself.

Our work on reactive goal management also suggests important avenues for future research. We should make the system more flexible by making the shape of monotonic curves used in continuous matching be concept dependent and enabling it to handle uncertainty in beliefs. We should also extend the architecture to learn goal priority values based on its experience.

In summary, our research started with ICARUS, an existing cognitive architecture, and extended it in two major directions. The augmented system coordinates concurrent execution of multiple skills subject to different constraints. The new ICARUS also supports reactive goal management at the top level, letting the architecture nominate, retract, and prioritize its goals. Examples in urban driving demonstrate the new system's superior abilities and improved parsimony over the original framework.

Appendix A

Basic Program for Urban Driving

```
concepts:

((stopped ?self)
 :percepts  ((self ?self speed ?speed))
 :tests     ((> ?speed -1)
             (< ?speed 1)))

((moving ?self)
 :percepts  ((self ?self))
 :relations ((not (stopped ?self))))

((steering-straight ?self)
 :percepts  ((self ?self wheel-angle ?angle))
 :tests     ((< ?angle 0.1)
             (> ?angle -0.1)))

((steering-to-left ?self)
 :percepts  ((self ?self wheel-angle ?angle))
 :tests     ((< ?angle -30)))

((steering-to-right ?self)
 :percepts  ((self ?self wheel-angle ?angle))
 :tests     ((> ?angle 30)))

((slow-for-cruise ?self)
 :percepts  ((self ?self speed ?speed))
 :tests     ((< ?speed 10)))

((at-cruising-speed ?self)
 :percepts  ((self ?self speed ?speed))
 :tests     ((>= ?speed 10)
             (<= ?speed 15)))

((fast-for-cruise ?self)
 :percepts  ((self ?self speed ?speed))
 :tests     ((> ?speed 15)))

((slow-for-turns ?self)
 :percepts  ((self ?self speed ?speed))
 :tests     ((< ?speed 15)))
```

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

((fast-for-turns ?self)
 :percepts  ((self ?self speed ?speed))
 :tests     ((> ?speed 20)))

((line-on-left ?self ?line)
 :percepts  ((self ?self segment ?sg)
             (lane-line ?line segment ?sg dist ?dist))
 :tests     ((< ?dist 0)))

((line-on-right ?self ?line)
 :percepts  ((self ?self segment ?sg)
             (lane-line ?line segment ?sg dist ?dist))
 :tests     ((>= ?dist 0)))

((right-of ?right ?left)
 :percepts  ((lane-line ?right dist ?dist1 segment ?segment)
             (lane-line ?left dist ?dist2 segment ?segment))
 :tests     ((> ?dist1 ?dist2)))

((in-between ?between ?left ?right)
 :percepts  ((lane-line ?between dist ?dist)
             (lane-line ?left dist ?dist1)
             (lane-line ?right dist ?dist2))
 :relations ((right-of ?between ?left)
             (right-of ?right ?between)))

((lane ?left ?right)
 :percepts  ((lane-line ?right)
             (lane-line ?left))
 :relations ((right-of ?right ?left)
             (not (in-between ?any ?left ?right))))

((in-lane ?car ?line1 ?line2)
 :percepts  ((self ?car segment ?sg)
             (lane-line ?line1 segment ?sg)
             (lane-line ?line2 segment ?sg))
 :relations ((lane ?line1 ?line2)
             (line-on-left ?self ?line1)
             (line-on-right ?self ?line2)))

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

((sidewalk ?line)
 :percepts  ((lane-line ?line color SIDEWALK)))

((lane-on-left ?self ?line1 ?line2)
 :percepts  ((self ?self segment ?sg)
             (lane-line ?yellow segment ?sg)
             (lane-line ?line1 segment ?sg)
             (lane-line ?line2 segment ?sg))
 :relations ((yellow-line ?yellow)
             (lane ?line1 ?line2)
             (right-of ?line2 ?yellow)
             (line-on-left ?self ?line2)))

((lane-on-left ?self ?line1 ?line2)
 :percepts  ((self ?self segment ?sg)
             (lane-line ?line1 segment ?sg)
             (lane-line ?line2 segment ?sg))
 :relations ((yellow-line ?line1)
             (lane ?line1 ?line2)
             (line-on-left ?self ?line2)))
```

```
((lane-on-right ?self ?line1 ?line2)
 :percepts  ((self ?self segment ?sg)
             (lane-line ?sidewalk segment ?sg)
             (lane-line ?line1 segment ?sg)
             (lane-line ?line2 segment ?sg))
 :relations ((sidewalk ?sidewalk)
             (lane ?line1 ?line2)
             (right-of ?sidewalk ?line1)
             (line-on-right ?self ?line1)))

((lane-on-right ?self ?line1 ?line2)
 :percepts  ((self ?self segment ?sg)
             (lane-line ?line1 segment ?sg)
             (lane-line ?line2 segment ?sg))
 :relations ((sidewalk ?line2)
             (lane ?line1 ?line2)
             (line-on-right ?self ?line1)))

((closest-lane-on-left ?self ?line0 ?line1)
 :relations ((in-lane ?self ?line1 ?line2)
             (lane-on-left ?self ?line0 ?line1)))

((closest-lane-on-right ?self ?line2 ?line3)
 :relations ((in-lane ?self ?line1 ?line2)
             (lane-on-right ?self ?line2 ?line3)))

((leftmost-lane ?line1 ?line2)
 :percepts  ((lane-line ?line1)
             (lane-line ?line2))
 :relations ((yellow-line ?line1)
             (lane ?line1 ?line2)))

((rightmost-lane ?line1 ?line2)
 :percepts  ((lane-line ?line1)
             (lane-line ?line2))
 :relations ((sidewalk ?line2)
             (lane ?line1 ?line2)))

((in-leftmost-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2)
             (leftmost-lane ?line1 ?line2)))

((in-rightmost-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2)
             (rightmost-lane ?line1 ?line2)))

((aligned-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 angle ?angle)
             (lane-line ?line2))
 :relations ((steering-straight ?self)
             (in-lane ?self ?line1 ?line2))
 :tests     ((>= ?angle -3)
             (<= ?angle 3)))
```

```
((segment-on-left ?self ?sg ?target)
 :percepts  ((self ?self)
             (street ?target)
             (segment ?sg street ?target angle ?angle))
 :relations ((not (in-segment ?self ?sg))
             (not (on-street ?self ?target)))
 :tests     ((< ?angle -40)
             (> ?angle -90)))

((segment-on-right ?self ?sg ?target)
 :percepts  ((self ?self)
             (street ?target)
             (segment ?sg street ?target angle ?angle))
 :relations ((not (in-segment ?self ?sg))
             (not (on-street ?self ?target)))
 :tests     ((> ?angle 40)
             (< ?angle 90)))

((almost-aligned-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 angle ?angle)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((>= ?angle -3)
             (<= ?angle 3)))

((misaligned-to-left-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 angle ?angle)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((> ?angle 3)
             (<= ?angle 6)))

((misaligned-to-right-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 angle ?angle)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((< ?angle -3)
             (>= ?angle -6)))

((too-misaligned-to-left-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 angle ?angle)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((> ?angle 6)))

((too-misaligned-to-right-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 angle ?angle)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((< ?angle -6)))

((centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 dist ?dist)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((>= ?dist -6)
             (<= ?dist -4)))
```

```
((slightly-uncentered-to-left-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 dist ?dist)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((> ?dist -4)
             (< ?dist -2)))

((slightly-uncentered-to-right-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 dist ?dist)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((< ?dist -6)
             (> ?dist -8)))

((uncentered-to-left-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 dist ?dist)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((> ?dist -2)))

((uncentered-to-right-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self)
             (lane-line ?line1 dist ?dist)
             (lane-line ?line2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((< ?dist -8)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :relations ((aligned-in-lane ?self ?line1 ?line2)
             (centered-in-lane ?self ?line1 ?line2)))

((cruising-in-lane ?self ?line1 ?line2)
 :relations ((at-cruising-speed ?self)
             (aligned-and-centered-in-lane ?self ?line1 ?line2)))

((on-wrong-side-of-road ?self)
 :percepts  ((self ?self)
             (lane-line ?line))
 :relations ((yellow-line ?line)
             (line-on-right ?self ?line)))

((on-right-side-of-road ?self)
 :percepts  ((self ?self)
             (lane-line ?line))
 :relations ((yellow-line ?line)
             (line-on-left ?self ?line)))

((in-segment ?self ?sg)
 :percepts  ((self ?self segment ?sg)
             (segment ?sg dist ?dist))
 :tests     ((> ?dist 0)))

((in-intersection ?self ?int)
 :percepts  ((self ?self segment ?int)
             (intersection ?int)))

((in-intersection ?self ?int)
 :percepts  ((self ?self segment ?sg)
             (segment ?sg dist ?dist)
             (intersection ?int dist ?idist))
 :tests     ((< ?dist 0)
             (> ?idist -5)
             (< ?idist 5)))
```

```
((segment-ahead ?self ?sg)
 :percepts  ((self ?self)
             (segment ?sg dist ?dist angle ?angle))
 :relations ((not (in-segment ?self ?sg))
             (not (in-intersection ?self ?int)))
 :tests     ((> ?dist 0)
             (> ?angle -40)
             (< ?angle 40)))

((segment-ahead ?self ?sg)
 :percepts  ((self ?self)
             (segment ?sg dist ?dist angle ?angle))
 :relations ((not (in-segment ?self ?sg))
             (in-intersection ?self ?int))
 :tests     ((> ?dist 0)
             (> ?angle -45)
             (< ?angle 45)))

((segment-behind ?self ?sg)
 :percepts  ((self ?self)
             (segment ?sg dist ?dist angle ?angle))
 :relations ((not (in-segment ?self ?sg))
             (not (in-intersection ?self ?int)))
 :tests     ((< ?dist 0)
             (or (< ?angle -140) (> ?angle 140))))

((segment-behind ?self ?sg)
 :percepts  ((self ?self)
             (segment ?sg dist ?dist angle ?angle))
 :relations ((not (in-segment ?self ?sg))
             (in-intersection ?self ?int))
 :tests     ((< ?dist 0)
             (or (< ?angle -135) (> ?angle 135))))

((intersection-ahead ?self ?int ?cross)
 :percepts  ((self ?self)
             (intersection ?int street ?st cross ?cross dist ?dist))
 :relations ((on-street ?self ?st)
             (not (in-intersection ?self ?int)))
 :tests     ((> ?dist 0)))

((intersection-ahead ?self ?int ?cross)
 :percepts  ((self ?self)
             (intersection ?int street ?cross cross ?st dist ?dist))
 :relations ((on-street ?self ?st)
             (not (in-intersection ?self ?int)))
 :tests     ((> ?dist 0)))

((intersection-behind ?self ?int ?cross)
 :percepts  ((self ?self)
             (intersection ?int street ?st cross ?cross dist ?dist))
 :relations ((on-street ?self ?st)
             (not (in-intersection ?self ?int)))
 :tests     ((< ?dist -5)))

((intersection-behind ?self ?int ?cross)
 :percepts  ((self ?self)
             (intersection ?int street ?cross cross ?st dist ?dist))
 :relations ((on-street ?self ?st)
             (not (in-intersection ?self ?int)))
 :tests     ((< ?dist -5)))
```

```
((close-to-intersection ?self ?int)
 :percepts  ((self ?self)
             (intersection ?int street ?st dist ?dist))
 :relations ((on-street ?self ?st)
             (not (in-intersection ?self ?int)))
 :tests     ((< ?dist 40)
             (> ?dist 0)))

((close-to-intersection ?self ?int)
 :percepts  ((self ?self)
             (intersection ?int cross ?st dist ?dist))
 :relations ((on-street ?self ?st)
             (not (in-intersection ?self ?int)))
 :tests     ((< ?dist 40)
             (> ?dist 0)))

((in-intersection-for-right-turn ?self ?int ?current ?target)
 :percepts  ((self ?self)
             (intersection ?int street ?current cross ?target)
             (street ?target))
 :relations ((on-street ?self ?current)
             (in-intersection ?self ?int)))

((in-intersection-for-right-turn ?self ?int ?current ?target)
 :percepts  ((self ?self)
             (intersection ?int street ?target cross ?current)
             (street ?target))
 :relations ((on-street ?self ?current)
             (in-intersection ?self ?int)))

((ready-for-right-turn ?self)
 :relations ((in-rightmost-lane ?self ?line1 ?line2)
             (at-turning-speed ?self)))

((ready-for-right-turn ?self)
 :relations ((in-intersection ?self ?int)))

((no-road-ahead ?self)
 :percepts  ((self ?self))
 :relations ((not (segment-ahead ?self ?sg))))

((facing-street ?self ?target)
 :percepts  ((self ?self)
             (street ?target)
             (lane-line ?line segment ?sg angle ?angle)
             (segment ?sg street ?target))
 :tests     ((>= ?angle -3)
             (<= ?angle 3)))

((on-street ?self ?st)
 :percepts  ((self ?self)
             (segment ?sg street ?st)
             (street ?st))
 :relations ((in-segment ?self ?sg)))

((on-street ?self ?st)
 :percepts  ((self ?self)
             (intersection ?int street ?st)
             (segment ?sg street ?st)
             (street ?st))
 :relations ((in-intersection ?self ?int)
             (segment-ahead ?self ?sg)))
```

```
((on-street ?self ?st)
 :percepts  ((self ?self)
             (intersection ?int cross ?st)
             (segment ?sg street ?st)
             (street ?st))
 :relations ((in-intersection ?self ?int)
             (segment-ahead ?self ?sg)))

((on-street ?self ?st)
 :percepts  ((self ?self)
             (intersection ?int street ?st)
             (segment ?sg street ?st)
             (street ?st))
 :relations ((in-intersection ?self ?int)
             (not (segment-ahead ?self ?any))
             (segment-behind ?self ?sg)))

((on-street ?self ?st)
 :percepts  ((self ?self)
             (intersection ?int cross ?st)
             (segment ?sg street ?st)
             (street ?st))
 :relations ((in-intersection ?self ?int)
             (not (segment-ahead ?self ?any))
             (segment-behind ?self ?sg)))


skills:

((at-cruising-speed ?self)
 :percepts  ((self ?self speed ?speed))
 :start     ((slow-for-cruise ?self))
 :actions   ((*gas 20)))

((at-cruising-speed ?self)
 :percepts  ((self ?self))
 :start     ((fast-for-cruise ?self))
 :requires  ((fast-for-cruise ?self))
 :actions   ((*cruise)))

((at-turning-speed ?self)
 :percepts  ((self ?self speed ?speed))
 :start     ((slow-for-turns ?self))
 :actions   ((*gas 15)))

((at-turning-speed ?self)
 :percepts  ((self ?self speed ?speed))
 :start     ((fast-for-turns ?self))
 :actions   ((*brake 20)))

((in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((lane-on-left ?self ?line1 ?line2)
             (misaligned-to-left-in-lane ?self ?line3 ?line4))
 :requires  ((misaligned-to-left-in-lane ?self ?line3 ?line4))
 :actions   ((*straighten)))

((in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((lane-on-right ?self ?line1 ?line2)
             (misaligned-to-right-in-lane ?self ?line3 ?line4))
 :requires  ((misaligned-to-right-in-lane ?self ?line3 ?line4))
 :actions   ((*straighten)))
```

```
((in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((lane-on-left ?self ?line1 ?line2)
             (too-misaligned-to-left-in-lane ?self ?line3 ?line4))
 :requires  ((too-misaligned-to-left-in-lane ?self ?line3 ?line4))
 :actions   ((*straighten)))

((in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((lane-on-right ?self ?line1 ?line2)
             (too-misaligned-to-right-in-lane ?self ?line3 ?line4))
 :requires  ((too-misaligned-to-right-in-lane ?self ?line3 ?line4))
 :actions   ((*straighten)))

((in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((lane-on-left ?self ?line1 ?line2))
 :requires  ((not (misaligned-to-left-in-lane ?self ?line3 ?line4))
             (not (too-misaligned-to-left-in-lane ?self ?line3 ?line4)))
 :actions   ((*steer -30)))

((in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((lane-on-right ?self ?line1 ?line2))
 :requires  ((not (misaligned-to-right-in-lane ?self ?line3 ?line4))
             (not (too-misaligned-to-right-in-lane ?self ?line3 ?line4)))
 :actions   ((*steer 30)))

((in-leftmost-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((leftmost-lane ?line1 ?line2))
 :subgoals  ((in-lane ?self ?line1 ?line2)))

((in-rightmost-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((rightmost-lane ?line1 ?line2))
 :subgoals  ((in-lane ?self ?line1 ?line2)))

((on-right-side-of-road ?self)
 :percepts  ((self ?self))
 :start     ((on-wrong-side-of-road ?self))
 :actions   ((*steer 35)))

((in-intersection-for-right-turn ?self ?int ?current ?target)
 :percepts  ((self ?self)
             (street ?current)
             (street ?target)
             (intersection ?int))
 :start     ((on-street ?self ?current)
             (ready-for-right-turn ?self))
 :requires  ((not (in-intersection-for-right-turn ?self ?any ?current ?other)))
 :actions   ((*cruise)
             (*straighten)))

((facing-street ?self ?target)
 :percepts  ((self ?self)
             (street ?target)
             (segment ?sg angle ?angle))
 :start     ((segment-on-right ?self ?sg ?target))
 :actions   ((*steer ?angle)))
```

```
((on-street ?self ?target)
 :percepts  ((self ?self)
             (street ?target)
             (intersection ?int))
 :start     ((in-intersection-for-right-turn ?self ?int ?current ?target)
             (at-turning-speed ?self)
             (facing-street ?self ?target))
 :actions   ((*cruise)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((too-misaligned-to-left-in-lane ?self ?line1 ?line2))
 :actions   ((*steer 30)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((too-misaligned-to-right-in-lane ?self ?line1 ?line2))
 :actions   ((*steer -30)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((misaligned-to-left-in-lane ?self ?line1 ?line2))
 :actions   ((*steer 20)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((misaligned-to-right-in-lane ?self ?line1 ?line2))
 :actions   ((*steer -20)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((almost-aligned-in-lane ?self ?line1 ?line2)
             (slightly-uncentered-to-left-in-lane ?self ?line1 ?line2))
 :actions   ((*steer 5)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((almost-aligned-in-lane ?self ?line1 ?line2)
             (slightly-uncentered-to-right-in-lane ?self ?line1 ?line2))
 :actions   ((*steer -5)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((almost-aligned-in-lane ?self ?line1 ?line2)
             (uncentered-to-left-in-lane ?self ?line1 ?line2))
 :actions   ((*steer 10)))

((aligned-and-centered-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :start     ((almost-aligned-in-lane ?self ?line1 ?line2)
             (uncentered-to-right-in-lane ?self ?line1 ?line2))
 :actions   ((*steer -10)))

((cruising-in-lane ?self ?line1 ?line2)
 :percepts  ((self ?self))
 :subgoals  ((at-cruising-speed ?self)
             (aligned-and-centered-in-lane ?self ?line1 ?line2)))

((ready-for-right-turn ?self)
 :subgoals  ((in-rightmost-lane ?self ?line1 ?line2)
             (at-turning-speed ?self)))
```

```
((on-street ?self ?target)
 :percepts  ((self ?self)
             (street ?st)
             (street ?street)
             (intersection ?int))
 :start     ((no-road-ahead ?self)
             (in-intersection-for-right-turn ?self ?int ?st ?street))
 :requires  ((no-road-ahead ?self)
             (in-intersection-for-right-turn ?self ?int ?st ?street))
 :subgoals  ((on-street ?self ?street)
             (on-street ?self ?target)))

((on-street ?self ?target)
 :percepts  ((self ?self)
             (street ?st)
             (street ?street)
             (intersection ?int))
 :start     ((no-road-ahead ?self)
             (intersection-ahead ?self ?int ?street))
 :requires  ((no-road-ahead ?self)
             (intersection-ahead ?self ?int ?street))
 :subgoals  ((ready-for-right-turn ?self)
             (in-intersection-for-right-turn ?self ?int ?st ?street)
             (on-street ?self ?street)
             (on-street ?self ?target)))

((on-street ?self ?target)
 :percepts  ((self ?self)
             (street ?st)
             (street ?street)
             (intersection ?int))
 :start     ((parallel ?target ?st)
             (in-intersection-for-right-turn ?self ?int ?st ?street))
 :requires  ((parallel ?target ?st)
             (in-intersection-for-right-turn ?self ?int ?st ?street))
 :subgoals  ((on-street ?self ?street)
             (on-street ?self ?target)))

((on-street ?self ?target)
 :percepts  ((self ?self)
             (street ?st)
             (street ?street)
             (intersection ?int))
 :start     ((parallel ?target ?st)
             (intersection-ahead ?self ?int ?street))
 :requires  ((parallel ?target ?st)
             (intersection-ahead ?self ?int ?street))
 :subgoals  ((ready-for-right-turn ?self)
             (in-intersection-for-right-turn ?self ?int ?st ?street)
             (on-street ?self ?street)
             (on-street ?self ?target)))

((on-street ?self ?target)
 :percepts  ((self ?self)
             (street ?target)
             (intersection ?int))
 :start     ((in-intersection-for-right-turn ?self ?int ?current ?target))
 :requires  ((in-intersection-for-right-turn ?self ?int ?current ?target))
 :subgoals  ((facing-street ?self ?target)
             (on-street ?self ?target)))
```

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

Appendix B

Additional Knowledge for High-level Tasks

This appendix presents the complete list of additional knowledge necessary for reactive goal management in urban driving. We refer to some of these structures in Chapter 5 when explaining components of the extended system.

```
concepts:

((pedestrian-ahead ?self ?ped)
 :percepts  ((self ?self)
             (pedestrian ?ped dist ?dist angle ?angle alive ?alive))
 :tests     ((= ?alive 1)
             (< ?dist 30)
             (< ?angle 30)
             (> ?angle -30)))

((vehicle-ahead ?self ?car)
 :percepts  ((self ?self)
             (car ?car dist ?cdist angle ?cangle value ?value)
             (lane-line ?line1 dist ?dist1 angle ?angle)
             (lane-line ?line2 dist ?dist2))
 :relations ((in-lane ?self ?line1 ?line2))
 :tests     ((< ?cdist 70)
             (> ?cangle -90)
             (< ?cangle 90)
             (> (* ?cdist (sin (/ (* (- ?cangle ?angle) pi) 90))) ?dist1)
             (< (* ?cdist (sin (/ (* (- ?cangle ?angle) pi) 90))) ?dist2)))

((clear ?self ?obj)
 :percepts  ((self ?self)
             (pedestrian ?obj))
 :relations ((not (pedestrian-ahead ?self ?obj))))
```

```
((clear ?self ?obj)
 :percepts  ((self ?self)
             (car ?obj))
 :relations ((not (vehicle-ahead ?self ?obj))))

((stopped-and-clear ?self ?obj)
 :percepts  ((self ?self)
             (pedestrian ?obj))
 :relations ((stopped ?self)
             (clear ?self ?obj)))

((signal-ahead ?self ?signal)
 :percepts  ((self ?self)
             (signal ?signal color ?color exitpath ?street dist ?dist angle ?angle))
 :relations ((on-street ?self ?street)
             (intersection-ahead ?self ?int ?target))
 :tests     ((< ?dist 50)
             (< ?angle -4)))

((okay-to-go ?self ?signal)
 :percepts  ((self ?self)
             (signal ?signal color ?color exitpath ?street))
 :relations ((on-street ?self ?street)
             (not (stop-sign ?signal)))
 :tests     ((or (equal ?color 'green)
                 (equal ?color 'greenyellow))))

((not-emergency ?self)
 :percepts  ((self ?self level ?level))
 :tests     ((= ?level 10))
 :pivot     (?level))

((at-address ?self ?address)
 :percepts  ((self ?self)
             (building ?building address ?address street ?st c1dist ?dist1
              c1angle ?angle1 c2dist ?dist2 c2angle ?angle2))
 :relations ((on-street ?self ?st)
             (in-rightmost-lane ?self ?line1 ?line2))
 :tests     ((> ?angle1 0)
             (< ?angle1 90)
             (> ?angle2 90)
             (< ?angle2 180)))



skills:

((okay-to-go ?self ?signal)
 :percepts  ((self ?self)
             (signal ?signal exitpath ?street))
 :start     ((on-street ?self ?street))
 :requires  ((signal-ahead ?self ?signal))
 :actions   ((*brake 25)))

((stopped-and-clear ?self ?obj)
 :percepts  ((self ?self)
             (pedestrian ?obj))
 :requires  ((not (stopped ?self)))
 :actions   ((*brake 1000)))

((clear ?self ?car)
 :percepts  ((self ?self))
 :start     ((vehicle-ahead ?self ?car)
             (closest-lane-on-right ?self ?line1 ?line2))
 :subgoals  ((in-lane ?self ?line1 ?line2)))
```

```
((clear ?self ?car)
 :percepts  ((self ?self))
 :start     ((vehicle-ahead ?self ?car)
             (closest-lane-on-left ?self ?line1 ?line2))
 :subgoals  ((in-lane ?self ?line1 ?line2)))
```

References

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

Anderson, J. R., & Lebiere, C. (2003). The newell test for a theory of cognition. *Behavioral and Brain Sciences*, *26*, 587–637.

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.

Asgharbeygi, N., Stracuzzi, D., & Langley, P. (2006). Relational temporal difference learning. In *Proceedings of the Twenty-Third International Conference on Machine Learning* (pp. 49–56). Pittsburgh, PA.

Bargh, J. A. (1990). Auto-motives: Preconscious determinants of thought and behavior. In E. T. Higgins & R. M. Sorrentino (Eds.), *Handbook of motivation and cognition: Foundations of social behavior* (Vol. 2, pp. 93–130). New York, NY: Guilford Press.

Beaudoin, L. P., & Sloman, A. (1993). A study of motive processing and attention. In A. Sloman, D. Hogg, G. Humphreys, A. Ramsay, & D. Partridge (Eds.), *Prospects for artificial intelligence: Proceedings of AISB-93* (pp. 229–238). Amsterdam: IOS Press.

Berenji, H. R. (1992). A reinforcement learning-based architecture for fuzzy logic control. *International Journal of Approximate Reasoning*, *6*, 267–292.

Black, M. (1937). Vagueness: an exercise in logical analysis. *Philosophy of Science*, *4*, 427–455.

Broadbent, D. E. (1958). *Perception and communication.* Elmsford, NY: Pergamon Press.

Broersen, J., Dastani, M., Hulstijn, J., & van der Torre, L. (2002). Goal generation in the BOID architecture. *Cognitive Science Quarterly*, *2*, 428–447.

Cassimatis, N. L., Bello, P., & Langley, P. (2008). Ability, breadth, and parsimony in computational models of higher-order cognition. *Cognitive Science*, *32*, 1304–1322.

Choi, D., Kang, Y., Lim, H., & You, B.-J. (2009). Knowledge-based control of a humanoid robot. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems.* St. Louis, MO: IEEE press.

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., Könik, T., Nejati, N., Park, C., & Langley, P. (2007). Structural transfer of cognitive skills. In *Proceedings of the Eighth International Conference on Cognitive Modeling.* Ann Arbor, MI.

Choi, D., & Langley, P. (2005). Learning teleoreactive logic programs from problem solving. In *Proceedings of the Fifteenth International Conference on Inductive Logic Programming* (pp. 51–68). Bonn, Germany: Springer-Verlag.

Craik, K. J. W. (1948). Theory of the human operator in control systems: II. Man as an element in a control system. *British Journal of Psychology*, *38*, 142–148.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, *12*(3), 231–272.

Firby, R. J. (1994). Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems* (pp. 49–54).

Forgy, C. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*(1), 17–37.

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.

Genesereth, M. R., Love, N., & Pell, B. (2005). General Game Playing: Overview of the AAAI competition. *AAAI Magazine*, *26*(2), 62–72.

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.

Gordon, E., & Logan, B. (2005). Managing goals and resources in dynamic environments. In D. N. Davis (Ed.), *Visions of mind: Architectures for cognition and affect* (pp. 225–253). Idea Group.

Gray, J. R., & Braver, T. S. (2002). Integration of emotion and cognitive control: A neurocomputational hypothesis of dynamic goal regulation. In S. C. Moore & M. Oaksford (Eds.), *Emotional cognition: From brain to behaviour* (pp. 289–316). Philadelphia, PA: John Benjamins Publishing Company.

Hanheide, M., Hawes, N., Wyatt, J., Göbelbecker, M., Brenner, M., Sjöö, K., et al. (2010). A framework for goal generation and management. In *Proceedings of the AAAI-2010 Workshop on Goal-Directed Autonomy.* Atlanta, GA: AAAI Press.

Hawes, N., Zender, H., Sjöö, K., Brenner, M., Kruijff, G.-J. M., & Jensfelt, P. (2009). Planning and acting with an integrated sense of space. In *Proceedings of the First International Workshop on Hybrid Control of Autonomous Systems* (pp. 25–32).

Jennings, N. R., & Cohn, A. G. (2006). Motivation, planning and interaction. In R. Morris, L. Tarassenko, & M. Kenward (Eds.), *Cognitive systems: Information processing meets brain science* (pp. 163–188). Elsevier.

Kahneman, D. (1973). *Attention and effort*. Englewood Cliffs: NJ: Prentice Hall.

Keele, S. W. (1973). *Attention and human performance*. Pacific Palisades, CA: Goodyear.

Könik, T., O'Rorke, P., Shapiro, D., Choi, D., Nejati, N., & Langley, P. (2009). Skill transfer through goal-driven representation mapping. *Cognitive Systems Research*, *10*(3), 270–285.

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. (2009). Acquisition of hierarchical reactive skills in a unified cognitive architecture. *Cognitive Systems Research*, *10*(4), 316–332.

Luck, M., & d'Inverno, M. (1998). Motivated behaviour for goal adoption. In C. Zhang & D. Lukose (Eds.), *Multi-agent systems: Theories, languages and applications - Proceedings of the Fourth Australian Workshop on Distributed Artificial Intelligence* (pp. 58–73). Springer-Verlag.

Meyer, D. E., & Kieras, D. E. (1997). A computational theory of executive cognitive processes and multiple-task performance: Part 1. Basic mechanisms. *Psychological Review*, *104*, 3–65.

Miller, G. A., Galanter, E., & Pribram, K. H. (1960). *Plans and the structure of behavior*. New York: Holt, Rinehart & Winston.

Molineaux, M., Klenk, M., & Aha, D. W. (2010). Goal-driven autonomy in a Navy strategy simulation. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. Atlanta, GA: AAAI Press.

Moskowitz, G. B., & Gesundheit, Y. (2009). Goal priming. In G. B. Moskowitz & H. Grant (Eds.), *The psychology of goals* (pp. 203–233). New York, NY: Guilford Press.

Myers, K. L. (1996). A procedural knowledge approach to task-level control. In *Proceedings of the Third International Conference on AI Planning Systems.*

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., et al. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, *20*, 379–404.

Navon, D., & Gopher, D. (1979). On the economy of the human processing system. *Psychological Review*, *86*(3), 214–255.

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

Nilsson, N. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, *1*, 139–158.

Norman, D. A., & Bobrow, D. G. (1975). On data-limited and resource-limited processes. *Cognitive Psychology*, *7*, 44–64.

Norman, D. A., & Shallice, T. (1986). Attention to action: Willed and automatic control of behavior. In R. J. Davidson, G. E. Schwartz, & D. Shapiro (Eds.), *Consciousness and self-regulation: Advances in research and theory* (pp. 1–18). New York: Plenum Press.

Ohlsson, S., & Rees, E. (1991). Adaptive search through constraint violations. *Journal of Experimental & Theoretical Artificial Intelligence*, *3*, 33–42.

Pashler, H. (1984). Processing stages in overlapping tasks: Evidence for a central bottleneck. *Journal of Experimental Psychology: Human Perception and Performance*, *10*(3), 358–377.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine learning*, *62*, 107–136.

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

Simon, H. A. (1967). Motivational and emotional controls of cognition. *Psychological Review*, *74*(1), 29–39.

Sloman, A. (1987). Motives, mechanisms, and emotions. *Cognition & Emotion*, *1*(3), 217–233.

Sloman, A. (2002). How many separately evolved emotional beasties live within us? In R. Trappl, P. Petta, & S. Payr (Eds.), *Emotions in humans and artifacts* (pp. 35–114). MIT Press.

Smith, M. C. (1967). Theories of the psychological refractory period. *Psychological Bulletin*, *67*, 202–213.

Stracuzzi, D. J., Li, N., Cleveland, G., & Langley, P. (2009). Representing and reasoning over time in a symbolic cognitive architecture. In *Proceedings of the Thirty-First Annual Meeting of the Cognitive Science Society* (pp. 2986–2991). Amsterdam, Netherlands: Cognitive Science Society, Inc.

Sun, R. (2007). The motivational and metacognitive control in CLARION. In W. Gray (Ed.), *Modeling integrated cognitive systems* (pp. 63–75). New York, NY: Oxford University Press.

Taatgen, N. (2005). Modeling parallelization and flexibility improvements in skill acquisition: from dual tasks to complex dynamic skills. *Cognitive Science*, *29*, 421–455.

Telford, C. W. (1931). The refractory phase of voluntary and associative response. *Journal of Experimental Psychology*, *14*(1), 1–36.

Thomason, R. H. (2000). Desires and defaults: A framework for planning with inferred goals. In *Proceedings of the Seventh International Conference on the Principles of Knowledge Representation and Reasoning* (pp. 702–713). San Mateo, CA: Morgan Kaufmann.

Tulving, E. (1972). Episodic and semantic memory. In E. Tulving & W. Donaldson (Eds.), *Organization of memory* (pp. 381–403). New York: Academic Press.

Vince, M. A. (1948). The intermittency of control movements and the psychological refractory period. *British Journal of Psychology*, *38*, 149–157.

Welford, A. T. (1952). The "psychological refractory period" and the timing of high speed performance: A review and a theory. *British Journal of Psychology*, *43*, 2–19.

Welford, A. T. (1967). Single channel operation in the brain. *Acta Psychologica*, *27*, 5–22.

Wickens, C. D. (1984). Processing resources in attention. In R. Parasuraman, J. Beatty, & R. Davies (Eds.), *Varieties of attention* (pp. 63–101). New York, NY: Wiley.

Youngblood, G. M., Nolen, B., Ross, M., & Holder, L. (2006). Building test beds for AI with the Q3 mod base. In *Proceedings of the Annual Artificial Intelligence for Interactive Digital Entertainment Conference* (p. 153-154). Marina del Rey, CA: AAAI Press.

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, *8*(3), 338–353.