ICARUS User's Manual

PAT LANGLEY (LANGLEY@CSLI.STANFORD.EDU)
DONGKYU CHOI (DONGKYUC@STANFORD.EDU)

Computational Learning Laboratory
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305 USA

Abstract

In this document, we introduce ICARUS, a cognitive architecture for physical agents that utilizes hierarchical concepts and skills for inference, execution, and problem solving. We first review the assumptions typically made in work on cognitive architectures and explain how ICARUS differs from earlier candidates. After this, we present the framework's approach to conceptual inference, its mechanisms for teleoreactive execution, its processes for means-ends problem solving, and its techniques for learning new skills. In each case, we discuss the memories on which the processes rely, the structures they contain, and the manner in which they operate. In closing, we describe the commands available for creating ICARUS programs, running them in simulated environments, and tracing their behavior.

Draft 3
January 24, 2008

Comments on this document are welcome, but please do not distribute it without permission.

1. Introduction

This document describes Icarus, a cognitive architecture that supports the development of intelligent agents. In writing it, we have attempted to balance two aims: conveying the framework's theoretical commitments about memories, representations, and processes; and providing readers with enough details to use the associated software to construct and run interesting agents. Because some readers may not be familiar with the cognitive architecture paradigm, we start by reviewing this notion and the assumptions it makes, along with Icarus' relation to alternative proposals. After this, we present the architecture's four modules and the memories on which they rely, illustrating each with simple examples. We conclude by discussing the programming language that lets users construct intelligent agents and run them in simulated environments.

1.1 Aims of Icarus

Like other cognitive architectures, Icarus attempts to support a number of goals. First, it aims to provide a computational theory of intelligent behavior that addresses interactions among different facets of cognition. We can contrast this objective with most research in artificial intelligence and cognitive science, which focuses on individual components of intelligence, rather than on how they fit together. Thus, Icarus offers a systems-level account of intelligent behavior.

However, there are different paths to developing complex intelligent systems. The most common view, assumed in software engineering and multi-agent systems, is to design modules as independently as possible and let them interact through well-defined communication protocols. Research on cognitive architectures, Icarus included, instead aims to move beyond such integrated systems to provide a unified theory. The resulting artifacts still have identifiable modules, but they are strongly constrained and far from independent. The interacting constraints embody theoretical commitments about the character of intelligence.

Unlike most AI research, the cognitive architecture movement borrows many ideas and terms from the field of cognitive psychology. Different frameworks take distinct positions on whether they attempt to model the details of human behavior. Like Soar (Laird, Newell, & Rosenbloom, 1987) and Prodigy (Carbonell, Knoblock, & Minton, 1990), Icarus incorporates many findings about high-level cognition because they provide useful insights into how one might construct intelligent systems, rather than aiming to model human cognition in its own right. Of course, this does not keep one from using Icarus to model particular psychological phenomena (e.g., Langley & Rogers, 2005), but we will not emphasize that capability here.

However, we definitely want the framework to support the effective and efficient construction of intelligent systems. Like other cognitive architectures, Icarus rejects the common idea that this is best accomplished with software libraries and instead offers a high-level programming language for specifying agent behavior. We will see that this formalism's syntax is tied closely to Icarus' theoretical commitments, giving the sections that follow the dual charge of presenting both the language itself and its connection to these ideas.

As a side effect of these varied goals, Icarus also makes contributions to research on knowledge representation and on mechanisms that manipulate mental structures. These include the performance processes that draw inferences, execute procedures, and solve novel problems, as well

as the learning mechanisms that create and revise these structures. However, we will not focus here on their individual contributions, since we are concerned with communicating how Icarus' components work together to generate intelligent behavior and how one can craft agents that take advantage of these capabilities.

1.2 Commitments of Cognitive Architectures

Newell's (1990) vision for research on cognitive architectures held that they should incorporate strong assumptions about the nature of the mind. In his view, a cognitive architecture specifies the underlying infrastructure for an intelligent system that is constant over time and across different application domains. Consider the architecture for a building, which consists of its stable features, such as the foundation, roof, walls, ceilings, and floors, but not those that one can move or replace, such as the furniture and appliances. By analogy, a cognitive architecture also consists of its permanent features rather than its malleable components.

Over the years, research in this area has converged on a number of key properties that define a given cognitive architecture and that constitute its theoretical claims. These include:

- the short-term and long-term memories that store the agent's beliefs, goals, and knowledge, along with the characteristics of those memories;
- the representation of elements that are contained in these memories and their organization into larger-scale mental structures;
- the functional processes that operate on these structures, including the performance mechanisms that utilize them and the learning mechanisms that alter them.

Because the contents of an agent's memories can change over time, we would not consider the knowledge encoded therein to be part of that agent's architecture. Just as different programs can run on the same computer architecture, so different knowledge bases can be interpreted by the same cognitive architecture.

Of course, distinct architectures may differ in the specific assumptions they make about these issues. In addition to making different commitments about how to represent, use, or acquire knowledge, alternative frameworks may claim that more or less is built into the architectural level. For example, Icarus views some capabilities as unchanging that Soar views as encoded in knowledge. Such assumptions place constraints on how one constructs intelligent agents within a given architecture. They also help determine the syntax and semantics of the programming language that comes with the architecture for use in constructing knowledge-based systems.

Although research on cognitive architectures aims to determine the unchanging features that underlie intelligence, in practice a framework will change over time as its developers determine that new structures and processes are required to support new functionality. However, an architectural design should be revised gradually, and with caution, after substantial evidence has been accumulated that it is necessary. Moreover, early design choices should constrain those made later in the process, so that new modules and capabilities build on what has come before. The current version of Icarus has evolved in this manner over a number of years, and we intend future revisions to follow the same guide.

1.3 Theoretical Claims of Icarus

As suggested above, Icarus has much in common with previous cognitive architectures like Soar (Laird et al., 1987), ACT-R (Anderson, 1993), and Prodigy (Carbonell et al., 1990). Like its predecessors, the framework makes strong commitments about memories, representations, and cognitive processes that support intelligent behavior. Some shared assumptions include claims that:

- short-term memories, which contain dynamic information, are distinct from long-term memories, which store more stable content;
- both short-term and long-term memories contain modular elements that can be composed dynamically during performance and learning;
- these memory elements are cast as symbolic list structures, with those in long-term memory being accessed by matching their patterns against elements in short-term memory;
- cognitive behavior occurs in cycles that first retrieve and instantiate relevant long-term structures, then use selected elements to carry out mental or physical actions; and
- learning is incremental and tightly interleaved with performance, with structural learning involving the monotonic addition of new symbolic structures to long-term memory.

Most of these ideas have their origins in theories of human memory, problem solving, reasoning, and skill acquisition. They are widespread in research on cognitive architectures, but they are relatively rare in other branches of artificial intelligence and computer science.

Despite these similarities, Icarus also incorporates some theoretical claims that distinguish it from other architectures. These include assumptions that:

- cognition occurs in the context of physical environments, and mental structures are ultimately grounded in perception and action;
- concepts and skills encode different aspects of knowledge, and they are stored as distinct but interconnected cognitive structures;
- each element in a short-term memory must have a corresponding generalized structure in some long-term memory, with the former being an instance of the latter;
- the contents of long-term memories are organized in a hierarchical fashion that defines more complex structures in terms of simpler ones;
- conceptual inference and skill execution are more basic to the architecture than problem solving, which builds on these processes;
- both skill and concept hierarchies are acquired in a cumulative manner, with simpler structures being learned before more complex ones.

These ideas distinguish Icarus from most other cognitive architectures that have been developed within the Newell tradition. We will not argue that they make it superior to earlier frameworks, but we believe they do make Icarus an interesting alternative within the space of candidate architectures, as we hope becomes apparent in the pages that follow.

Many of these claims involve matters of emphasis rather than irreconcilable differences. For example, Soar and ACT-R have been extended to interface with external environments, but both frameworks focused initially on central cognition, whereas Icarus began as an architecture for reactive execution and places greater importance on interaction with the physical world. ACT-R states that elements in short-term memory are active versions of structures in long-term declarative memory, but does not make Icarus' stronger claim that the former must be specific instances of the latter. Soar incorporates an elaboration stage that plays a similar role to conceptual inference in our architecture, although our mechanism links this process to a conceptual hierarchy that Soar lacks. ACT-R programs often include production rules that match against goals and set subgoals, whereas Icarus elevates this idea to an architectural principle about the hierarchical organization of skills. These similarities reflect an underlying concern with many of the same issues, but they also reveal distinct philosophies about how to approach them.

1.4 List Structures and Pattern Matching

Icarus relies on two ideas which may be unfamiliar to those with training in mainstream computer science – list structures and pattern matching – that nevertheless are far from new. The first list-processing language, IPL (Newell & Shaw, 1957; Newell & Tonge, 1960), was developed in the 1950s about the same time as Fortran. The fields of artificial intelligence and cognitive science have relied heavily on list processing throughout their history, and techniques for pattern matching over such structures have been utilized for just as long (Newell & Shaw, 1957).

Like other cognitive architectures, Icarus uses list structures to encode both its programs and the data over which those programs operate. The building blocks of such structures are atoms – opaque symbols like *on* and *blockA*, numbers like *-5* and *3.21*, or Boolean truth values like *T* and *NIL*. A *list* is an ordered set of such atoms delimited by parentheses, as in *(on blockA blockB)* or *(block blockA height 1 width 2)*. A *list structure* is simply a list that contains atoms, lists, or other list structures as its elements, as in *(a (b c (d e) (f)) g h)*. A list structure can be viewed as a tree in which each sublist corresponds to a subtree and each atom to a terminal node. They are well suited for representing beliefs, goals, and other content that arises in building intelligent agents.

However, the examples above are all highly specific, in that they refer to particular objects or entities. We would like the programs that operate over these structures to be more general, and we can achieve this end by using symbolic *patterns*. These also take the form of list structures, but they replace some elements with a special form of atom known as a *pattern-match variable*, which we denote here with a leading question mark, as in *?x* or *?any-block*. The resulting structures are more general than ones without variables in that they can match against any specific structures that bind variables in a consistent way. For example, the pattern *(on ?x ?y)* would match against *(on blockA blockB)* with *?x* binding to *blockA* and *?y* to *blockB*, but it would also match against *(on blockB blockC)* with *?x* binding to *blockB* and *?y* to *blockC*.

The notion of consistent binding becomes more important when dealing with sets of conjunctive patterns, such as *((on ?x ?y) (on ?y ?z))*. Suppose we want to find all ways in which this set matches against the unordered set of specific elements *((on blockA blockB) (on blockB blockC) (on blockC blockD))*. In this case, there are two consistent bindings: *?x* to *blockA*, *?y* to *blockB*, *?z* to

blockC, which maps the two patterns onto the first two elements, and *?x* to *blockB*, *?y* to *blockC*, *?z* to *blockD*, which maps them onto the last two elements. We will refer to each set of consistent bindings as an *instantiation*. The pattern does not produce an instantiation for the first and third elements because there is no way to replace the variables in the pattern with concrete symbols that will produce these specific structures.

We will not go into the technical details of pattern matching here, since they are available in textbooks (e.g., Norvig, 1992) and they are not required to understand its use. But we should note that short-term memories in Icarus typically contain specific elements (sometimes called *ground literals*), whereas its long-term memories usually contain general patterns, with the latter serving as programs that manipulate the former as data. Matching patterns plays a central role in Icarus' interpretation of these programs, as it does in other cognitive architectures.

1.5 Organization of the Document

We might have organized this document in a number of different ways. One conventional scheme would first describe the natre and content of the architecture's memories, then turn to the mechanisms that operate over them. However, Icarus' various processing modules interact with some memories but not others, which suggests that we might instead organize the text around these modules and the memories on which they depend. Moreover, some of the architecture's mechanisms build directly on other processes, which suggests a natural order of presentation.

For these reasons, we first discuss Icarus' most basic mechanism, conceptual inference, along with the short-term and long-term memories that it inspects and alters. After this, we present the processes for goal selection and skill execution, which take as input the results of inference, along with the additional memories that they utilize. Next, we consider the architecture's module for problem solving, which builds on both inference and execution, after which we examine its learning processes, which operate over the results of the agent's problem solving. In closing, we cover the details needed to write, load, run, and trace Icarus programs, including the parameters that control its behavior.

2. Beliefs, Concepts, and Inference

In order to carry out actions that achieve its goals, an agent must understand its current situation. Icarus incorporates a module for this cognitive task that operates by matching conceptual structures against perceived objects and inferred beliefs. However, before we can describe the process itself, we must first examine the contents and representation of elements on which it operates, including the long-term and short-term memories in which they reside. We will take our examples from the Blocks World, which many readers with backgrounds in artificial intelligence or cognitive science should find familiar.

2.1 The Perceptual Buffer

Recall that Icarus is designed to support intelligent agents which operate in some external environment, and which thus require information about the state of its surroundings. To this end, it incorporates a memory called the *perceptual buffer* that describes aspects of the environment

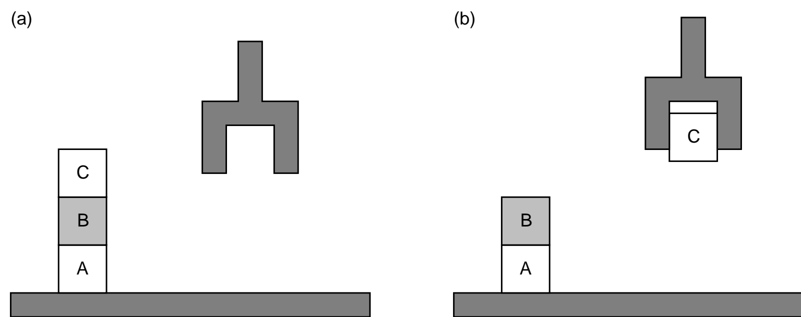

Figure 1. Two states from the Blocks World domain used to illustrate ICARUS' representations and processes.

the agent perceives on the current cycle. Each element, or *percept*, in this short-term memory corresponds to a particular object and specifies the object's type, a unique name, and a set of attribute-value pairs that characterize the object on the current time step. The values of attributes are typically numeric, but they may be symbols. They can even refer to the name of another object, giving the ability to encode structured entities. However, an attribute's value must be an atom and may not be a list or list structure.

Figure 1 depicts a simple situation from the Blocks World that involves three blocks, a table, and a robot hand. Table 1 shows the contents of a perceptual buffer that describes this situation. This includes one percept for each object, three of which have type *block*, one of which has type *table*, and one of which has type *hand*. Each block has four attributes – *xpos*, *ypos*, *width*, and *height* – all of which take numeric values. The table has similar attributes, but the hand has only the attribute *status*, which takes on either the symbolic value *empty* or the name of the block it holds.

The specific characteristics of percepts are not part of the ICARUS architecture; they are determined by the simulated environment in which the agent operates. Other environments, including different simulations of the Blocks World, may have different types and numbers of objects, as well as alternative attributes that describe them. However, the programmer must ensure that the simulator provides percepts to the architecture in the attribute-value format described above.

2.2 Belief Memory

Although one could create an agent that operates directly on perceptual information, its behavior would not reflect what we normally mean by the term 'intelligent'. Thus, ICARUS also includes a *belief memory* that contains higher-level inferences about the agent's situation. Whereas percepts describe attributes of specific objects, beliefs are inherently relational. They may involve isolated objects, such as individual blocks, but they typically characterize physical relations among objects, such as the relative positions of blocks. Each element in this belief memory is a list that consists of a predicate followed by a set of symbolic arguments.[1] Each argument in a belief must refer to some object that appears in the perceptual buffer.

Table 2 presents some plausible contents of belief memory for the Blocks World situation in Figure 1 based on the percepts in Table 1. Each belief corresponds to some *concept* which appears

1. Some readers will recognize these as analogous to the ground literals that appear in the PROLOG language.

Table 1. Contents of ICARUS' perceptual buffer for the two Blocks World states from Figure 1.

(a)	(b)
(block A xpos 10 ypos 2 width 2 height 2) (block B xpos 10 ypos 4 width 2 height 2) (block C xpos 10 ypos 6 width 2 height 2) (table T1 xpos 20 ypos 0 width 20 height 2) (hand H1 status empty)	(block A xpos 10 ypos 2 width 2 height 2) (block B xpos 10 ypos 4 width 2 height 2) (block C xpos 10 ypos 16 width 2 height 2) (table T1 xpos 20 ypos 0 width 20 height 2) (hand H1 status C)

in a separate long-term memory that we will discuss shortly. Instances of primitive concepts, such as *(on B C)*, *(ontable C T1)*, and *(holding A)*, are supported directly by percepts. In contrast, instances of nonprimitive concepts, like *(stackable A B)* and *(unstacked A B)*, are supported by other concept instances, like *(clear B)* and *(holding A)*. Belief memory does not encode these support relations explicitly, but they are apparent from examining the concept definitions we present in the next subsection.

ICARUS imposes a strong correspondence between belief memory and conceptual memory, in that every predicate in the former must refer to some concept defined in the latter. An ICARUS agent cannot represent short-term beliefs unless they have some long-term analog. However, for the sake of programming convenience, one can specify a set of static beliefs, such as *(wider T1 A)*, that will not change over time and thus need not be linked to either long-term concepts or to percepts. Such facts also reside in belief memory, despite their nondynamic character.

2.3 Conceptual Memory

We have noted that ICARUS beliefs are instances of concepts, which means that the architecture must store definitions of these concepts. These reside in conceptual memory, which contains long-term structures that describe classes of situations in the environment. The formalism used to state these logical concepts is similar to that for Horn clauses, which are central to the programming language PROLOG (Clocksin & Mellish, 1981). Like beliefs, concepts in ICARUS are inherently symbolic and relational structures.

Each clause in the long-term conceptual memory includes a head that specifies the concept's name and arguments, along with a body that states the conditions under which the clause should match against the contents of short-term memories. This body includes a `:percepts` field, which describes perceived objects that must be present, a `:relations` field, which gives lower-level concepts that must match or not match, and a `:tests` field, which specifies numeric relations and other Boolean constraints that must be satisfied. Table 3 presents some sample concepts from the Blocks World. For instance, the relation *on* describes a perceived situation in which two blocks have the same x position and the bottom of one has the same y position as the top of the other. The concept *clear* instead refers to a single block, but one that cannot serve as the second argument to any belief that involves an *on* relation.

Table 2. Contents of Icarus' belief memory for the two Blocks World states from Figure 1.

(a)	(b)
(unstackable C B) (three-tower C B A T1) (hand-empty) (clear C) (ontable A T1) (on B A) (on C B)	(putdownable C T1) (stackable C B) (holding C) (clear B) (clear C) (ontable A T1) (on B A)

Let us consider the contents of each component of a conceptual clause in more detail. The `:percepts` field refers to objects that the agent has sensed in the external environment. Each element specifies the object type, a variable (starting with a question mark) that matches against the object's name, and a set of attribute-value pairs that refer to perceived characteristics of the object. These descriptions are unordered and include only those attributes that are relevant to the current concept, even if the matched object has additional features.

The `:relations` field contains a set of relational structures which refer to other concepts that must match consistently against elements in belief memory for the clause to match. A positive literal specifies a concept name followed by variables that match against the objects that serve as its arguments. If the same variable occurs in more than one element, it must match against the same object. The same holds for variables already mentioned in the head or the `:percepts` field. Positive conditions may also introduce new variables not mentioned in the head or percepts.

The `:relations` field may also refer to relational literals that must *not* be present for the clause to match. These take the same form as positive conditions, except that this structure is the second element in a list that begins with *not*. If such a negated condition refers to a variable that occurs in a positive condition in the `:percepts` fields, then it refers to the same object. However, variables that do not occur in positive conditions are unconstrained. Such *unbound* variables are universally quantified, in that a negated condition containing them is satisfied only if there exists no element in belief memory that matches against them consistently. Moreover, if the same unbound variable appears in different negated conditions, they need not refer to the same object. This limits the representational power of negations but not that of Icarus. If relations among negated conditions are important, one can define a concept that encodes those relations and negate it instead.

The `:tests` field contains a set of arithmetic or logical functions, each of which returns true (T) or false (NIL). Each element may contain variables, but these must also occur in the `:relations` or `:percepts` field to ensure they are bound. Although each top-level function must be Boolean in nature, it may call other functions that carry out arithmetic calculations, such as addition and division. However, the purpose of this field is to support simple, constrained tests on the values of perceptual attribute. An Icarus program should not hide complex computations within the `:tests` field of its various concept definitions.

Table 3. Some ICARUS concepts for the Blocks World, with pattern-match variables indicated by question marks. Percepts refer only to objects and attribute values used elsewhere in the concept definition.

```
((on ?block1 ?block2)
 :percepts  ((block ?block1 xpos ?xpos1 ypos ?ypos1)
             (block ?block2 xpos ?xpos2 ypos ?ypos2 height ?height2))
 :tests     ((equal ?xpos1 ?xpos2) (>= ?ypos1 ?ypos2) (<= ?ypos1 (+ ?ypos2 ?height2))))
((ontable ?block1 ?block2)
 :percepts  ((block ?block1 xpos ?xpos1 ypos ?ypos1)
             (table ?block2 xpos ?xpos2 ypos ?ypos2 height ?height2))
 :tests     ((>= ?ypos1 ?ypos2) (<= ?ypos1 (+ ?ypos2 ?height2))))
((clear ?block)
 :percepts  ((block ?block))
 :relations ((not (on ?other-block ?block))))
((holding ?block)
 :percepts  ((hand ?hand status ?block) (block ?block)))
((hand-empty)
 :percepts  ((hand ?hand status empty))
 :relations ((not (holding ?any))))
((three-tower ?x ?y ?z ?table)
 :percepts  ((block ?x) (block ?y) (block ?z) (table ?table))
 :relations ((on ?x ?y) (on ?y ?z) (ontable ?z ?table)))
((unstackable ?block ?from)
 :percepts  ((block ?block) (block ?from))
 :relations ((on ?block ?from) (clear ?block) (hand-empty)))
((pickupable ?block ?from)
 :percepts  ((block ?block) (table ?from))
 :relations ((ontable ?block ?from) (clear ?block) (hand-empty)))
((stackable ?block ?to)
 :percepts  ((block ?block) (block ?to))
 :relations ((clear ?to) (holding ?block)))
((putdownable ?block ?to)
 :percepts  ((block ?block) (table ?to))
 :relations ((holding ?block)))
((unstacked ?from ?block)
 :percepts  ((block ?from) (block ?block))
 :relations ((holding ?block) (not (on ?block ?from))))
((picked-up ?block ?from)
 :percepts  ((block ?block) (table ?from))
 :relations ((holding ?block) (not (ontable ?block ?from))))
```

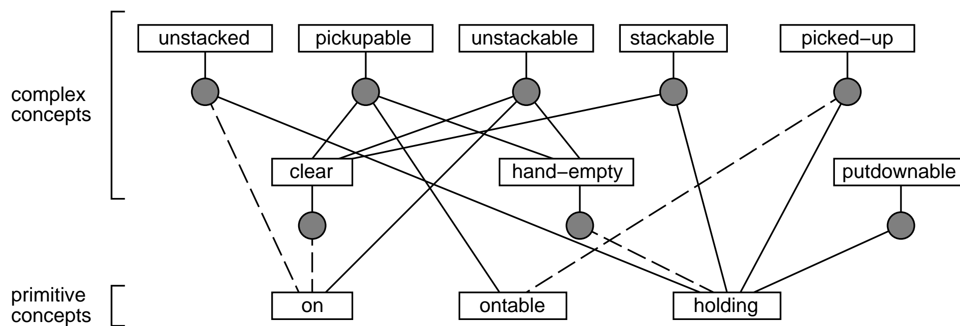

Figure 2. A graphical depiction of the hierarchy that corresponds to the conceptual clauses in Table 3, with rectangles denoting conceptual predicates and circles indicating distinct clauses. Solid lines represent positive conditions, whereas dashed lines stand for negated ones. Primitive concepts appear at the bottom, with percepts omitted.

ICARUS distinguishes between nonprimitive conceptual clauses, which contain references to other concepts in the `:relations` fields, and primitive clauses, which refer only to percepts and arithmetic tests. Taken together, the set of clauses define a conceptual hierarchy or lattice, with more basic concepts at the bottom and more complex concepts at higher levels. Figure 2 depicts the hierarchy imposed by the concepts in Table 3, with a rectangle for each predicate and a circle for each conceptual clause. This shows graphically that the concept *unstackable* is defined in terms of *on*, *clear*, and *hand-empty*, that *clear* is defined in terms of *on*, and that the predicate *on* corresponds to a primitive concept.

For readers who are familiar with production system architectures, this lattice has a structure similar to the Rete networks (Forgy, 1982) used to support efficient matching in that framework, except that each node in the ICARUS hierarchy corresponds to a meaningful concept. One key difference from Rete networks is that a conceptual predicate can appear in more than one clause, much as in the programming language PROLOG (Clocksin & Mellish, 1981). This means the conceptual hierarchy actually takes the form of an AND-OR lattice. This facility also lets one define recursive concepts, since one clause for a given concept may refer, directly or through other concepts, to itself. For instance, one might define the concept *tower* using two clauses, one for the base case and another for the recursive case. More typically, different clauses for the same concept simply specify distinct conditions under which the concept is satisfied.

2.4 Conceptual Inference

The architecture's most basic activity is conceptual inference. On each cycle, the environmental simulator returns a set of perceived objects, including their types, names, and descriptions in the format described earlier. ICARUS deposits this set of elements in the perceptual buffer, where they initiate matching against long-term conceptual definitions. The overall effect is that the system adds to its belief memory all elements that are implied deductively by these percepts and concept definitions. ICARUS repeats this process on every cycle, so that it constantly updates its beliefs about the environment.

The inference module operates in a bottom-up, data-driven manner that starts from descriptions of perceived objects. The architecture matches these percepts against the bodies of primitive concept clauses and adds any supported beliefs (i.e., concept instances) to belief memory. These trigger matching against higher-level concept clauses, which in turn produces additional beliefs. This process continues until ICARUS has added to memory all beliefs that are implied by its perceptions and concept definitions. Although this mechanism reasons over structures similar to PROLOG clauses, its operation is closer to the elaboration process in Soar (Laird et al., 1987).

A primitive conceptual clause matches when the elements in its `:percepts` field match against distinct percepts in the perceptual buffer and when, after replacing variables in the `:tests` field with values bound in the percepts, each test element returns true (T). Note that a clause can match against the contents of memory in more than one way; we refer to each such match as a concept *instantiation*. Different instantiations may or may not produce the same belief or concept *instance*, which is a predicate followed by specific arguments, such as *(on A B)*.

For example, reconsider the concept definitions in Table 3 and the percepts in Table 1. In this situation, the primitive clause for *on* would support the concept instance *(on B A)* because its first perceptual element matches against the block *B* in the perceptual buffer, binding *?block1* to *B*, its second element matches against the block *A*, binding *?block2* to *A*, and its three arithmetic tests return true given the two other bindings acquired from the percepts. In contrast, ICARUS would not infer the concept instance *(on A B)* because, although the perceptual elements would match, the second arithmetic test would fail.

A nonprimitive conceptual clause matches when the elements in its `:percepts` field match against different percepts, when the positive conditions in its `:relations` field match against beliefs that have already been inferred, provided their variables bind in a consistent way with each other and with those bound in the percepts, when none of the negated conditions in its `:relations` field match against any inferred beliefs in a manner consistent with variables already bound, and when each element in the `:tests` field returns true after replacing variables bound in other fields with their values.

For instance, the conceptual clause for *clear* supports the inference *(clear B)* from the percepts in Table 1 because its single perceptual condition matches against block *B* and because short-term memory contains no belief that matches against the partially instantiated negated condition *(on ?other-block B)*. The module infers the concept instance *(stackable C B)* because its two percepts match against the blocks *C* and *B*, and because its two positive conditions match against the beliefs *(clear B)* and *(holding C)* with bindings that are consistent with those from the percepts.

The details of ICARUS' pattern-matching method, and the order in which it infers beliefs, should not concern the reader. The important points to note are that the inference module finds, on each cycle, all beliefs about the environment that are implied logically by the contents of long-term conceptual memory and by the elements in the perceptual buffer. The resulting beliefs provide the material that the architecture uses to make decisions about other aspects of the agent's behavior, to which we now turn.

3. Goals, Skills, and Execution

We have seen that Icarus can utilize its conceptual knowledge to infer and update beliefs about its surroundings, but an intelligent agent must also take action in its environment. To this end, the architecture includes additional memories that concern goals the agent wants to achieve, skills the agent can execute to reach them, and intentions about which skills to invoke. These are linked by performance mechanisms that select goals to pursue and execute associated skills, thus changing the environment and, hopefully, taking the agent closer to these goals.

3.1 Goal Memory

To support these processes, Icarus incorporates a *goal memory* that contains the agent's top-level objectives. A goal is some conceptual literal that the agent wants to satisfy. Thus, goal memory takes much the same form as belief memory, in that each element specifies a predicate followed by its arguments, with the predicate being defined in long-term conceptual memory. As a result, goal memory has the appearance of a short-term memory, even though its top-level goals do not change during the course of a run. However, in later sections we will see that other aspects of the memory do vary across cycles.

One important difference between goals and beliefs is that goals may have as arguments either specific objects, as in *(on A B)*, pattern-match variables, as in *(on ?x ?y)*, or some mixture, as in *(on ?x B)*. Another difference is that goal memory may contain negated goals, whereas belief memory includes only positive elements. A positive goal like *(on ?x B)* specifies a situation that the agent desires to be true, whereas a negated goal like *(not (on ?x B))* indicates a situation that the agents wants *not* to hold.

Because goals always refer to predicates defined in the long-term conceptual memory, the contents of goal memory can indicate the agent's objectives at many different levels of detail. One goal may refer to a desired primitive concept that is stated in terms of percepts, while another goal may refer to a predicate that resides much higher in the concept hierarchy. For example, given the concepts from Table 3, an Icarus agent might have the low-level goal *(on A B)* or the higher-level objective *(unstacked A B)*. The current architecture does not explain the origin of such top-level goals, so the agent developer must provide them.

3.2 Goal Selection

Because goal memory may contain multiple elements, Icarus must address the issue of cognitive attention. The architecture makes the common assumption that an agent can focus on only one goal at a time, which means that it must select among the known goals on each cycle. However, if the agent believes a particular goal is satisfied, there is no reason to pursue it, so in making this selection it only considers alternatives that are currently unsatisfied.

The architecture determines whether a goal is satisfied on a given cycle by comparing it to elements in belief memory. A concrete goal like *(on A B)* is satisfied when the agent believes that *(on A B)* is true. In contrast, Icarus treats an abstract positive goal like *(on ?x B)* as existentially

quantified, in that it is satisfied if any element in belief memory, say *(on A B)*, matches it. Finally, the architecture interprets an abstract negative goal like *(not (on ?x B))* as universally quantified, in that it is satisfied onlt if belief memory contains no elements that match against its pattern.

Icarus treats the contents of goal memory as an ordered list. On each cycle, it finds the first goal in this list that is unsatisfied and makes it the focus of cognitive attention. This may mean dropping a goal that was the focus on the previous cycle if it occurs later in the list and thus has lower priority. However, the agent will eventually return to this earlier goal after it achieves the one with higher priority. If all elements in goal memory are satisfied, then Icarus has no focus on that cycle, although in dynamic environments this may change later, leading to the system refocus on goals that it achieved previously.

3.3 Skill Memory

Icarus stores knowledge about how to accomplish its goals in a long-term *skill memory* that contains skills it can execute in the environment. These take a form similar to conceptual clauses but have a somewhat different meaning because they operate over time and under the agent's intentional control. Each skill clause includes a head that specifies the skill's objective, as well as a body which indicates the concepts that must hold to initiate the skill and one or more components. The body includes a `:percepts` field, which describes perceptual entities that must be present, a `:start` field, which states the concepts that must match to initiate the clause, and a `:requires` field, which gives the concepts that must match to continue its execution.[2]

Let us consider the contents of each field in some detail. The `:percepts` field in a skill clause plays the same role, and has the same format, as the analogous field in concept clauses. Each element includes the type of perceived object, followed by a variable that matches that object's name, after which comes an unordered subset of the object's attributes (constants) and values (variables) that describe aspects of the object. The `:start` field contains a set of relational literals that must match consistently against instances of the same concepts in belief memory. Each literal includes a predicate (the concept name) followed by variables which match against objects that serve as the predicate's arguments. If a variable occurs in more than one element, in the clause head, or in the `:percepts` field, it refers to the same object. The `:requires` field contains a set of relational literals, with variables mentioned in either the start conditions or percepts constrained to match the same objects. Both start conditions and requirements may include unbound variables not mentioned previously, which have the same meaning as in conceptual clauses.

Icarus distinguishes between primitive and nonprimitive skill clauses. Primitive clauses include in their bodies an `:actions` field, which contains a list of actions that the agent can execute directly in the environment. These play the same role as Strips operators (Fikes, Hart, & Nilsson, 1972) in AI planning systems, but they can be durative in nature, in that their execution may continue across multiple cycles. In contrast, nonprimitive skill clauses include a `:subgoals` field, which refers to other goals that the agent should achieve to implement the current clause and achieve its head.

2. We do not provide examples of the `:requires` field here, as they make sense only for domains which involve primitive durative skills that require multiple cycles to complete.

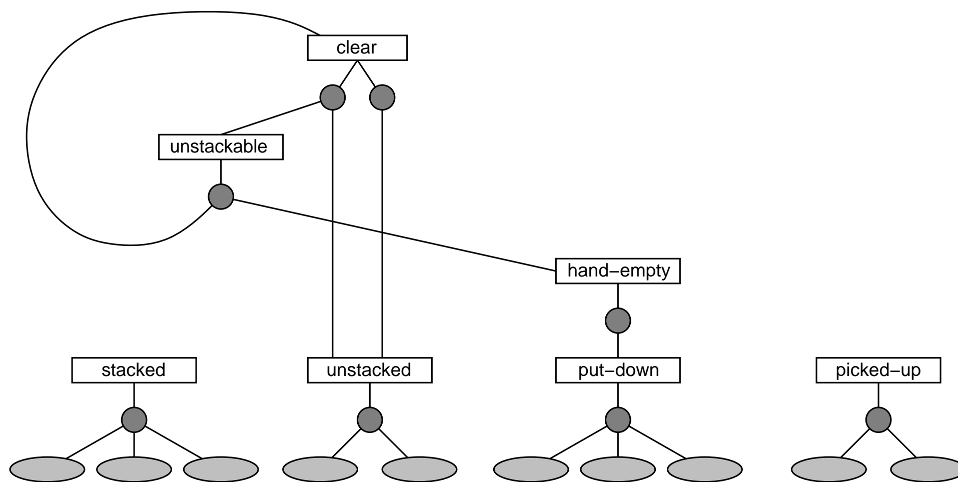

Figure 3. A graphical depiction of the hierarchy that corresponds to the skill clauses in Table 4, with each rectangle denoting a goal/concept predicate and each circle indicating a distinct clause. Primitive skills appear at the bottom, with executable actions shown as ellipses.

Each element in an `:actions` field takes the form of a functional Lisp expression. This is a list structure which starts with the name of the function that implements the action, after which come the function's arguments. An argument may be a constant number or symbol, a variable already mentioned elsewhere in the skill, or an embedded functional expression with the same syntax. Each element in a `:subgoals` field must start with a conceptual predicate that appears in the head of at least one other skill clause and that is defined in conceptual long-term memory. Each predicate argument must be a variable that corresponds to some object. These may be mentioned elsewhere in the clause, such as the `:percepts` or `:start` fields, or they may be unbound.

Table 4 shows a number of skill clauses from the Blocks World. For example, the last clause refers to two percepts, *(block ?B)* and *(block ?A)*, that must match against two distinct blocks in the perceptual buffer. It also includes two start conditions, *(on ?B ?A)* and *(hand-empty)*, which must match against elements in belief memory. This clause has no requirements for continuation, but does specify two subgoals, *(unstackable ?B ?A)* and *(unstacked ?B ?A)*, that the agent should achieve during its execution. The table also includes four primitive skill clauses at the beginning. For instance, the clause *unstacked* has two percepts, *(block ?block)* and *(block ?from)*, along with a single start condition, *(unstackable ?block ?from)*, a concept that characterizes the situation in which the agent can initiate this activity. The clause also specifies two actions, *(*grasp ?block)* and *(*vertical-move ?block)*, which it executes in the environment in that order on every invocation.

One of Icarus' important theoretical commitments is that the head of each skill clause corresponds to a concept that clause will achieve if executed to completion. This strong connection between skills and concepts may seem counterintuitive, since the same predicates appear in both long-term memories. However, it figures centrally in the architecture's performance and learning mechanisms, and should not cause problems provided one keeps in mind the distinction between concepts and skills. This assumption also imposes a hierarchical structure on the skill memory, as Figure 3 shows for the skills in Table 4, with a rectangle for each conceptual predicate and a circle for each skill clause.

Table 4. Some skills for the Blocks World domain that involve recursive calls. Each skill clause has a head that specifies the goal it achieves, a set of variable arguments, and optional sets of percepts, start conditions, and requirements. The first four (primitive) skills specify executable actions (marked by asterisks), whereas the last four (nonprimitive) clauses specify subgoals. Numbers after the heads distinguish among different clauses that achieve the same goal concept.

```
((unstacked ?from ?block)  1
 :percepts  ((block ?from) (block ?block))
 :start     ((unstackable ?from ?block))
 :actions   ((*grasp ?block) (*vertical-move ?block)))

((picked-up ?block ?from)  2
 :percepts  ((block ?block) (table ?from))
 :start     ((pickupable ?block ?from))
 :actions   ((*grasp ?block) (*vertical-move ?block)))

((stacked ?block ?to)  3
 :percepts  ((block ?block) (block ?to))
 :start     ((stackable ?block ?to))
 :actions   ((*horizontal-move ?block ?xpos) (*vertical-move ?block) (*ungrasp ?block)))

((put-down ?block ?to)  4
 :percepts  ((block ?block) (table ?to))
 :start     ((putdownable ?block ?to))
 :actions   ((*horizontal-move ?block) (*vertical-move ?block) (*ungrasp ?block)))

((clear ?B)  5
 :percepts  ((block ?C) (block ?B))
 :start     ((unstackable ?C ?B))
 :subgoals  ((unstacked ?C ?B)))

((hand-empty)  6
 :percepts  ((block ?C) (table ?T1))
 :start     ((putdownable ?C ?T1))
 :subgoals  ((put-down ?C ?T1)))

((unstackable ?B ?A)  7
 :percepts  ((block ?A) (block ?B))
 :start     ((on ?B ?A) (hand-empty))
 :subgoals  ((clear ?B) (hand-empty)))

((clear ?A)  8
 :percepts  ((block ?B) (block ?A))
 :start     ((on ?B ?A) (hand-empty))
 :subgoals  ((unstackable ?B ?A) (unstacked ?B ?A)))
```

The architecture supports two forms of skill heads. One variety consists of a conceptual predicate followed by zero or more variables that serve as its arguments. A skill clause of this form specifies a goal concept that will become satisfied once the skill has been executed to completion. All of the clauses in Table 4 are examples of this variety. The alternative form of head consists of a negated predicate followed by arguments, which indicates a goal concept that will become unsatisfied upon the skill's completion. This distinction makes sense for skill clauses, as opposed to conceptual

clauses, because skills are about carrying out action to achieve goals, which may themselves be positive or negative.

Note that two or more skill clauses may have the same positive or negated concept in their heads. For instance, the fifth clause in Table 4 refers to the same conceptual predicate, *clear*, as does the final clause already discussed. The two structures specify different ways to achieve the same goal under distinct conditions, with the second entry having one start condition, *(unstackable ?C ?B)*, and the single subgoal *(unstacked ?C ?B)*. Also note that this facility supports recursion. The first *clear* clause refers to *unstackable* as its first subgoal, but the *unstackable* clause in turn invokes *clear* as its first subgoal. The second *clear* clause serves as the base case that terminates the recursion by calling on subgoal associated with the primitive skill *unstacked*.

Icarus also supports an alternative notation for primitive skill clauses that makes their relation to Strips operators more transparent. In this scheme, the `:percepts`, `:start`, and `:requires` fields remain unchanged, but each primitive clause has a unique name, which serves as the predicate in its head. The clause also includes an `:adds` field, which indicates concepts that are made true when the skill continues to completion, and a `:deletes` field, which states concepts that are made untrue during the same process. Icarus lets users enter and display primitive skills clauses in this fashion, but it stores them internally using the other notation. In particular, it defines a concept based on those in the `:adds` and `:deletes` field, which it then uses in the clause head.

For example, Table 5 shows the operator *unstack*, an alternative version of the primitive clause *unstacked* from Table 4. These have the same start conditions and executable actions, but they differ in other respects. The Strips variant has an `:adds` field with one element, *(holding ?block)*, along with a `:deletes` field with another element, *(on ?block ?from)*. These map directly onto the positive and negative conditions, respectively, of the concept *unstacked* in Table 3, which serves as the head of the analogous clause shown in Table 4. Again, Icarus converts primitive skills in the Strips notation into its own formalism, in which the effects are encoded in the clause's head.

Because Icarus concepts and skills utilize a syntax similar to that found in logic programming languages like Prolog, we refer to the combination of these long-term memory structures as a *teleoreactive logic program* (Choi & Langley, 2005). This phrase conveys both their structural similarity to traditional logic programs and their ability to behave reactively in a goal-driven manner, following Nilsson's (1994) notion of a teleoreactive system, as we discuss below. At the same time, Icarus is also closely related to formalisms for *hierarchical task networks* (Wilkins & desJardins, 2001), especially the SHOP framework, which has a very similar syntax (Nau, Cao, Lotem, & Muñoz-Avila, 1999).

3.4 Skill Selection and Execution

Once Icarus has chosen an unsatisfied goal to achieve, the execution module selects skill clauses that it believes will take it toward this end. Because the architecture can execute directly only primitive skills, it must find a path downward from this goal to some terminal node in the skill hierarchy. A skill path is a list of instantiated skill clauses, with associated variable bindings, in

Table 5. Primitive skills for the Blocks World in a STRIPS-like syntax. Each clause has a head that specifies the skill's name and arguments, a set of typed percepts, a single start condition, a set of executable actions, an add list, and a delete list.

```
((unstacked ?block ?from)
 :percepts  ((block ?block) (block ?from))
 :start     ((unstackable ?block ?from))
 :actions   ((*grasp ?block) (*vertical-move ?block))
 :adds      ((holding ?block))
 :deletes   ((on ?block ?from)))

((picked-up ?block ?from)
 :percepts  ((block ?block) (table ?from))
 :start     ((pickupable ?block ?from))
 :actions   ((*grasp ?block) (*vertical-move ?block))
 :adds      ((holding ?block)))
 :deletes   ((ontable ?block ?from)))

((stacked ?block ?to)
 :percepts  ((block ?block) (block ?to))
 :start     ((stackable ?block ?to))
 :actions   ((*horizontal-move ?block ?xpos) (*vertical-move ?block) (*ungrasp ?block))
 :adds      ((on ?block ?to))
 :deletes   ((holding ?block)))

((put-down ?block ?to)
 :percepts  ((block ?block) (table ?to))
 :start     ((putdownable ?block ?to))
 :actions   ((*horizontal-move ?block) (*vertical-move ?block) (*ungrasp ?block))
 :adds      ((ontable ?block ?to))
 :deletes   ((holding ?block)))
```

which each clause head is a subgoal of the clause that precedes it in the path. We describe the content of these paths at more length in Section 3.5.

The execution module only considers skill paths that are *applicable* given its current beliefs, which holds if each instantiated clause along a given path is applicable. A skill clause is applicable if, for its current variable bindings, its head is not satisfied and its start conditions and requirements are satisfied by the current contents of beliefs memory. Moreover, a nonprimitive skill clause is applicable only if at least one of its subgoals is applicable. Because this test is recursive, a skill is applicable only when there is at least one acceptable path downward to a primitive skill. If the agent executed a given skill instance on the previous cycle in which it was pursuing the same top-level goal, then its start condition need not be satisfied. In such cases, the requirements test is more relevant, because a skill may take many cycles to achieve its objective, making it important to distinguish between its initiation and its continuation on successive cycles.

For example, suppose ICARUS has retrieved the instantiated skill clause *(clear A) 8* to achieve the goal *(clear A)*, given the situation depicted in Figure 1 and Table 2. In this case, one applicable path

through the skill hierarchy (from bottom to top and omitting the variable bindings) is: [*(unstacked C B) 1*, *(clear B) 5*, *(unstackable B A) 7*, *(clear A) 8*]. This holds because none of the instantiated heads (the literals along the path) are satisfied and because the instantiated start conditions of each clause (e.g., *(on B A)* and *(hand-empty)* for the topmost skill) are present in belief memory. We cannot easily denote this skill path in Figure 3 due to its recursive structure, but readers may want to trace the loops it takes through the hierarchy.

Determining whether a given skill clause is applicable relies on the same match process utilized in conceptual inference and goal satisfaction. Matching the `:percepts`, `:start`, and `:requires` fields of a skill involves comparing the generalized structures to elements in the perceptual buffer and conceptual short-term memory, as does matching the clause head. The key difference is that variables in the head are typically already bound because a skill clause usually invokes its subgoals with arguments that are mentioned elsewhere in its body. This reflects the top-down nature of Icarus' skill selection process.

Once the architecture has selected a path for execution, it replaces all variables that appear in the `:actions` field of the final (primitive) skill and evaluates each of the resulting functional expressions in order. This alters the environment, which in turn produces a different perceptual buffer on the next cycle and a different set of inferred beliefs. On each cycle, Icarus checks the skill path from the previous round to determine whether it remains applicable. If so, then it executes the primitive skill at the path's end again. If not, then it attempts to find another applicable path through the skill hierarchy that it can execute.

Returning to the example above, suppose the system selects the path that terminates in *(unstacked C B)*. Executing the actions associated with this primitive skill would alter the environment, producing a revised set of beliefs that make the path [*(put-down C T) 4*, *(hand-empty) 6*, *(unstackable C B) 7*, *(clear B) 8*, *(unstackable B A) 7*, *(clear A) 8*] acceptable. This would produce a belief state that enables the next step in the procedure, which would continue until the agent had satisfied its top-level goal, *(clear A)*. Note that this process operates much like the proof procedure in Prolog, except that it involves activities that extend over time. On each cycle, the system finds a single path through an AND tree, with later paths visiting terminal nodes (primitive skills) that occur later in the tree.

Because Icarus may encounter more than one applicable path through the skill hierarchy, it utilizes two preferences to select among the alternatives. First, given a choice between applicable skill paths, it selects the one that shares the most elements from the start of the path selected on the previous cycle. This bias encourages the agent to keep executing a high-level skill that it has started until it achieves the associated goal or becomes inapplicable. Second, given a choice between two or more subskills, it selects the first one for which the instantiated head is not satisfied. This bias supports reactive control, since the agent reconsiders previously completed subskills and, if unexpected events have undone their effects, reexecutes them to correct the situation.

These two preferences provide Icarus with a balance between persistence and reactivity. In predictable domains that involve routine behavior, such as unstacking and stacking objects in the Blocks World, the execution module will initiate a high-level skill (e.g., for building a tower) and run it to completion, invoking the necessary subskills at appropriate stages. In less predictable

domains, such as a world in which an adversary undoes goals that the agent has already achieved, the system behaves more reactively, halting a procedure in midstream and returning to earlier steps that must be repeated. This typically involves reexecuting portions of the same top-level skill, but in extreme cases it requires selecting a new top-level skill and executing that instead. Icarus is a teleoreactive architecture precisely because it can respond reactively yet in a goal-driven manner.

3.5 The Execution Stack

As we have discussed, the skill execution process is influenced by the path through the skill hierarchy that Icarus selected on the previous cycle. This information about the agent's intentions, which we call the *execution stack*, is stored with the element in goal memory that led to its selection. To include this content, elements in this memory are slightly more complex that we suggested previously. In fact, each element has two components, a `:goal` field, which specifies a top-level goal the agent wants to achieve, and an `:executed` field, which contains the most recent skill path that the architecture has executed in its pursuit of that goal.

Let us consider the structure of a skill path P in more detail. The *final* element F in this path represents the agent's intention to execute a skill clause that will achieve the top-level goal G. In many cases, the same predicate appears in F and in G, but the only requirement is that the achievement of F implies that G has been achieved. If the path P contains another element E, this must serve as a subgoal of the clause associated with F, and so forth. The initial element of P corresponds to a primitive skill clause with executable actions. Each element in the path P is one step in a chain from a high-level skill that should achieve the goal G to a primitive skill that the agent can execute. The reason that elements lower in the skill hierarchy appear earlier, and for calling the structure the 'execution stack', should become apparent when we discuss problem solving in the next section.

Each such element in the skill path P specifies the predicate in the skill clause's head, the clause's identifier, and a set of variable bindings. The bindings indicate the objects to which each variable in the skill clause was bound during the match process. These include variables that appear as arguments in the head, that represent the names of objects in the `:percepts` field, and that encode objects in the `:start` and `:requires` fields. The bindings do not include variables that denote the values of perceptual attributes, since these may well change across cycles.

Table 6 shows the contents of a execution stack that might have led to the situation in Figure 1 given the skills in Table 4. The first element in the path is *(unstacked C B) 1*, an instance of a primitive skill that involves grasping block C and lifting it off block B. The second element is *(unstackable C B) 7*, which refers to the first subskill of the *clear* clause. After this comes *(clear B) 8*, which is required to achieve *(unstackable B A)*, the penultimate goal in the stack, again the first subskill of *clear*. The final element is *(clear A) 8*, an instance of the first *clear* skill clause that would achieve the top-level goal *(clear A)*. This stack makes the recursive structure of the clauses in Table 2 even more obvious.

Icarus updates the execution stack for a given goal whenever it executes a skill path for that goal. This provides the architecture with current information about its progress in carrying out

Table 6. Contents of an Icarus execution stack for the Blocks World that could have produced the state in Figure 1 (b) and the belief memory shown in Table 2 (b), given the skills in Table 4, starting from the state in Figure 1 (a).

```
(goal :goal     (clear A)
      :executed (((unstacked C B) 1 ((?from .  B) (?block .  C)))
                 ((clear B) 8 ((?b .  C) (?a .  B)))
                 ((unstackable B A) 7 ((?a .  A) (?b .  B)))
                 ((clear A) 8 ((?b .  B) (?a .  A)))))
```

a complex task. However, recall that the system may interrupt its pursuit of a given goal if one with higher priority becomes unsatisfied. In this case, the execution stack associated with the suspended goal remains unchanged, making it available for use if that goal returns to the focus of cognitive attention later. If other activities have not altered the environment in ways that derail the high-level skill the agent was executing, the execution stack lets it continue from where it was interrupted. If such changes have occurred, Icarus uses the stack to find the appropriate path through the skill hierarchy to restart its efforts toward achieving the goal.

4. Goal Stacks and Problem Solving

In the previous section, we explained how Icarus selects goals and executes its hierarchical skills to behave reactively in an environment. This capability is sufficient when the agent has stored skills for the goals and situations it encounters, but crafting skills by hand is a tedious process that can introduce errors. For this reason, the architecture also includes a module for achieving its goals through problem solving, which involves composing known skills dynamically into plans and executing these structures. In this section, we discuss the operation of this process, starting with the memory structures that it manipulates.

4.1 Goal Stacks

We have already seen how execution stacks are stored with elements in goal memory to support Icarus' execution process. The problem-solving module takes advantage of another extension to this memory known as *goal stacks*. In fact, each element in goal memory is actually a list of elements, the last of which encodes information related to the selected top-level goal. In familiar situations, the stack never contains more than this element, since the execution module can rely entirely its long-term skills and the execution stack. However, as we will see shortly, in less routine settings the goal stack varies in depth as the problem solver deliberately adds and removes subgoals. On any cycle, the first element in the list contains information about the current subgoal, which drives behavior until the agent achieves or abandons it.

Each stack in the goal memory encodes information similar to that in the execution stack, but that includes detail required because problem solving operates in precisely those situations where

Table 7. Two Icarus' goal stacks produced at different stages of problem solving in an effort to achieve the top-level Blocks World goal *(clear A)* using the four primitive skills in Table 4.

```
(a) [(goal :type      skill
           :goal      (clear B)
           :intention ((unstacked C B) 1)
     (goal :type      concept
           :goal      (unstackable B A))
     (goal :type      skill
           :goal      (clear A)
           :intention ((unstacked B A) 1)
           :failed    (((unstacked C A) 1))]
```

```
(b) [(goal :type      skill
           :goal      (hand-empty)
           :intention ((put-down C T1) 4))
     (goal :type      concept
           :goal      (unstackable B A)
     (goal :type      skill
           :goal      (clear A)
           :intention ((unstacked B A) 1)
           :failed    (((unstacked C A) 1)))]
```

known hierarchical skills are not sufficient to achieve the agent's desires. To this end, each element in a given goal stack incorporates a number of fields, two of which we discussed earlier. The most basic is the `:goal` field, which specifies a concept instance the agent desires to achieve. When the architecture first creates a goal or subgoal, it contains only information of this sort. The second is the `:executed` field, which is empty until the system begins executing skills to achieve it, in which case it contains the most recent path taken through the skill hierarchy to this end.

As we discuss below, problem solving leads to the introduction of content into other fields. For instance, the `:type` field indicates whether the agent is attempting to achieve the goal by chaining off a skill clause or a conceptual clause. If the former, then the element also has an `:intention` field, which is either empty or contains an instantiated skill clause that, if executed, would achieve the goal. Such an element may also have an optional `:precursor` field that contains the instantiated start concept for the skill in the `:intention` field, indicating that it is not yet satisfied and must be achieved before the skill's execution. Finally, each element in a goal stack may have a `:failed` field, which includes any instantiated skill clauses it has tried and rejected (when chaining off skills) and any subgoals it has rejected (when chaining off concepts), as we will discuss shortly.

For example, Table 7 presents two goal stacks from a Blocks World task that requires achieving *(clear C)* given the leftmost state in Figure 1 and only the primitive skills in Table 2. The first goal stack occurs early in the agent's effort, when it has pushed a number of subgoals onto the stack but not yet executed any primitive skills. The second goal stack occurs near the end of the

attempt, when the problem solver has executed a number of skills and achieved certain subgoals, but has not yet reached the top-level goal. Again, note that the goal stack contains a more detailed version of the information that would be in the execution stack if stored skills had been available to achieve the top-level goal. This connection will become more apparent when we discuss learning in Section 5.

4.2 Problem Solving

ICARUS invokes its problem solver whenever it encounters an impasse, which occurs on any cycle in which the architecture cannot retrieve an applicable skill path that would achieve its current goal. This process both relies on and manipulates the goal stack we have just described. The final element on this stack describes the agent's top-level goal, whereas the first element describes its current subgoal.[3]

On each cycle, the problem solver checks to determine whether the current goal G on the goal stack has been achieved. If so, then the module pops the stack and addresses G's parent goal or, upon achieving the top-level goal, does nothing. If the current goal G is not satisfied, then the system retrieves all skill clauses with heads that match or unify with G, selects one of these candidates, and stores it in G's `:intention` field. When selecting among these alternatives, ICARUS prefers skills with satisfied start conditions but otherwise chooses a candidate at random. If the start condition C of the selected skill is satisfied, the architecture attempts to execute the skill on the next cycle. This execution may require many cycles, but eventually it produces a new environmental state that either satisfies G or constitutes another impasse. If the skill's instantiated start condition C is not satisfied, the system makes C the current goal by pushing it onto the goal stack, where it drives further processing.

If the problem solver cannot find any skill clause that would achieve the current goal G, it resorts to chaining off G's concept definition. Here it uses the definition to decompose the goal into subgoals. If more than one subgoal is unsatisfied, the system selects one at random and calls the problem solver on it recursively, which makes it the current goal by pushing it onto the stack. This leads to chaining off the start conditions of additional skills and/or the definitions of other concepts. Upon achieving a subgoal, the system pops the stack and, if other subconcepts remain unsatisfied, turns its attention to achieving them. Once all have been satisfied, this means the goal G has been achieved, so the system pops the stack again and returns to G's parent.

Of course, the problem-solving module must make decisions about which skills to select during skill chaining and the order in which it should tackle subconcepts during concept chaining. The system may well make the incorrect choice at any point, which can lead to failure on a given subgoal when no alternatives remain or when it reaches the maximum depth of the goal stack, which is a system-wide parameter. In such cases, it pops the current goal, stores the rejected candidate in the `:failed` field of its parent goal to avoid considering it in the future, and backtracks to consider other options. This strategy produces depth-first search through the problem space.

3. Readers should not to confuse the top-level goal that is the focus of attention on the current cycle with the 'current' goal here, which is the subgoal that the agent is pursuing in order to achieve that top-level goal.

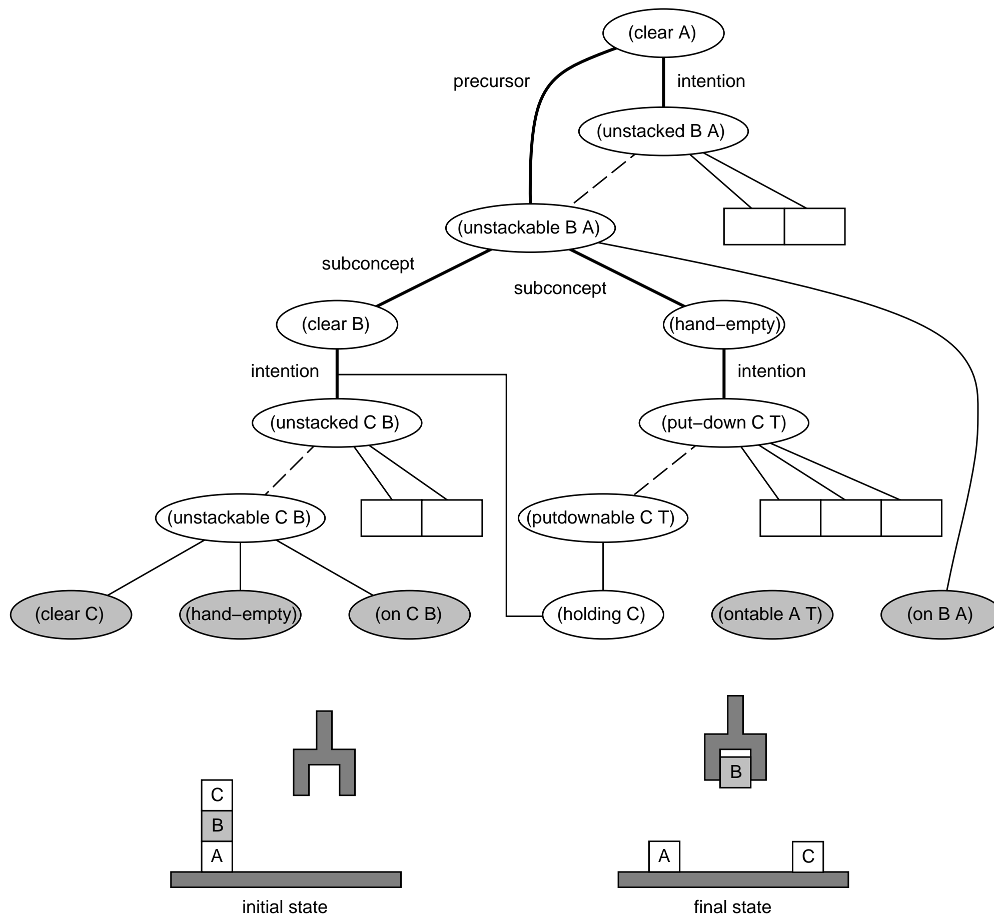

Figure 4. A trace of successful problem solving in the Blocks World, with ellipses indicating goals and rectangles denoting executable actions associated with primitive skills. Dashed lines denote the start condition of a skill clause, whereas shaded ellipses give beliefs that hold in the initial state, shown at bottom left, which the solution transforms into the final state, shown at bottom right. Thick lines indicate the structure of the learned skill hierarchy.

Figure 4 shows an example of the problem solver's behavior on the Blocks World in a situation where block A is on the table, block B is on A, block C is on B, and the hand is empty. Upon being given the goal *(clear A)*, the system looks for any executable skill with this objective as its head or, alternatively, for a skill with a head that implies the goal logically. In this case, invoking the primitive skill instance *(unstacked B A)* would produce the desired result (see Tables 3 and 4). However, this cannot yet be applied because its instantiated start condition, *(unstackable B A)*, does not hold, so the problem solver stores the skill instance with the initial goal and pushes this subgoal onto the stack.

Next, the problem solver attempts to retrieve skills that would achieve *(unstackable B A)* but, because it has no such skills in memory, it resorts to chaining off the definition of *unstackable*. This involves three instantiated subconcepts – *(clear B)*, *(on B A)*, and *(hand-empty)* – but only the first of these is unsatisfied, so the system pushes this onto the goal stack. In response, it considers skills that would produce this literal if executed successfully and retrieves the skill instance *(unstacked C B)*, which it stores with the current goal in the stack.

In this case, the start condition of the selected skill, *(unstackable C B)*, already holds, so the system executes *(unstacked C B)*, which alters the environment and causes the agent to infer *(clear B)* from its percepts. In response, it pops this goal from the stack and reconsiders its parent, *(unstackable B A)*. Unfortunately, this has not yet been achieved because executing the skill has caused the third of its component concept instances, *(hand-empty)*, to become false. Thus, the system pushes this onto the stack and, upon inspecting memory, retrieves the skill instance *(put-down C T)*, which it can and does execute.

This second step achieves the subgoal *(hand-empty)*, which in turn lets the agent infer *(unstackable B A)*. Thus, the problem solver pops this element from the goal stack and executes the skill instance it had selected originally, *(unstack B A)*, in the new situation. Upon completion, the system perceives that the altered environment satisfies the top-level goal, *(clear A)*, which means it has solved the problem. Both our textual description and the graph in Figure 4 represent the trace of successful problem solving; as mentioned earlier, finding such a solution may involve search, but we have omitted missteps that require backtracking for the sake of clarity. Also note that Icarus never holds the complete trace in goal memory at one time; rather, the goal stack contains subpaths of the graph in Figure 4 that grow and shrink across cycles.

The problem-solving strategy we have just described is a version of means-ends analysis (Newell & Simon, 1961), which was a central mechanism in the Prodigy architecture (Carbonell et al., 1990). However, the problem solver's behavior differs from the standard formulation in that it is tightly integrated with the execution process. Icarus backward chains off concept or skill definitions when necessary, but it executes the skill associated with the current stack entry as soon as it becomes applicable. Moreover, because the architecture can chain over hierarchical reactive skills, their execution may continue for many cycles before it resumes problem solving. Another difference is that Icarus' problem solver assumes both the start conditions of skills and goals are cast as single relational literals. Finally, backward chaining can occur not only off the start condition of skills but also off the definition of a concept, which means the single-literal assumption causes no loss of generality. As we will see shortly, the last two of these assumptions play key roles in the mechanism for learning new skills.

5. Learning Hierarchical Skills

Although Icarus' problem-solving module can let it overcome impasses and achieve goals for which it has no stored skills, the process can require considerable search and backtracking. The natural response is to learn from solved problems so that, when the agent encounters a similar situation in the future, no impasse will occur. In this section we describe the architecture's approach to learning from successful problem solving.

5.1 Determining Hierarchy Structure

Icarus' commitment to the hierarchical organization of skills means that the learning mechanism must determine this structure. Fortunately, the problem-solving process decomposes the original problem into subproblems, which is exactly what is needed. To take advantage of this insight,

Icarus learns a skill clause whenever it achieves a goal on the goal stack. Icarus shares this idea with earlier systems like Soar and Prodigy, although the details differ substantially.

To understand better how problem solving determines the structure of the learned skill hierarchy, examine the Blocks World trace in Figure 4, which takes the form of a semilattice in which each subplan has a single root node. This follows from the assumption that each primitive skill has one start condition and each goal is cast as a single literal. Because the means-ends problem solver chains backward off skill and concept definitions, the result is a hierarchical structure that suggests a new skill clause for each subgoal.

However, Icarus must also decide when two learned skill clauses should have the same head. The response here also draws on the problem-solving trace: the head of a learned skill clause is the goal literal achieved on the subproblem that produces it. This follows from the assumption that the head of each skill clause specifies some concept that the clause will produce if executed to completion. Such a strategy leads directly to the creation of recursive skills whenever a conceptual predicate P is a goal and P also appears as a subgoal. In this example, because *(clear A)* is the top-level goal and *(clear B)* occurs as a subgoal, one of the clauses learned for *clear* is defined recursively, although this happens indirectly through the predicate *unstackable*.

5.2 Determining Start Conditions

Of course, Icarus must also determine the start conditions on each learned clause to guard against overgeneralization during their use in skill execution. The response differs depending on whether the problem solver resolves an impasse by chaining backward on a primitive skill or on a concept definition. Let us start by examining skill clauses that the system acquires as the result of chaining backward off an existing skill.

Suppose Icarus achieves a subgoal G through skill chaining, say by first applying skill clause S_1 to satisfy the start condition for skill clause S_2 and then executing S_2, after which learning produces a clause with head G and ordered subgoals based on S_1 and S_2. In this case, the start condition for the new clause is the same as that for S_1, since when S_1 is applicable, the successful completion of this skill will ensure the start condition for S_2, which in turn will achieve G. This differs from traditional methods for constructing macro-operators (Iba, 1988) and composing production rules (Neves & Anderson, 1981), which analytically combine the preconditions of the first operator and those preconditions of later operators it does not achieve. However, either S_1 was selected because it achieves S_2's start condition or it was learned during its achievement, both of which mean that S_1's start condition is sufficient for the composed skill.[4]

In contrast, suppose the agent achieves a goal concept G through concept chaining by satisfying the subconcepts $G_1, \ldots, G_k$, in that order, while subconcepts $G_{k+1}, \ldots, G_n$ were true at the outset. In this case, Icarus constructs a new skill clause with head G and the ordered subgoals $G_1, \ldots, G_k$, each of which the system already knew and used to achieve the associated subgoal or which it

4. If the skill S_2 can be executed without invoking another skill to meet its start condition, the learning method creates a new clause G with S_2 as its only subgoal. This clause simply restates the original skill in a different form with G in its head.

Table 8. Four skill clauses for the Blocks World domain that Icarus learns from the problem-solving trace in Figure 4.

```
((clear ?B)  5
 :percepts  ((block ?C) (block ?B))
 :start     ((unstackable ?C ?B))
 :subgoals  ((unstacked ?C ?B)))
((hand-empty)  6
 :percepts  ((block ?C) (table ?T1))
 :start     ((putdownable ?C ?T1))
 :subgoals  ((put-down ?C ?T1)))
((unstackable ?B ?A)  7
 :percepts  ((block ?A) (block ?B))
 :start     ((on ?B ?A) (hand-empty))
 :subgoals  ((clear ?B) (hand-empty)))
((clear ?A)  8
 :percepts  ((block ?B) (block ?A))
 :start     ((on ?B ?A) (hand-empty))
 :subgoals  ((unstackable ?B ?A) (unstacked ?B ?A)))
```

learned from the successful solution of one of the subproblems. Under these circumstances, the start condition for the new clause is the conjunction of subgoals that were already satisfied beforehand. This prevents execution of the learned clause when some of $G_{k+1}, \ldots, G_n$ are not satisfied, in which case the sequence $G_1, \ldots, G_k$ may not achieve the goal G.

Table 8 presents the conditions selected for each of the skill clauses learned from the trace in Figure 4. The first and second clauses are trivial because they result from degenerate subproblems that the system solves by chaining off a single primitive operator. The third skill clause is more interesting because it results from chaining off the definition of the concept *unstackable*. This has the start conditions *(on ?B ?A)* and *(hand-empty)* because the subconcept instances *(on B A)* and *(hand-empty)* held at the outset.[5] The final clause is most intriguing because it results from using the third clause followed by the primitive skill instance *(unstacked B A)*. In this case, the start condition is the same as that for the third skill clause.

At first glance, the start conditions for the clause that achieves *unstackable* may appear overly general. However, recall that Icarus' execution module interprets skill clauses not in isolation but as parts of chains through the skill hierarchy. The interpreter will not select a path for execution unless all conditions along the path from the top clause to a primitive skill are satisfied. This lets the learning method store simple conditions on new clauses with less danger of overgeneralization. On reflection, this scheme makes sense for recursive control programs, since static preconditions cannot characterize such structures. Rather, one must compute appropriate preconditions dynamically, depending on the depth of recursion. The Prolog-like interpreter used for skill selection provides this flexibility and helps guard against overly general behavior.

5. Although primitive skills have only one start condition, Icarus does not place this constraint on learned clauses, thus making the acquired programs more readable but also making them unavailable for use in problem solving.

Table 9. The two goal stacks from Table 7 with additional fields (`:achieved`, `:subgoals`, and `:skills`) that ICARUS uses in constructing new skill clauses.

```
(a) [(goal :type      skill
           :goal      (clear B)
           :intention ((unstacked C B) 1))
     (goal :type      concept
           :goal      (unstackable B A)
           :achieved  ((on B A) (hand-empty)))
     (goal :type      skill
           :goal      (clear A)
           :intention ((unstacked B A) 1)
           :failed    (((unstacked C A) 1)))]
```

```
(b) [(goal :type      skill
           :goal      (hand-empty)
           :intention ((put-down C T1) 4))
     (goal :type      concept
           :goal      (unstackable B A)
           :skills    (((clear B) 5) ((hand-empty) 6) ((clear B) 5))
           :subgoals  ((clear B) (hand-empty) (clear B))
           :achieved  ((on B A) (hand-empty)))
     (goal :type      skill
           :goal      (clear A)
           :intention ((unstacked B A) 1)
           :failed    (((unstacked C A) 1)))]
```

5.3 The Extended Goal Stack

To support skill learning that results from backward chaining off a concept definition, ICARUS must retain additional details in the goal stack that it needs to construct new clauses. In particular, the system must remember which subconcepts held at the outset, since it incorporates generalized versions of them into the new skill clause's `:start` field. The problem solver stores this information in the *:achieved* field associated with goal elements of type *concept*. The system must also keep track of which subconcepts it achieved in satisfying the parent goal, as well as the order in which it arrived at them, since this determines the order of subgoals in the new clause. The architecture stores this content in the *:subgoals* field, which retains subgoals in the reverse order that the agent achieved them. Finally, the *:skills* field includes information about the specific skill clauses it used to achieve each subgoal, which lets it access the `:start` fields of these structures.

The architecture enters the initially satisfied subconcepts into the *:achieved* field when it first decides to try achieving a goal by chaining off a concept definition. This requires only that it check belief memory to determine which subconcepts are already present and then deposit them in the field. ICARUS updates the other two fields whenever it achieves a subgoal that resulted from

concept chaining. Both the subconcept and the skill clause used to achieve it are present in the subgoal being popped, so the system simply adds this information to the lists in the *:subgoals* and *:skills* fields. If achieving the subgoal required solving an unfamiliar problem, the latter will refer to the new skill clause that the module created upon its solution. Table 9 shows extended versions of the goal stacks from Table 7 that include these additional fields.

6. Using Icarus

Now that we have described Icarus' memories, representations, and processes, we can discuss how you can use the framework to construct programs for intelligent agents, load and run these programs, trace their behavior over time, and inspect the memories that result. In this section, we discuss each of these topics in turn.

6.1 Writing Icarus Programs and Defining Environments

An Icarus program consists of a set of conceptual clauses, skill clauses, goal stacks, and beliefs that take the forms described in the previous sections. You can create such a program in any text editor, provided you follow the syntax we have already covered. However, because the architecture encodes these structures in different ways, you must use different commands to specify the type of structure you are creating in each case. Icarus implements these commands as Lisp functions that take an arbitrary numbers of arguments with starting and closing parentheses. They include:

- *create-goals* and *cg*, both of which store their arguments in goal memory, creating a one-element goal stack for each;
- *create-belief* and *cb*, which store their arguments in belief memory as static beliefs that do not change across cycles;
- *create-concepts* and *cc*, both of which store in concept memory the conceptual clauses provided as arguments;
- *create-skills* and *cs*, each of which stores in skill memory the skill clauses given as arguments;
- *create-operators* and *co*, which accept as arguments primitive skills stated in Strips syntax that are transformed and stored in skill memory.

Each function examines the syntax of its arguments for obvious problems. The *create-concepts* and *create-belief* commands also check to ensure that no defined concepts appear as predicates in static beliefs, which the architecture does not allow.

Appendix A shows the contents of a file that contains a simple Icarus program for the Blocks World. Note that each command starts with a left parenthesis and ends with a right parenthesis, as do all Lisp functions, but that the arguments should not be preceded by a quotation mark, as Lisp typically requires. The file may spread statements about individual memory elements across multiple lines because they too are delimited by parentheses. Finally, note that the file includes comment lines that document the program, each beginning with a semi-colon so the Lisp interpreter and compiler knows to ignore them.

Icarus also supports a number of commands for removing structures from a given memory, each of which accepts a list of predicate names as arguments. These include:

- *(remove-concepts)* and *(rc)*, which remove all conceptual clauses from concept memory;
- *(remove-skills)* and *(rs)*, which deletes all skill clauses from skill memory;
- *(remove-goals)* and *(rg)*, which eliminate all elements from goal memory;
- *(remove-beliefs)* and *(rb)*, which remove all static beliefs from belief memory.

You may also call these functions with one or more predicate names as arguments, as in *(remove-concepts on clear)*. For conceptual and skill memory, this deletes all clauses with heads that contain those predicates, for belief memory it removes all elements that begin with those predicates, and for goal memory it eliminates elements that have the predicates in their `:goal` field. In addition, you may give *remove-beliefs* specific beliefs as arguments, which will remove them from memory.

Recall that Icarus is designed to support the construction of physical agents, which requires that they operate in some environment distinct from the agents themselves. You must provide such an environment for a given Icarus program, along with an interface that specifies what the agent will perceive and what effects are produced by its actions. You can define these most easily as Lisp functions that are stored in a separate file, although you can also write programs in other languages like C and access them through the Lisp "foreign function" facility.

You can decide to simulate the environment using any data structures and mechanisms that you like, but you must define a function called *update-world* that, when evaluated, returns a set of percepts in Icarus syntax that describes the current state of the environment. The environment file must also define one function for each action that your program may call in its primitive skills. These may take any number of numeric or symbolic arguments, but they should also alter the environment in some manner.[6] On each cycle, the Icarus interpreter calls on the *update-world* function when updating the perceptual buffer and evaluates each action function that appears at the top of the selected skill path. Appendix B shows the contents of an environment file for the Blocks World that satisfies these constraints and that can be used with the Icarus program in Appendix A.

6.2 Installing Icarus, Loading Files, and Running Programs

Because Icarus is implemented in the programming language Lisp (McCarthy et al., 1962; Norvig, 1992), you will need a running version of Lisp on your computer before you can install the architecture and run programs written in it. We have tested the current Icarus code in CLISP (http://clisp.cons.org/) and CMUCL (http://www.cons.org/cmucl/), both of which are in the public domain and which you can download from the World Wide Web. Instructions for installing and running these dialects of Lisp are available at their Web sites. Once you have installed a version of Lisp, you should download the Icarus source from http://icaria.dhcp.asu.edu/cogsys/icarus.html and place it in a directory or folder on your computer. After starting Lisp from that directory, you should enter the command *(compile-file "icarus.lisp")*, which will compile the architecture file and store the result in the directory.

6. By convention, the names of action functions begin with an asterisk to ease readability, but this is not required.

After you have written an Icarus program and specified the environment in which it will operate, you are ready to load and run the program. The various commands and function definitions will have no effect until Lisp processes them. To this end, you should start Lisp on your computer and type *(load "icarus")*, which will load the architecture files you compiled earlier. You should then load your environment code followed by the Icarus program. For example, if you have specified the former in the file *blockenv.lisp* and the latter in *block.lisp*, you would type *(load "blockenv.lisp")* followed by *(load "block.lisp")*.

If the first command produces errors, you may have problems in the Lisp code that defines the environment and you should examine the code or consult a Lisp manual. If the second command generates errors, they indicate that the Icarus compiler has detected problems with your syntax with the concepts, skills, or goals. In either case, you should exit Lisp, correct the problems in a text editor, and try loading the files again.

Once you have loaded the environment and Icarus files without error messages, you are ready to run the program. The architecture provides two commands that produce slightly different effects:

- *(run N)* causes Icarus to run the program for *N* cycles, even if it takes no actions in the environment. Calling the *run* function with no arguments instead runs the program indefinitely, until the user enters an interrupt (control-C in most dialects of Lisp).
- *(grun N)* causes Icarus to run the program for *N* cycles or until it satisfies all top-level goals in the goal memory, whichever comes first. Calling *grun* with no arguments instead runs the program indefinitely until it achieves all top-level goals.

The architecture also lets you continue running a program after it has halted. For this purpose, you can use either *(cont N)* or *(gcont N)*, which restart the program from the current state of memory but otherwise have analogous meaning to the *run* and *grun* commands, respectively.

In some cases, you may want to run a program with some of the architecture's components disabled, say for the purpose of demonstration or experimental comparison. Icarus lets you accomplish this effect for the problem-solving module and the skill-learning mechanism. To achieve this end, you should type *(switches solving off learning off)* or include this command in one of the files you load. When you need to reactivate these processes, you should type *(switches solving on learning on)*, which will cause the Icarus interpreter to invoke them in the manner we described in Sections 4 and 5. Of course, you can also turn each module off and on in isolation by providing only its name to the *switches* command. You can also use this function to alter the maximum depth of goal stack the problem solver will construct before backtracking, with *(switches stack-depth 6)* producing the default setting.

6.3 Tracing Behavior and Inspecting Memory

Icarus also provides you with trace information about the course of a program run. Appendix C presents a sample run of the Icarus program from Appendix A. The default trace prints out considerable detail about events on each cycle, including the cycle number and the contents of the perceptual buffer, belief memory, and goal memory. The latter includes the goal stack for each

element in the memory and the execution stack for each element in the goal stack that has led to skill execution. If the Icarus program takes action on a given cycle, then the trace also shows the instantiated actions in their order of application. When the architecture invokes problem solving, it displays alternatives for backward chaining off skills and concepts, along with the candidate selected. Finally, if learning occurs on a cycle, the trace includes the skill clauses that Icarus constructed.

Such detailed traces are very useful for debugging Icarus programs, but they are seldom required once the agent is behaving as desired. Upon reaching this stage, you may want more cursory traces of system behavior. To this end, Icarus includes a variety of commands for turning off particular facets of the trace or, alternatively, for turning them back on when desired. These include:

- *(ctrace off)*, which omits the cycle number from the trace, and *(ctrace on)*, which includes it;
- *(ptrace off)*, which keeps the perceptual buffer from being displayed, and *(ptrace on)*, which presents its contents;
- *(btrace off)*, which makes the trace hide the contents of belief memory, and *(btrace on)*, which displays them;
- *(gtrace off)*, which eliminates the contents of goal memory from the trace, and *(gtrace off)*, which shows them, including the elements in each goal stack,
- *(etrace off)*, which makes the trace omit the execution stack for each element in each goal stack, and *(etrace off)*, which shows them, provided the goal memory is displayed.
- *(atrace off)*, which does not show actions executed on the current cycle, and *(atrace on)*, which prints them in their order of invocation;
- *(mtrace off)*, which hides information about skill and concept clauses considered during problem solving, and *(mtrace on)*, which presents them, along with the one selected; and
- *(ltrace off)*, which leaves out details about learning, and *(ltrace on)*, which shows new skills and clauses created on the current cycle.

Appendix C also shows a reduced trace for the Blocks World program that Icarus produces when some of these parameters are turned off. You can avoid any trace at all by calling each of the commands with the *off* argument either in Lisp or in one of the files you load. Alternatively, you can use the command *(alltrace off)* to turn off all trace information or, conversely, use *(alltrace on)* to turn on the most detailed level.

Once you have completed a run, you may want to inspect the state of short-term memories, especially if you did not utilize a detailed trace. For this purpose, you can use the commands:

- *(print-percepts)* or *(pp)*, which print the contents of the perceptual buffer;
- *(print-beliefs)* or *(pb)*, which display the contents of belief memory;
- *(print-goals)* or *(pg)*, which show the contents of goal memory, including the goal stacks; and
- *(print-goal-paths)* or *(pgp)*, which print the contents of goal memory, including the execution stack for each element in the goal stack.

You may also want to examine the contents of long-term memory, especially if learning has occurred, but also to check that newly loaded concepts and skills have been stored correctly. To this end, you can use the functions:

- *(print-concepts)* or *(pc)*, which print the contents of conceptual memory in their order of their storage;
- *(print-skills)* or *(ps)*, which display the contents of skill memory, also in their order of their storage; and
- *(print-operators)* or *(po)*, which shows all primitive skills using the Strips notation, with add and delete lists.

In addition, each function may be called with one or more predicate names as arguments, as in *(print-beliefs on clear)*. Taken together, these inspection and trace commands should provide enough information about Icarus operation and its cognitive state for most purposes, ranging from the details needed for debugging to the blank slate desirable for automated experimentation.

7. Concluding Remarks

In summary, Icarus is a cognitive architecture that is designed to support the construction of intelligent agents. Like other architectures, it encodes information using list structures, relies on pattern matching to access relevant content, operates in cycles, and makes theoretical commitments about the representation, use, and acquisition of knowledge. The framework includes a conceptual inference process that generates short-term beliefs from concept definitions and percepts. Icarus also incorporates a module that selects top-level goals and one that finds applicable paths through a skill hierarchy for execution in the environment. Finally, it includes mechanisms for solving novel problems through means-ends analysis and for caching generalized versions of solutions into new skill clauses.

Icarus also comes with a high-level programming language that has a syntax for concepts, skills, beliefs, goals, and intentions, along with an interpreter for running programs stated in this syntax. The language includes commands that let users load, run, and trace the operation of programs, as well as inspect the contents of various memories. Icarus attempts to incorporate the best ideas from previous research on cognitive architectures while introducing new structures and mechanisms to provide new capabilities. The framework will undoubtedly evolve over time, but its current version embodies a set of important claims about the nature of intelligence.

Acknowledgements

Development of Icarus was funded in part by Grant HR0011-04-1-0008 from DARPA IPTO and by Grant IIS-0335353 from the National Science Foundation. Discussions with Nima Asgharbeygi, Kirstin Cummings, Glenn Iba, Negin Nejati, David Nicholas, Seth Rogers, Stephanie Sage, and Dan Shapiro contributed to the ideas we have incorporated into the architecture and its associated programming language.

References

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

Carbonell, J. G., Knoblock, C. A., & Minton, S. (1990). Prodigy: An integrated architecture for planning and learning. In K. Van Lehn (Ed.), *Architectures for intelligence*. Hillsdale, NJ: Lawrence Erlbaum.

Clocksin, W. F., & Mellish, C. S. (1981). *Programming in* Prolog. Berlin: Springer-Verlag.

Fikes, R., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, *3*, 251–288.

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

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1–64.

Langley, P., & Rogers, S. (2005). An extended theory of human problem solving. *Proceedings of the Twenty-seventh Annual Meeting of the Cognitive Science Society*. Stresa, Italy.

McCarthy, J., Abrahams, P. W., Edwards, J., Hart, T. P., & Levin, M. I. (1962). *LISP 1.5 programmer's manual*. Cambridge, MA: MIT Press.

Nau, D., Cao, Y., Lotem, A., Muñoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (pp. 968–973). Stockholm: Morgan Kaufmann.

Neches, R., Langley, P., & Klahr, D. (1987). Learning, development, and production systems. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development.* Cambridge, MA: MIT Press.

Neves, D. M., & Anderson, J. R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, NJ: Lawrence Erlbaum.

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

Newell, A., & Shaw, F. C. (1957). Programming the Logic Theory Machine. *Proceedings of the Western Joint Computer Conference* (pp. 230–240).

Newell, A., & Simon, H. A. (1961). GPS, A program that simulates human thought. In H. Billing (Ed.), *Lernende automaten*. Munich: Oldenbourg KG. Reprinted in E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill, 1963.

Newell, A., & Tonge, F. M. (1960). An introduction to Information Processing Language V. *Communications of the ACM*, *3*, 205–211.

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

Norvig, P. (1992). *Paradigms of artificial intelligence programming: Case studies in Common Lisp*. San Francisco: Morgan Kaufmann.

Wilkins, D. E., & desJardins, M. (2001). A call for knowledge-based planning. *AI Magazine*, *22*, 99–115

Appendix A. A Sample Icarus Program

```
; BLOCK.LISP
; Icarus concepts, primitive skills, and goals for the Blocks World.

; Create a set of conceptual clauses for the Blocks World.
(create-concepts

; ON describes a situation in which one block is on top of another.
((on ?block1 ?block2)
 :percepts  ((block ?block1 xpos ?xpos1 ypos ?ypos1)
             (block ?block2 xpos ?xpos2 ypos ?ypos2 height ?height2))
 :tests     ((equal ?xpos1 ?xpos2) (>= ?ypos1 ?ypos2) (<= ?ypos1 (+ ?ypos2 ?height2))))

; ONTABLE describes a situation in which a block in sitting on the table.
((ontable ?block1 ?block2)
 :percepts  ((block ?block1 xpos ?xpos1 ypos ?ypos1)
             (table ?block2 xpos ?xpos2 ypos ?ypos2 height ?height2))
 :tests     ((>= ?ypos1 ?ypos2) (<= ?ypos1 (+ ?ypos2 ?height2))))

; CLEAR matches when a block has no other block on top of it.
((clear ?block)
 :percepts  ((block ?block))
 :relations ((not (on ?other-block ?block))))

; HOLDING is satisfied when the hand is holding a block.
((holding ?block)
 :percepts  ((hand ?hand status ?block) (block ?block)))

; HAND-EMPTY describes a situation in which the hand is not holding a block.
((hand-empty)
 :percepts  ((hand ?hand status ?status))
 :relations ((not (holding ?any)))
 :tests     ((eq ?status 'empty)))

; THREE-TOWER matches when there exists a tower composed of three blocks.
((three-tower ?x ?y ?z ?table)
 :percepts  ((block ?x) (block ?y) (block ?z) (table ?table))
 :relations ((on ?x ?y) (on ?y ?z) (ontable ?z ?table)))

; UNSTACKABLE is satisfied when it is possible to unstack one block from another.
((unstackable ?block ?from)
 :percepts  ((block ?block) (block ?from))
 :relations ((on ?block ?from) (clear ?block) (hand-empty)))

; PICKUPABLE matches when it is possible to pick a block up from the table.
((pickupable ?block ?from)
 :percepts  ((block ?block) (table ?from))
 :relations ((ontable ?block ?from) (clear ?block) (hand-empty)))
```

```
; STACKABLE describes a situation in which a block can be stacked on another.
((stackable ?block ?to)
 :percepts  ((block ?block) (block ?to))
 :relations ((clear ?to) (holding ?block)))

; PUTDOWNABLE is satisfied when it is possible to put a held block on the table.
((putdownable ?block ?to)
 :percepts  ((block ?block) (table ?to))
 :relations ((holding ?block)))

; UNSTACKED matches when the agent is holding one block that is not on another.
((unstacked ?from ?block)
 :percepts  ((block ?from) (block ?block))
 :relations ((holding ?block) (not (on ?block ?from))))

; PICKED-UP matches when the agent is holding one block that is not on the table.
((picked-up ?block ?from)
 :percepts  ((block ?block) (table ?from))
 :relations ((holding ?block) (not (ontable ?block ?from))))

; STACKED matches when the agent is not holding one block that is on another.
((stacked ?block ?to)
 :percepts  ((block ?block) (block ?to))
 :relations ((on ?block ?to) (not (holding ?block))))

; PUT-DOWN matches when the agent is not holding a block that is on the table.
((put-down ?block ?to)
 :percepts  ((block ?block) (table ?to))
 :relations ((ontable ?block ?to) (not (holding ?block))))
)

; Create a set of primitive skill clauses for the Blocks World.
(create-skills

; Achieve UNSTACKED by grasping and moving an unstackable block.
((unstacked ?from ?block)
 :percepts  ((block ?from) (block ?block))
 :start     ((unstackable ?block ?from))
 :actions   ((*grasp ?block) (*vertical-move ?block)))

; Achieve PICKED-UP by grasping and moving a pickuable block.
((picked-up ?block ?from)
 :percepts  ((block ?block) (table ?from))
 :start     ((pickupable ?block ?from))
 :actions   ((*grasp ?block) (*vertical-move ?block)))
```

```
; Achieve STACKED by moving and ungrasping a stackable block.
((stacked ?block ?to)
 :percepts  ((block ?block) (block ?to))
 :start     ((stackable ?block ?to))
 :actions   ((*horizontal-move ?block ?xpos) (*vertical-move ?block) (*ungrasp ?block)))

; Achieve PUT-DOWN by moving and ungrasping a putdownable block.
((put-down ?block ?to)
 :percepts  ((block ?block) (table ?to))
 :start     ((putdownable ?block ?to))
 :actions   ((*horizontal-move ?block) (*vertical-move ?block) (*ungrasp ?block)))
)

; Create the single goal of making block A clear.
(create-goals (clear A))
```

Appendix B. A Sample Environment File

```
; BLOCKENV.LISP
; Perceptions and actions for a simulated Blocks World environment.

; Create a data structure that holds the state of the environment.
(setq initial*
      (list '(block A xpos 10 ypos 2 width 2 height 2)
            '(block B xpos 10 ypos 4 width 2 height 2)
            '(block C xpos 10 ypos 6 width 2 height 2)
            '(block D xpos 10 ypos 8 width 2 height 2)
            '(table T1 xpos 20 ypos 0 width 20 height 2)
            '(hand H1 status empty)))

; Define a function that Icarus will call to initialize the environment.
(defun initialize-world ()
  (setq state* (rcopy initial*))
  nil)

; Define a function that Icarus will call to update the environment.
(defun update-world () nil)

; Define a function that Icarus will call to update the perceptual buffer.
(defun preattend () state*)

; Define an action for grasping a block.
(defun *grasp (block)
  (let* ((object (assoc 'hand state*))
         (rest (member 'status object)))
    (cond ((not (null atrace*))
           (terpri)(princ "Grasping ")(princ block)))
    (setf (cadr rest) block)))
```

```
; Define an action for ungrasping a block.
(defun *ungrasp (block)
  (let* ((object (assoc 'hand state*))
         (rest (member 'status object)))
    (cond ((not (null atrace*))
           (terpri)(princ "Ungrasping ")(princ block)))
    (setf (cadr rest) 'empty)))

; Define an action for moving a block vertically.
(defun *vertical-move (block ypos)
  (let* ((object (sassoc block state*))
         (rest (member 'ypos object)))
    (cond ((not (null atrace*))
           (terpri)(princ "Moving ")(princ block)
           (princ " to vertical position ")(princ ypos)))
    (setf (cadr rest) ypos)))

; Define an action for moving a block horizontally.
(defun *horizontal-move (block xpos)
  (let* ((object (sassoc block state*))
         (rest (member 'xpos object)))
    (cond ((not (null atrace*))
           (terpri)(princ "Moving ")(princ block)
           (princ " to horizontal position ")(princ xpos)))
    (setf (cadr rest) xpos)))
```

Appendix C. A Sample Icarus Run

```
> lisp
* (load "icarus")
; Loading #p"/home/langley/code/icarus/icarus.l".
T
* (load "block")
; Loading #p"/home/langley/code/icarus/recur/block.l".
T
* (switch depth-limit 4)
4
* (grun 5)
----------
Cycle 1
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (CLEAR A))]
Perceptual buffer:
((BLOCK A XPOS 10 YPOS 2 WIDTH 2 HEIGHT 2)
 (BLOCK B XPOS 10 YPOS 4 WIDTH 2 HEIGHT 2)
 (BLOCK C XPOS 10 YPOS 6 WIDTH 2 HEIGHT 2)
 (TABLE T1 XPOS 20 YPOS 0 WIDTH 20 HEIGHT 2) (HAND H1 STATUS EMPTY))
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Choices for skill chaining:
((UNSTACKED 1) C A)
((UNSTACKED 1) B A)
Selecting ((UNSTACKED 1) C A)
----------
Cycle 2
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) C A))]
Perceptual buffer:
((BLOCK A XPOS 10 YPOS 2 WIDTH 2 HEIGHT 2)
 (BLOCK B XPOS 10 YPOS 4 WIDTH 2 HEIGHT 2)
 (BLOCK C XPOS 10 YPOS 6 WIDTH 2 HEIGHT 2)
 (TABLE T1 XPOS 20 YPOS 0 WIDTH 20 HEIGHT 2) (HAND H1 STATUS EMPTY))
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
----------
Cycle 3
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (UNSTACKABLE C A))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) C A))]
Perceptual buffer:
```

```
((BLOCK A XPOS 10 YPOS 2 WIDTH 2 HEIGHT 2)
 (BLOCK B XPOS 10 YPOS 4 WIDTH 2 HEIGHT 2)
 (BLOCK C XPOS 10 YPOS 6 WIDTH 2 HEIGHT 2)
 (TABLE T1 XPOS 20 YPOS 0 WIDTH 20 HEIGHT 2) (HAND H1 STATUS EMPTY))
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Choices for concept chaining:
(ON C A)
Selecting (ON C A)
----------
Cycle 4
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (ON C A))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE C A)
       :ACHIEVED  ((CLEAR C) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) C A))]
Perceptual buffer:
((BLOCK A XPOS 10 YPOS 2 WIDTH 2 HEIGHT 2)
 (BLOCK B XPOS 10 YPOS 4 WIDTH 2 HEIGHT 2)
 (BLOCK C XPOS 10 YPOS 6 WIDTH 2 HEIGHT 2)
 (TABLE T1 XPOS 20 YPOS 0 WIDTH 20 HEIGHT 2) (HAND H1 STATUS EMPTY))
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Choices for skill chaining:
((STACKED 3) C A)
Selecting ((STACKED 3) C A)
----------
Cycle 5
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (ON C A)
       :INTENTION ((STACKED 3) C A))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE C A)
       :ACHIEVED  ((CLEAR C) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) C A))]
Perceptual buffer:
((BLOCK A XPOS 10 YPOS 2 WIDTH 2 HEIGHT 2)
 (BLOCK B XPOS 10 YPOS 4 WIDTH 2 HEIGHT 2)
 (BLOCK C XPOS 10 YPOS 6 WIDTH 2 HEIGHT 2)
 (TABLE T1 XPOS 20 YPOS 0 WIDTH 20 HEIGHT 2) (HAND H1 STATUS EMPTY))
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
```

```
NIL
* (ptrace off)
T
* (gcont 50)
----------
Cycle 6
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (STACKABLE C A))
 (GOAL :TYPE      SKILL
       :GOAL      (ON C A)
       :INTENTION ((STACKED 3) C A))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE C A)
       :ACHIEVED  ((CLEAR C) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) C A))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
-Backtracking because depth limit exceeded.
----------
Cycle 7
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (ON C A)
       :INTENTION FAILED
       :FAILED    (((STACKED 3) C A)))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE C A)
       :ACHIEVED  ((CLEAR C) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) C A))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
----------
Cycle 8
Goal memory:
[(GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE C A)
       :ACHIEVED  ((CLEAR C) (HAND-EMPTY))
       :FAILED    ((ON C A)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) C A))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
----------
```

```
Cycle 9
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION FAILED
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Choices for skill chaining:
((UNSTACKED 1) B A)
Selecting ((UNSTACKED 1) B A)
----------
Cycle 10
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
----------
Cycle 11
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (UNSTACKABLE B A))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Choices for concept chaining:
(CLEAR B)
Selecting (CLEAR B)
----------
Cycle 12
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (CLEAR B))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
```

```
Choices for skill chaining:
((UNSTACKED 1) C B)
((UNSTACKED 1) A B)
Selecting ((UNSTACKED 1) C B)
----------
Cycle 13
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR B)
       :INTENTION ((UNSTACKED 1) C B))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Executing: (*GRASP 'C)
           (*VERTICAL-MOVE 'C)
Skill stack: ((((UNSTACKED 1) C B)
               ((UNSTACKED 1 C B) (?YPOS . 6) (?FROM . B) (?BLOCK . C)))
              (((UNSTACKED 1) B A))
              (((STACKED 3) C A))
              (((UNSTACKED 1) C A)))
Grasping C
Moving C to vertical position 16
----------
Cycle 14
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR B)
       :INTENTION ((UNSTACKED 1) C B))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
Storing new skill clause:
CLEAR (?B)  id: 5
 :percepts ((BLOCK ?C) (BLOCK ?B))
 :start    ((UNSTACKABLE ?C ?B))
 :subgoals ((UNSTACKED ?C ?B))
----------
Cycle 15
```

```
Goal memory:
[(GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((CLEAR 5) B))
       :SUBGOALS  ((CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
Choices for concept chaining:
(HAND-EMPTY)
----------
Cycle 16
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (HAND-EMPTY))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((CLEAR 5) B))
       :SUBGOALS  ((CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
Choices for skill chaining:
((PUT-DOWN 4) C T1)
((PUT-DOWN 4) B T1)
((PUT-DOWN 4) A T1)
((STACKED 3) C B)
((STACKED 3) C A)
((STACKED 3) B C)
((STACKED 3) B A)
((STACKED 3) A C)
((STACKED 3) A B)
Selecting ((STACKED 3) C B)
----------
Cycle 17
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (HAND-EMPTY)
       :INTENTION ((STACKED 3) C B))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((CLEAR 5) B))
```

```
       :SUBGOALS  ((CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
Executing: (*HORIZONTAL-MOVE 'C)
           (*VERTICAL-MOVE 'C)
           (*UNGRASP 'C)
Skill stack: ((((STACKED 3) C B)
               ((STACKED 3 C B) (?HEIGHT . 2) (?YPOS . 4) (?XPOS . 10) (?TO . B)
                (?BLOCK . C)))
              (((UNSTACKED 1) B A))
              (((STACKED 3) C A))
              (((UNSTACKED 1) C A)))
Moving C to horizontal position 10
Moving C to vertical position 6
Ungrasping C
----------
Cycle 18
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (HAND-EMPTY)
       :INTENTION ((STACKED 3) C B))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((CLEAR 5) B))
       :SUBGOALS  ((CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Storing new skill clause:
HAND-EMPTY NIL  id: 6
 :percepts ((BLOCK ?C) (BLOCK ?B))
 :start    ((STACKABLE ?C ?B))
 :subgoals ((STACKED ?C ?B))
----------
Cycle 19
Goal memory:
[(GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((HAND-EMPTY 6)) ((CLEAR 5) B))
       :SUBGOALS  ((HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
```

```
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Choices for concept chaining:
(CLEAR B)
----------
Cycle 20
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (CLEAR B))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((HAND-EMPTY 6)) ((CLEAR 5) B))
       :SUBGOALS  ((HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Choices for skill chaining:
((UNSTACKED 1) C B)
((UNSTACKED 1) A B)
Selecting ((UNSTACKED 1) C B)
----------
Cycle 21
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR B)
       :INTENTION ((UNSTACKED 1) C B))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((HAND-EMPTY 6)) ((CLEAR 5) B))
       :SUBGOALS  ((HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Executing: (*GRASP 'C)
           (*VERTICAL-MOVE 'C)
Skill stack: ((((UNSTACKED 1) C B)
               ((UNSTACKED 1 C B) (?YPOS . 6) (?FROM . B) (?BLOCK . C)))
```

```
              (((UNSTACKED 1) B A))
              (((STACKED 3) C A))
             (((UNSTACKED 1) C A)))
Grasping C
Moving C to vertical position 16
----------
Cycle 22
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR B)
       :INTENTION ((UNSTACKED 1) C B))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((HAND-EMPTY 6)) ((CLEAR 5) B))
       :SUBGOALS  ((HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
New clause would be equivalent to CLEAR 5
----------
Cycle 23
Goal memory:
[(GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((CLEAR 5) B) ((HAND-EMPTY 6)) ((CLEAR 5) B))
       :SUBGOALS  ((CLEAR B) (HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
Choices for concept chaining:
(HAND-EMPTY)
----------
Cycle 24
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (HAND-EMPTY))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((CLEAR 5) B) ((HAND-EMPTY 6)) ((CLEAR 5) B))
       :SUBGOALS  ((CLEAR B) (HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
```

```
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
Choices for skill chaining:
((PUT-DOWN 4) C T1)
((PUT-DOWN 4) B T1)
((PUT-DOWN 4) A T1)
((STACKED 3) C B)
((STACKED 3) C A)
((STACKED 3) B C)
((STACKED 3) B A)
((STACKED 3) A C)
((STACKED 3) A B)
Selecting ((PUT-DOWN 4) C T1)
----------
Cycle 25
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (HAND-EMPTY)
       :INTENTION ((PUT-DOWN 4) C T1))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((CLEAR 5) B) ((HAND-EMPTY 6)) ((CLEAR 5) B))
       :SUBGOALS  ((CLEAR B) (HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
Executing: (*HORIZONTAL-MOVE 'C)
           (*VERTICAL-MOVE 'C)
           (*UNGRASP 'C)
Skill stack: ((((PUT-DOWN 4) C T1)
               ((PUT-DOWN 4 C T1) (?HEIGHT . 2) (?YPOS . 0) (?XPOS . 20)
                (?TO . T1) (?BLOCK . C)))
              (((UNSTACKED 1) B A))
             (((STACKED 3) C A))
             (((UNSTACKED 1) C A)))
Moving C to horizontal position 105
Moving C to vertical position 2
Ungrasping C
----------
Cycle 26
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (HAND-EMPTY)
```

```
       :INTENTION ((PUT-DOWN 4) C T1))
 (GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((CLEAR 5) B) ((HAND-EMPTY 6)) ((CLEAR 5) B))
       :SUBGOALS  ((CLEAR B) (HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PICKUPABLE C T1) (UNSTACKABLE B A) (HAND-EMPTY) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ONTABLE C T1) (ON B A))
Storing new skill clause:
HAND-EMPTY NIL  id: 7
 :percepts ((BLOCK ?C) (TABLE ?T1))
 :start    ((PUTDOWNABLE ?C ?T1))
 :subgoals ((PUT-DOWN ?C ?T1))
----------
Cycle 27
Goal memory:
[(GOAL :TYPE      CONCEPT
       :GOAL      (UNSTACKABLE B A)
       :SKILLS    (((HAND-EMPTY 7)) ((CLEAR 5) B) ((HAND-EMPTY 6))
                   ((CLEAR 5) B))
       :SUBGOALS  ((HAND-EMPTY) (CLEAR B) (HAND-EMPTY) (CLEAR B))
       :ACHIEVED  ((ON B A) (HAND-EMPTY)))
 (GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PICKUPABLE C T1) (UNSTACKABLE B A) (HAND-EMPTY) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ONTABLE C T1) (ON B A))
Storing new skill clause:
UNSTACKABLE (?B ?A)  id: 8
 :percepts ((BLOCK ?A) (BLOCK ?B))
 :start    ((ON ?B ?A) (HAND-EMPTY))
 :subgoals ((CLEAR ?B) (HAND-EMPTY))
----------
Cycle 28
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :PRECURSOR ((UNSTACKABLE 8) B A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PICKUPABLE C T1) (UNSTACKABLE B A) (HAND-EMPTY) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ONTABLE C T1) (ON B A))
Executing: (*GRASP 'B)
           (*VERTICAL-MOVE 'B)
```

```
Skill stack: ((((UNSTACKED 1) B A)
               ((UNSTACKED 1 B A) (?YPOS . 4) (?FROM . A) (?BLOCK . B)))
              (((STACKED 3) C A))
              (((UNSTACKED 1) C A)))
Grasping B
Moving B to vertical position 14
----------
Cycle 29
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :PRECURSOR ((UNSTACKABLE 8) B A)
       :INTENTION ((UNSTACKED 1) B A)
       :FAILED    (((UNSTACKED 1) C A)))]
Belief memory:
((PUTDOWNABLE B T1) (STACKABLE B C) (STACKABLE B A) (HOLDING B) (CLEAR A)
 (CLEAR B) (CLEAR C) (ONTABLE A T1) (ONTABLE C T1))
Achieved goal on cycle 29.
Storing new skill clause:
CLEAR (?A)  id: 9
 :percepts ((BLOCK ?B) (BLOCK ?A))
 :start    ((ON ?B ?A) (HAND-EMPTY))
 :subgoals ((UNSTACKABLE ?B ?A) (UNSTACKED ?B ?A))
SUCCESS
* (initialize-world)
NIL
* (clear-goals)
NIL
* (create-goals (clear A))
[(GOAL :TYPE      NIL
       :GOAL      (CLEAR A))]
* (grun 20)
----------
Cycle 1
Goal memory:
[(GOAL :TYPE      NIL
       :GOAL      (CLEAR A))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Retrieving skill (CLEAR A)
Selecting ((CLEAR 9) A)
----------
Cycle 2
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((CLEAR 9) A))]
Belief memory:
((UNSTACKABLE C B) (THREE-TOWER C B A T1) (HAND-EMPTY) (CLEAR C) (ONTABLE A T1)
 (ON B A) (ON C B))
Executing: (*GRASP 'C)
```

```
          (*VERTICAL-MOVE 'C)
Skill stack: ((((CLEAR 9) A) ((CLEAR 9 A) (?B . B) (?A . A))
               ((UNSTACKABLE 8 B A) (?A . A) (?B . B))
               ((CLEAR 9 B) (?B . C) (?A . B))
               ((UNSTACKED 1 C B) (?YPOS . 6) (?FROM . B) (?BLOCK . C))))
Grasping C
Moving C to vertical position 16
----------
Cycle 3
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((CLEAR 9) A))]
Belief memory:
((PUTDOWNABLE C T1) (STACKABLE C B) (HOLDING C) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ON B A))
Executing: (*HORIZONTAL-MOVE 'C)
          (*VERTICAL-MOVE 'C)
          (*UNGRASP 'C)
Skill stack: ((((CLEAR 9) A) ((CLEAR 9 A) (?B . B) (?A . A))
               ((UNSTACKABLE 8 B A) (?A . A) (?B . B))
               ((HAND-EMPTY 7) (?T1 . T1) (?C . C))
               ((PUT-DOWN 4 C T1) (?HEIGHT . 2) (?YPOS . 0) (?XPOS . 20)
                (?TO . T1) (?BLOCK . C))))
Moving C to horizontal position 34
Moving C to vertical position 2
Ungrasping C
----------
Cycle 4
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((CLEAR 9) A))]
Belief memory:
((PICKUPABLE C T1) (UNSTACKABLE B A) (HAND-EMPTY) (CLEAR B) (CLEAR C)
 (ONTABLE A T1) (ONTABLE C T1) (ON B A))
Executing: (*GRASP 'B)
          (*VERTICAL-MOVE 'B)
Skill stack: ((((CLEAR 9) A) ((CLEAR 9 A) (?B . B) (?A . A))
               ((UNSTACKED 1 B A) (?YPOS . 4) (?FROM . A) (?BLOCK . B))))
Grasping B
Moving B to vertical position 14
----------
Cycle 5
Goal memory:
[(GOAL :TYPE      SKILL
       :GOAL      (CLEAR A)
       :INTENTION ((CLEAR 9) A))]
Belief memory:
((PUTDOWNABLE B T1) (STACKABLE B C) (STACKABLE B A) (HOLDING B) (CLEAR A)
 (CLEAR B) (CLEAR C) (ONTABLE A T1) (ONTABLE C T1))
Achieved goal on cycle 5.
```

```
SUCCESS
* (ps)
((UNSTACKABLE ?B ?A)  id: 8
 :percepts ((BLOCK ?A) (BLOCK ?B))
 :start    ((ON ?B ?A) (HAND-EMPTY))
 :ordered  ((CLEAR ?B) (HAND-EMPTY)))

((HAND-EMPTY)  id: 7
 :percepts ((BLOCK ?C) (TABLE ?T1))
 :start    ((PUTDOWNABLE ?C ?T1))
 :ordered  ((ON-TABLE-HAND-EMPTY ?C ?T1)))

((HAND-EMPTY)  id: 6
 :percepts ((BLOCK ?C) (BLOCK ?B))
 :start    ((STACKABLE ?C ?B))
 :ordered  ((STACKED ?C ?B)))

((CLEAR ?A)  id: 9
 :percepts ((BLOCK ?B) (BLOCK ?A))
 :start    ((ON ?B ?A) (HAND-EMPTY))
 :ordered  ((UNSTACKABLE ?B ?A) (UNSTACKED ?B ?A)))

((CLEAR ?B)  id: 5
 :percepts ((BLOCK ?C) (BLOCK ?B))
 :start    ((UNSTACKABLE ?C ?B))
 :ordered  ((UNSTACKED ?C ?B)))

((UNSTACKED ?BLOCK ?FROM)  id: 1
 :percepts ((BLOCK ?BLOCK YPOS ?YPOS) (BLOCK ?FROM))
 :start    ((UNSTACKABLE ?BLOCK ?FROM))
 :actions  ((*GRASP ?BLOCK) (*VERTICAL-MOVE ?BLOCK)))

((PICKED-UP ?BLOCK ?FROM)  id: 2
 :percepts ((BLOCK ?BLOCK) (TABLE ?FROM HEIGHT ?HEIGHT))
 :start    ((PICKUPABLE ?BLOCK ?FROM))
 :actions  ((*GRASP ?BLOCK) (*VERTICAL-MOVE ?BLOCK)))

((STACKED ?BLOCK ?TO)  id: 3
 :percepts ((BLOCK ?BLOCK) (BLOCK ?TO XPOS ?XPOS YPOS ?YPOS HEIGHT ?HEIGHT))
 :start    ((STACKABLE ?BLOCK ?TO))
 :actions  ((*HORIZONTAL-MOVE ?BLOCK ?XPOS)
            (*VERTICAL-MOVE ?BLOCK)))

((ON-TABLE-HAND-EMPTY ?BLOCK ?TO)  id: 4
 :percepts ((BLOCK ?BLOCK) (TABLE ?TO XPOS ?XPOS YPOS ?YPOS HEIGHT ?HEIGHT))
 :start    ((PUTDOWNABLE ?BLOCK ?TO))
 :actions  ((*HORIZONTAL-MOVE ?BLOCK)
            (*VERTICAL-MOVE ?BLOCK)))
```