ICARUS User's Manual

Pat Langley (langley@isle.org)
Dongkyu Choi (dongkyuc@gmail.com)

Institute for the Study of Learning and Expertise
2164 Staunton Court, Palo Alto, 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 4
February 22, 2011

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

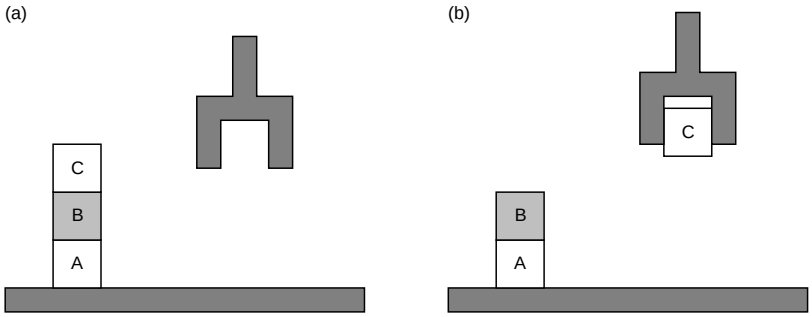

Figure 1. Two states from the Blocks World domain used to illustrate ICARUS' representations and processes.

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))))
```

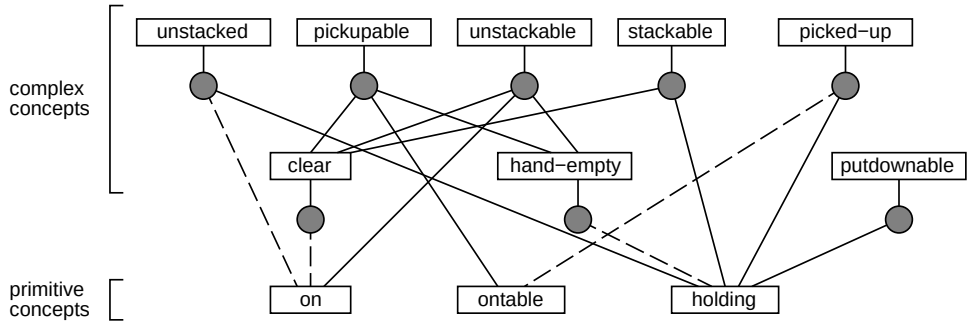

Figure 2. A graphical depiction of the hierarchy that corresponds to the conceptual clauses in Table 3, with rectangles denoting conceptual predicates and circles indicating distinct clauses. Solid lines represent positive conditions, whereas dashed lines stand for negated ones. Primitive concepts appear 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. Skills, Intentions, and Skill 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 store skills the agent can execute to carry out complex activities and intentions about which skills to invoke. These are linked by performance mechanisms that execute the associated skills, thus changing the environment and, hopefully, taking the agent closer to its objectives.

3.1 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 name and arguments, 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 `:conditions` field, which states the concepts that must match to initiate the clause, and an `:effects` field, which specifies relations that hold on its successful execution.

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 `:conditions` 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.[2] The `:effects` field contains a set of relational literals, with any variables mentioned in either the conditions or percepts being constrained to match the same objects. Both conditions and effects may include negated literals, but their meanings differ. A negated condition denotes a relational pattern that must not match against any elements in belief memory, whereas a negated effect indicates that belief memory contains no such elements on the skill's completion.

Icarus distinguishes between primitive and nonprimitive skill clauses. Primitive clauses include in their bodies an `:action` field, which specifies an action that the agent can execute directly in the environment. Primitive skills 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 `:subskills` field, which refers to other skills that the agent should carry out to implement the current clause.

2. Percepts and conditions may include unbound variables not mentioned in the head, which have the same meaning as in conceptual clauses.

Table 4. Primitive skills for the Blocks World domain. Each skill clause has a head that specifies its name and arguments, an optional set of percepts, conditions for application, and effects of application. Each skill specifies an executable action (marked by an asterisk).

```
((unstack ?block ?from)
 :percepts   ((block ?block) (block ?from))
 :conditions ((on ?block ?from) (not (on ?other ?block))
              (not (holding ?any)))
 :action     (*grasp-and-lift ?block)
 :effects    ((not (on ?block ?from)) (holding ?block)))

((pick-up ?block ?from)
 :percepts   ((block ?block) (table ?from))
 :conditions ((not (on ?other ?block)) (ontable ?block ?from)
              (not (holding ?any)))
 :action     (*grasp-and-lift ?block)
 :effects    ((not (ontable ?block ?from)) (holding ?block)))

((stack ?block ?to)
 :percepts   ((block ?block) (block ?to))
 :conditions ((holding ?block) (not (on ?other ?to)))
 :action     (*place-and-ungrasp ?block ?to)
 :effects    ((on ?block ?to) (not (holding ?block))))

((put-down ?block ?to)
 :percepts   ((block ?block) (table ?to))
 :conditions ((holding ?block))
 :action     (*place-and-ungrasp ?block ?to)
 :effects    ((ontable ?block ?to) (not (holding ?block))))
```

An element in an `:action` 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 `:subskills` field must start with a conceptual predicate that appears in the head of at least one other skill clause. 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 `:conditions` fields, or they may be unbound.

Table 4 presents four primitive skills for the blocks world, each of which describes the conditional effects of different actions. These include:

- *unstack*, whose action **grasp-and-lift* takes a block *?block* as its argument. This structure is applicable only when *?block* is on another block, when there is no other block on it, and when the hand is not holding anything. The two effects are that *?block* is no longer on the other block and the hand is holding it.
- *pickup*, which has the same action, **grasp-and-lift*. This skill matches only when *?block* is on the table, when there is no other block on it, and when the hand is empty. The effects are that *?block* is no longer on the table and the hand is holding it.

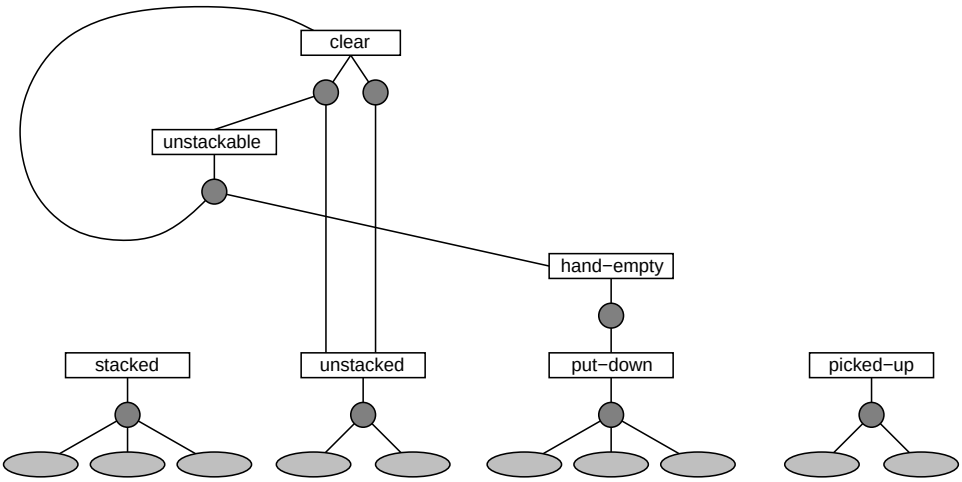

Figure 3. A graphical depiction of the hierarchy that corresponds to the skill clauses in Table 4, with each rectangle denoting a goal/concept predicate and each circle indicating a distinct clause. Primitive skills appear at the bottom, with executable actions shown as ellipses.

- *stack*, whose action **ungrasp-and-place* takes three arguments, the block *?block*, a x position, and a y position. This clause can apply when the hand is holding *?block* and another block, *?to*, with nothing on it resides at location *?x* and *y*. Upon execution, the hand is no longer holding *?block* and it is on *?to*, the other block.
- *putdown*, which has the same action, **ungrasp-and-place*. This skill is applicable whenever the hand is holding *?block*, and its effects are that the hand is no longer holding *?block* and this object is on the table.

These four skill clauses are directly analogous to the operators specified in traditional formulations of the blocks world. However, the latter typically include predicates like *clear* and *hand-empty* to avoid dealing with negated conditions. Also, traditional treatments do not ground their conceptual predicates in percepts, since they deal only with abstract planning rather than embodied execution.

These examples should clarify Icarus' syntax for skills. For instance, the head of the first clause is *(unstack ?block ?from)*, with *unstack* being the skill name and with *block* and *?from* being its arguments. The conditions for this clause, associated with the `:conditions` field, are *(block ?block)*, *(block ?from)*, *(on ?block ?from)*, *(not (on ?any ?block))*, and *(not (holding ?any2))*. Each one refers to a conceptual predicate, which must be defined in concept memory, and a set of variableized arguments. The `:effects` field specifies a set of effects, in this case *(not (on ?block ?from))* and *(holding ?b)*, that take the same form as conditions. However, rather than describing situations in which the clause is applicable, they describe changes that result from executing it.

Table 5 presents two nonprimitive skill clauses that describe a more complex activity in the Blocks World. Here the clauses for *make-clear* specify how to alter the world so that *?block*, their single argument, has no other blocks on it. Rather than having an `:action` field, these structures include a `:subskills` field that refers to other skills the agent should invoke, in the given order, when pursuing an activity. Each subskill specifies the name of a skill defined elsewhere in memory,

Table 5. Nonprimitive skill clauses that describe a complex procedure clearing objects in the Blocks World. Each clause has a head that specifies its name and arguments, an optional set of percepts, application conditions, and effects of application. Each skill also specifies a ordered set of subskills, which may include recursive calls.

```
((make-clear ?block)
 :percepts   ((block ?block) (block ?on))
 :conditions ((on ?on ?block) (not (on ?other ?on))
              (not (holding ?any)))
 :subskills  ((unstack ?on ?block) (put-down ?on ?table))
 :effects    ((not (on ?other ?block)) (not (on ?on ?block))))
((make-clear ?block)
 :percepts   ((block ?block) (block ?on))
 :conditions ((on ?on ?block) (on ?other ?on)
              (not (holding ?any)))
 :subskills  ((make-clear ?on) (unstack ?on ?block)
              (put-down ?on ?table))
 :effects    ((not (on ?other ?block)) (not (on ?on ?block))))
```

along with a set of variableized arguments. Again, these variables may also appear in the head, conditions, effects, or other subskills, which indicates they must bind to the same object.

The first clause handles situations in which a second block, *?on*, is on *block* but has no other blocks on top of it. This structure refers to two subskills, *(unstack ?on ?block)* and *(put-down ?on ?table)*, which we just encountered in Table 4. Their combined effects under these conditions are to remove *?on* from *?block* and to make the hand empty. The second skill clause for *make-clear* deals with more complicated situations in which, in addition to the block *?on* being on *block*, another block is sitting atop *?on*. Because the third object may, in turn, have something on it, the clause calls on *make-clear* recursively as its first subskill, followed by *unstack* and *put-down*. In this case, the `:effects` field specifies only some of the changes that result, since the recursive invocation of *make-clear* means that an aribtrary number of blocks may unstacked and placed on the table.

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) for specifying hierarchical, conditional procedures.

3.2 Intentions and Intention Memory

Although skills and their supporting concepts provide ICARUS with generalized long-term knowledge about complex activities, they are not sufficient to let the architecture carry out those activities.

Table 6. Four intentions generated by Icarus while making a block clear in the Blocks World. Each intention has a head and an ID, as well as a set of variable bindings, conditions, effects, and either subskills or an action. All but the top-level intention also include a :parent field.

```
((make-clear A)  ID: I1
 :bindings   ((?other .  C) (?on .  B) (?block .  A))
 :conditions ((block A) (block B) (on B A) (on C B) (not (holding ?any)))
 :subskills  ((make-clear B) (unstack B A) (put-down B ?table))
 :effects    ((not (on C A)) (not (on B A)))

((make-clear B)  ID: I2  Parent: I1
 :bindings   ((?on .  C) (?block .  B))
 :conditions ((block B) (block C) (on C B) (not (on ?other C))
              (not (holding ?any)))
 :subskills  ((unstack C B) (put-down C ?table))
 :effects    ((not (on ?other B)) (not (on C B)))

((unstack C B)  ID: I3  Parent: I2
 :bindings   ((?from .  B) (?block .  C))
 :conditions ((on C B) (not (on ?other C))
              (not (holding ?any)))
 :action     (*grasp-and-lift C)
 :effects    ((not (on C B)) (holding C))

((put-down C ?TO)  ID: I2  Parent: I1
 :bindings   ((?block .  C))
 :conditions ((holding C))
 action:     (*place-and-ungrasp C T1)
 :effects    ((ontable C ?to) (not (holding C)))
```

For this, it also requires more specific, short-term structures about particular instances of skills. The contents of Icarus' belief memory, which contains instances of general concepts, satisfies part of this need, in that it provides the elements against which skill conditions and effects match. However, the system also needs some means to specify which activity it intends to pursue out of the many possible candidates.

For this reason, Icarus also includes a memory for *intentions*. Each element in this short-term store is an instance of some skill clause, typically with specific objects as arguments of the head, although these may retain one or more variables in that role. An intention describes the agent's commitment to execute a particular skill in a particular way. For nonprimitive skill clauses, this means indicating that the ordered subskills should expand to the next lower level. For example, the intention *I1* in Table 6 is an instance of the second skill clause in Table 5, which states that, in order to *(make-clear A)*, the agent will first execute the subskill *(make-clear B)* followed by *(unstack B A)* and *(put-down B ?table)*. The intention also includes details about variable bindings, which may be incomplete. In this case, the final subskill is only partly instantiated, because binding information was unavailable at the time of intention creation.

Just as Icarus skills may have subskills, so intentions may have subintentions. However, because

the architecture may only focus on one intention at a time, it suffices to store information about an intention's immediate parent, rather than the the reverse. We will see that, as ICARUS descends and ascends through its skill hierarchy, it generates subintentions as necessary, creates an intention 'stack' that it encodes only implicitly through these parent links.

3.3 Basic Mechanisms for Skill Execution

Now that we have discussed how ICARUS represents and organizes skills and their associated intentions, we can describe how it uses them to carry out complex activities. We will assume, for now, that the architecture begins with a statement or instruction about the skill it plans to pursue, although in the next chapter we will consider other ways to drive the execution process. This tells the system at what point in the hierarchy to begin, but skill execution is a complex cognitive process that deserves examination in some detail.

As explained earlier, the first step on each cycle involves conceptual inference, a process that we covered in the previous chapter. Thus, given the scenario in Table 1 (a), the system generates three beliefs on the initial cycle – *(on C B)*, *(on B A)*, and *(ontable A T1)* from its percepts, in addition to statements about *A*, *B*, and *C* being blocks and *T1* being a table. Based on the top-level call *(make-clear A)*, ICARUS retrieves the two skill clauses from Table 5, which have the same predicate as the specified skill. Recall that these provide different ways to execute the skill that are appropriate to distinct situations. In this case, the system binds *?block* to *A* in the first clause and *?block* to *A* in the second, leaving the remaining variables unbound.

The next step involves determining which, if any, of these skill clauses is applicable. ICARUS considers a clause applicable if its conditions are satisfied and if its effects are not satisfied (if so, there is no reason to execute it). Thus, the architecture attempts to match the conditions of each clause against elements in belief memory, subject to partial bindings obtained from the head, using the same techniques discussed in the previous chapter for matching conceptual clauses. If a clause's conditions do not match against belief memory, or if its effects do match, the execution module drops it from consideration. In our example, both skill clauses have the same expected effect, *(not (on ?any A))*, that is unstatisfied. When any candidate clauses remain (in this case the first, recursive clause in the table), the system determines if their partially instantiated conditions match against the current beliefs.[3] Once this process is complete, the architecture has zero, one, or more instantiated clauses that are relevant to the current situation, from which it selects at random.

Once the architecture has selected an instantiated skill clause to execute, it creates an intention, *I1*, to that effect. This includes details about the clause identifier, bindings, subskills, conditions, and effects, with the bindings substituted into the last three components to at least partly instantiate them. In our blocks world example, ICARUS produces the intention *I1* with the head *(make-clear A)*. Table 6 presents the details of this intention, which includes the bindings for variables in its associated skill clause, as well as the instantiated conditions, subskills, and effects. This structure

3. Actually, a given clause may match in more than one way, giving different instantiations, each with distinct variable bindings.

becomes the controlling intention at the end of cycle 1, which lets it influence system behavior on successive rounds.

When the current intention is based on a clause that is nonprimitive, as in this case, the architecture accesses the first element in the `:subskills` field, uses the current bindings to instantiate this skill, and attempts to find a way to execute it. In this situation, the first (instantiated) subskill is *(make-clear B)*. Thus, on cycle 2, Icarus again retrieves all skill clauses with corresponding heads, uses the bound arguments to partially instantiate their conditions and effects, and determines which, if any, of these instantiated clauses is applicable. In our example, the only alternative is the first clause in Table 5, which the system uses to create a new intention, *I2*. As Table 6 clarifies, this structure has nearly the same head as the first intention, *(make-clear B)*, but its body corresponds to the base case rather than the recursive one.

On the next cycle, Icarus considers this new intention's instantiated subskills. The first is *(unstack C B)*, for which there is only one relevant structure, the first skill clause in Table 4. This is applicable, which leads it to generate a third intention, *I3*, to execute an instantiated version of this clause. On cycle 4, the architecture uses the bindings obtained from the head and conditions to instantiate the contents of its `:action` field and, because the clause that produced the intention is primitive, invokes them in the environment. This alters the agent's surroundings, as shown in Figure1 (b), which leads to new inferences on the next cognitive cycle.

In this instance, the situation has changed in two ways, with the belief *(on B C)* becoming false and the belief *(holding C)* becoming true. The execution module does not assume, just because it has applied a primitive intention, that its effects will be achieved on the next cycle, but it does check for this possibility. If the new beliefs match all of the effects, as in this case, then Icarus treats the intention *I3* as having completed successfully. However, the effects of its parent intention, *I2*, remain unsatisfied and more work remains. The architecture considers the next subskill, *(put-down B ?table)*, and on cycle 5 creates a corresponding intention, *I4*, based on the fourth clause in Table 4. This includes binding *?table* to *T1*, since *(table T1)* is the only belief that matches the clause's second condition.

The skill clause associated with this intention is again primitive and its conditions are satisfied. Accordingly, on cycle 6, the system executes the instantiated action *(*put-down B T1)* in the environment. This creates the new situation shown in Figure 3-2 (c). Another round of inference on cycle 7 reveals that *(holding C)* has become false and *(ontable C)* has become true. The new beliefs match the expected effects of the intention, which tells the system that *I4* has completed successfully. In response, on cycle 8 it marks *I4* as done and returns attention to its parent, *I2*, which has the head *(make-clear B)*.

However, Icarus notes that the effects associated with this intention are also satisfied, leading it to focus on the next subskill, *(unstack B A)*. As before, there is only one skill clause with a relevant head, in this case the first entry in Table 4, so on cycle 9 Icarus creates a new intention, *I5*, based on this structure. This has the instantiated conditions *(on B A)*, *(not (on ?any B))*, and *(not (holding ?any2))*, all of which are matched by the agent's current beliefs. Thus, on cycle 9 the system executes the instantiated action associated with the intention, *(*grasp-and-lift B)*, which produces a new situation.

Inference on the next cognitive cycle reveals that the intention's effects, *(not (on ?any A))* and *(holding B)*, are satisfied, leading ICARUS to mark *I5* as completed and return attention to its parent, *I1: (make-clear A)*. But this intention has one remaining subskill to execute, *(put-down B ?table)*, so on cycle 10 the system creates another intention, *I6*, to carry this out. Again, only one skill clause (the fourth in Table 3-1) is relevant, and matching its conditions against the current beliefs produces the instantiated versions *(block B)*, *(table T1)*, and *(holding T1)*, making the intention applicable. As a result, on cycle 11 ICARUS executes the associated action *(*put-down B T1)*, generating the environmental state in Figure 3-2 (e).

The remaining activities revolve around mental clean up. On cycle 12, the architecture confirms that the expected effects of intention *I6* are satisfied, so it marks this as completed and returns attention to *I1: (make-clear A)*. But on the ensuing cycle, the system notes that its effects also hold, so it marks this intention as complete as well. Because this was the top-level intention, ICARUS halts and awaits another command, having completed the task that it had been assigned.

3.4 Mechanisms for Execution Failure and Recovery

The mechanisms we have just described suffice for situations in which ICARUS' execution of hierarchical skills goes according to plan, but it requires additional processes to handle cases in which failures arise. Such failures can occur for a number of distinct reasons. One possibility is that the conditions of some skills are overly general, which can lead the system to select an incorrect clause that it cannot execute to completion. Another is that the effects of some skills are overly general, causing the architecture to think it has completed too soon, which can raise problems for its successers.

A third scenario involves overly specific conditions, which can keep ICARUS from considering a skill clause that is actually applicable. A fourth case concerns overly specific effects, which can lead the system to think that a skill has not terminated when it has already done so. Of course, an incorrect set of subskills can also cause difficulty, leading the agent to attempt subtasks for which there are no applicable clauses, or to omit subtasks that are key to completing the parent intention.

However, knowledge errors of this sort are not the only source of execution failures. The effects of actions may be probabilistic, with only a certain chance of having the desired outcome. Alternatively, the environment may contain other agents or forces that can alter the situation and thus interrupt the expected trajectory of states. There may even be cases in which different bindings for a single skill clause require the agent to guess, sometimes incorrectly, about which intention to pursue. Each can lead the architecture to begin executing a hierarchical skill only to find later that it has no available options.

Now that we have characterized the sources of execution failure in ICARUS, we can turn to how it deals with them. The initial step involves detecting that a failure has occurred. The architecture relies on three complementary mechanisms for this purpose. First, the system may generate an intention I based on an apparently relevant skill but then be unable to find an applicable intention for its first subskill. Second, ICARUS may have successfully executed some of intention I's subskills, but then find no way to carry out the next subskill in the list.

In fact, both of these can happen on the same task, with the former failure type occurring at one level, which in turn leads to the latter type one level above. A third detection mechanism, akin to the second, utilizes an expected duration parameter associated with each skill clause that indicates the number of cycles that clause should take to complete. When the architecture has pursued an intention for N times the expected duration,[4] it decides that its efforts have gone on too long and abandons the intention.

Once Icarus has detected an execution failure, it attempts to respond to the problem. The basic mechanism here involves abandoning the current intention I that the sustem cannot initiate or complete, returning attention to the parent intention P that invoked it to carry out subskill S, and storing its failure with P to prevent its consideration on future cycles. Once Icarus has done this, it attempts to recover from the failure by considering other ways to carry out the subskill S of the parent intention.

If the architecture can retrieve another skill instance that could enact skill S, then it creates an intention based on this instance and makes it the current focus of attention. If all proceeds well, then this alternative by passes the failure and lets the agent continue toward completing the top-level skill. If this choice also leads to failure, then Icarus abandons this option as well, marks it as problematic, and consider other candidates. This continues until the agent runs out of choices it has not already tried or until it exceeds the number of cycles allocated to complete the parent intention. In such cases, the system abandons the invoking skill S, which in turn leads to failure for the intention calling it. A special case arises when S is the top-level skill, in which case Icarus abandons the task entirely and redirects its efforts elsewhere or idles until another task arrives.

4. Goals, Problems, and Problem Solving

In the previous section, we explained how Icarus executes its hierarchical skills to behave reactively in an environment. This capability is sufficient when the agent has stored skills for the tasks 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 Representations for Problem Solving

We have already discussed the architecture's ability to encode beliefs as logical literals that describe entities and relations among them. The beliefs in memory on a given cognitive cycle denote inferences the agent has made based on observations of the environment. Thus, a set of beliefs provides a natural representation for the current problem state, which changes over time as the agent attempts to solve a problem. Also, recall that beliefs are always instances of some generic concept stored in conceptual memory, which ensures that the agent's problem state is always grounded on,

4. The factor N here is a global Icarus parameter set to 2.0 by default.

or at least connected to, its perceptions. We have also discussed ICARUS' use of intentions to represent the activities it aims to carry out in efforts to change the problem state. These intentions in turn are instances of generic skills that describe the effects of certain activities when applied under specified conditions. Thus, skills play the same role as operators in AI planning systems, since they are responsible for transforming one state into another. We will see that ICARUS' problem-solving mechanisms do not distinguish between primitive skills, which refer to executable actions, and nonprimitive ones, which refer to a sequence of subskills.

In summary, problem solving rests on the same representational constructs that ICARUS uses for conceptual inference and skill execution. However, it requires two new types of structure that these more basic modules do not assume. One is the notion of a *goal*, which describes some aspect of a situation that the agent considers desirable. Goals take a form similar to beliefs, except that they may contain pattern-match variables for some arguments and they may be negated. For example, possible goals for the blocks world include *(on A B)*, which denotes that the agent wants block B to support block B, *(on ?x b)*, which states that block B should support any other block, and *(not (on ?x B)*, which indicates that block B should not support any other block. The predicates in goals, like those in beliefs, always refer to some defined concept, and we say that a goal is *satisfied* if there exists some belief (concept instance) that matches (or does not, for negated goals) against the pattern.

However, goals often travel together, indicating that the agent desires to achieve a state that satisfies an entire set of goals jointly. We will refer to such a set of conjoined goals as a *problem*. The goal elements of a problem may share pattern-match variables that must bind consistently. When all of its goals are satisfied in this manner, we say that the problem has been *solved*. For example, the problem {*(on ?x ?y) (on ?y ?z)*} matches consistently against the pair of beliefs *(on A B)* and *(on B C)* but not against the pair *(on A B)* and *(on C D)*. Thus, the former situation constitutes a problem solution but the latter does not. Also, problems are typically more abstract than belief states in that they contain fewer literals, so that different belief states may solve a given problem in the same way.

Problem solving involves transforming the current state into one that solves the problem, which in turn depends on how they differ. This step is not as straightforward as it seems, since there may be different ways to characterize the difference. For instance, consider the problem {*(on ?w ?x) (on ?x ?y) (on ?y ?z)*} and the belief state {*(on A B) (on B C) (ontable C)*}. There exist two ways[5] that this state partially matches against the problem statement. If *?w* binds to A, *?x* binds to B, and *?y* binds to C, then the goals *(on ?w ?x)* and *(on ?x ?y)* are satisfied but *(on ?y ?z)* is not. On the other hand, if *?x* binds to A, *?y* binds to B, and *?z* binds to C, then the goals *(on ?x ?y)* and *(on ?y ?z)* are satisfied but *(on ?w ?x)* is not. We will refer to each such set of variable mappings as *bindings* and to the associated unmatched goal elements as *differences*. Naturally, the selection one makes in such cases determines which goals the agent must achieve to solve the problem, which justifies the need for cognitive structures that store these choices.

5. To be more precise, there are two maximally specific partial matches which subsume others that have fewer elements.

Table 7. Some problems and intentions from a Blocks World task. Each problem specifies a set of goals, bindings, and differences between the goals and beliefs. Bindings and differences can change during the course of problem solving. Each intention has a head, bindings, conditions, and effects, in addition to other content not shown. An intention always has a problem as its parent; a problem has an intention as its parent unless it occurs at the top level.

```
(problem :id P1
 :goals       ((not (on ?other A)) (ontable A) (not (holding ?any)))
 :bindings    ( )
 :differences ((not (on B A))))
(intention :id I1  :parent P1
 :head        (unstack B A)
 :bindings    ((?from .  A) (?on .  B))
 :conditions  ((on B A) (not (on ?other B)) (not (holding ?any)))
 :action      (*grasp-and-lift B)
 :effects     ((not (on B A)) (holding B))
(problem :id P2    :parent I1
 :goals       ((on B A) (not (on ?other B)) (not (holding ?any)))
 :bindings    ((?from .  A) (?on .  B))
 :differences ((not (on C B))))
(intention :id I2  :parent P1
 :head        (unstack C B)
 :bindings    ((?from .  B) (?on .  C))
 :conditions  ((on C B) (not (on ?other C)) (not (holding ?any)))
 :action      (*grasp-and-lift C)
 :effects     ((not (on C B)) (holding C))
```

Of course, representing goals, problems, and differences is only part the story; the architecture must also encode how the agent attempts to achieve them. Fortunately, Icarus already has a representation for intentions that we can use for this purpose. Typically, an intention for a given problem should either achieve one or more of the unsatisfied goals as one of its expected effects or its conditions will match the current belief state. The architecture stores this and other problem-related content in four attributes – a `:goals` field that specifies the problem elements, a `:bindings` field that maps problem variables onto constants, an associated `:differences` field that notes which problem goals are unsatisfied, and an `:intention` field that contains the skill instance the agent intends to execute. We will see shortly that a problem's bindings, differences, and intention change incrementally during the process of solving it.

Some problems are relatively easy in that the intentions which achieve goals are immediately applicable in the current environment, but the problem solver is not always so fortunate. In many cases, one or more of an intention's conditions are unsatisfied. In these situations, Icarus may specify a new problem that involves achieving these conditions, which it also stores in memory. This subproblem will have its own `:goals` (the intention's conditions), `:bindings`, `:differences`, and `:intention` with distinct contents, along with a `:parent` field that points to its parent problem. This problem may in turn point to another subproblem and so forth. In this way, Icarus can specify

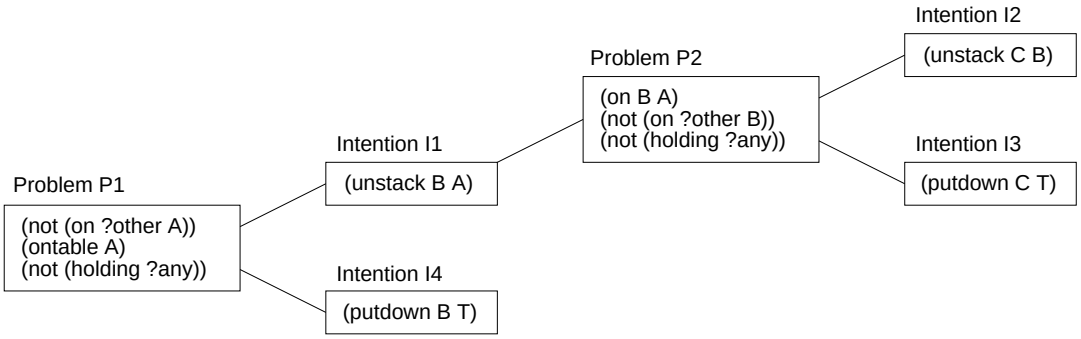

Figure 4. Problems and intentions selected in a successful solution to a Blocks World task. Problem and intention numbers indicate the order in which ICARUS generated them. The figure does not show the bindings and differences for problems from Table 7 because these change during the course of problem solving.

a chain of problems and subproblems relevant to the top-level task, with the chain terminating when an intention has satisfied conditions that make it applicable in the current belief (problem) state.

Table 7 presents one such chain that could arise during problem solving in the blocks world given the belief state *(on C B)*, *(on B A)*, and *(ontable A)*. The top-level problem, *P1*, involves three goal elements, *(not (on ?other A))*, *(ontable A)*, and *(not (holding ?any)*. In this case, there is only one set of bindings and associated differences. The bindings set is empty because the only positive goal has no variables to bind and the only difference is *(not (on B A))*, since the belief *(on B A)* keeps the first goal from being satisfied. In response, the architecture has chosen the intention *I1*, which has the instantiated head *(unstack B A)*. This in turn has led it to create the subproblem *P2*, which has as its goal elements the instantiated conditions of the intention, *(on B A)*, *(not (on ?other B))*, and *(not (holding ?any))* and *(not (on ?x C))*. This time the bindings are {*(?from . A) (?on . B)*} and again there is a single difference, *(not (on C B))*. In this case the problem solver has selected the intention *(unstack C B)*, which should eliminate the difference. The conditions for this skill instance are satisfied, so there is no need for additional recursion.

Figure 4 depicts a solution to the overall task that shows the problem goals and intentions selected during the course of problem solving. The diagram shows the top-level problem on the far left, the two intentions associated with it to the right, the subproblem created for the first intention to its right, and the two intentions that led to solving this subproblem. The tree structure of the figure reflects the manner in which ICARUS decomposed the problem, not the order in which it carries out intentions, distinguishing it from graphs often depicted in the AI planning literature (Weld, 1994). In this case, the architecture would execute intentions in the order *I2*, *I3*, *I1*, and *I4*. Another important distinction is that ICARUS never holds the entire plan of this sort in memory. As we will see in the next section, it extends, retracts, and revises a single chain, altering the subproblems, bindings, differences, and intentions as necessary. This contrasts sharply with the approach taken by many AI planning systems, which construct entire courses of action – no matter how complex – before executing them. We maintain that this scheme is inconsistent with the character of problem solving, to which we now turn.

4.2 Selecting Bindings, Differences, and Intentions

Now that we have described the cognitive structures that support problem solving in Icarus, we can examine the processes that operate over these structures. The first step in explaining these mechanisms involves characterizing when the architecture invokes them. This is more difficult it might seem because, as noted earlier, problem solving is interleaved tightly with skill execution. Thus, in some sense, execution is a component of problem solving, but we will generally reserve the latter phrase to those aspects of goal-directed behavior that occur outside the application of compiled skills. We will divide this material into four broad categories: generating bindings, differences, and intentions; selecting heuristically among alternatives; executing intentions and updating problems; and backtracking and recovering from failures. In each case, we first describe the general mechanism and then follow it with examples.

Recall that Icarus represents a problem as a set of goals that describe beliefs it wants to hold. When working on a problem, the architecture first checks to determine if the current belief state satisfies these goals in a consistent manner. If not, then no action is necessary and it can turn its attention to other problems. However, when such an impasse does occur, the comparison process produces one or more differences between the problem elements and the current beliefs, along with an associated set of bindings.

The mechanism responsible for this comparison carries out a repeated greedy search through the set of partial matches between the goal elements and the current beliefs. The system begins with an empty set of differences and an empty bindings list. On each step, it selects a goal randomly that it has not yet considered and checks for a belief that matches it in a way that is consistent with the current bindings. If it finds such a belief, Icarus adds any new bindings that result but does not alter the differences. If it does not find such a belief, then what happens depends on whether the unmatched goal predicate occurs in the effects of a skill or the head of a conceptual clause.

If the goal does not appear in one of these contexts, then the system realizes there is no way to achieve it, so it abandons the current match. If the predicate does appear as a skill effect or conceptual head, the system adds the goal to the differences but does not alter the bindings. This continues until it has considered all of the problem's goal elements, at which point it returns the differences and bindings it has collected. We assume the mechanism is unconscious and rapid, much like Icarus' mechanism for matching the conditions of skills. The architecture repeats this process a number of times in an attempt to generate multiple difference sets, from which it chooses one for further attention.

Once it has selected a set of differences, the problem solver then directs its efforts at finding some way to eliminate them. This involves selecting a skill instance that, if executed successfully, would make progress toward achieving the unsatisfied goals. To this end, Icarus generates two distinct sets of skill instances, one based on chaining backward from differences and another based on chaining forward from beliefs. The first set includes all instantiated skills that, if executed, would achieve at least one of the goals listed in its `:effects` field. Typically, one or more skill clauses will have elements that match in this manner; indeed, the same clause can match in multiple ways, provided it includes more than one effect with the same predicate as the goal. The second includes

all skill instances with conditions that are satisfied by the agent's current beliefs. Again, the same skill clause can match in different ways, leading to separate candidates. Once the architecture has found these candidates and their associated bindings, it selects one and stores it in the problem's `:intention` field. As noted earlier, an intention describes a tentative commitnment to apply a partially bound skill instance.

For example, suppose we provide Icarus with the top-level problem *P1* discussed earlier and shown in Table 7, which involves the three goals *(not (on ?other A))*, *(ontable A)*, and *(not (holding ?any))*, and with the four primitive skills from Table 4. Suppose further that the problem solver has selected the empty bindings and single difference shown for problem *P1* in the table. In this situation, the architecture would produce one skill instance by chaining backward, *(unstack B A)*, which would eliminate the difference *(not (on B A))*. The system would also generate a second skill instance, *(unstack C B)*, which has conditions that match the agent's belief state.

Once the architecture has selected and stored an intention, one of two things may happen. If the intention's conditions (i.e., the conditions of the skill clause with bindings substituted for variables) are matched by current beliefs, then it is applicable and Icarus initiates its execution in the environment, as we discuss later. However, if some of the intention's conditions remain unsatisfied, then the system creates a subproblem, basing the goals on the instantiated conditions and then attempting to solve this new problem using the same mechanisms that we have discussed already.[6] For instance, suppose that, given the skill instances described earlier, the problem solver created the intention *I1* with the head *(unstack B A)*. Because the conditions of this intention are not satisfied, it would generate the subproblem *P2* from Table 7, with the three component goals *(not (holding ?any))*, *(not (on ?other B))*, and *(on B A)*.

In some problems, the goals refer to nonprimitive concepts that let Icarus generate additional candidates. When the architecture selects one of these, it stores a pseudo-intention to 'apply' this inference rule, along with a new subproblem based on the rule's conditions. This step is similar to chaining backward over a skill, except that the goal elements correspond to instantiated subconcepts that would let the agent infer the parent concept, rather than conditions for skill execution. As before, then architecture then identifies differences between the goal elements and current beliefs, which can lead to further chaining, either through skills or another conceptual clause.

An important special case arises when the problem solver chains through a conceptual clause for a negated goal. Because negations involve universal quantification, the system must must satisfy only one negated subconcept to achieve its parent goal, but this requires special machinery. When chaining over the negation of a defined concept, Icarus creates a subproblem composed of *disjunctive* goals rather than the standard conjunctive ones. The system labels the structure as disjunctive, so that it recognizes it as achieved whenever any member of the goal set becomes satisfied. Otherwise, the problem-solving mechanisms remain oblivious to the type of problem, letting it interleave conjunctive and disjunctive tasks as necessary.

6. Creating subproblems by chaining through operators or skills is not a new technique; it has always played a central role in accounts of means-ends problem solving. Newell, Shaw, and Simon (1958) referred to this process as an 'Apply Operator' step, whereas Laird et al. (1987) use the phrase 'operator subgoaling'.

4.3 Heuristics for Skill and Concept Selection

Icarus encounters two sources of search during problem solving. One involves selecting among different bindings and their associated differences. This does not typically involve many choices, and the architecture picks from them at random. The other source comes from the possibility of chaining through different skills or even different instances of the same skill. Later we will examine how Icarus handles situations in which it makes an incorrect choice, but there is no reason these choices must be uninformed. In fact, the architecture incorporates two heuristics to guide the selection of skills during problem solving that reduce the probability of making wrong decisions, although, like other heuristics, they provide no guarantees.

The first heuristic prefers skill instances with effects that reduce more of the differences associated with the current problem. More precisely, the system prefers skill instances that would achieve more currently unsatisfied goals over ones that would achieve fewer of them, which should reduce the number of steps required to solve the problem. For example, consider again the four primitive Blocks World skills in Table 4. Given a task in this domain with two goals, *(not (holding ?any))* and *(on A B)*, Icarus would prefer the skill instance *(stack A B)* over *(putdown A T1)*, because the former has instantiated effects that match both goals while the second's effects match only one. This heuristic also provides a convenient mechanism for masking lower-level skills that have fewer effects with higher-level ones that produce more results.

The other heuristic deals with conditions rather than effects and prefers skill instances with fewer unsatisfied conditions over ones with more. The intuition here should be obvious; the fewer conditions that are unsatisfied, the less effort the agent must devote before it can execute the selected skill. An extreme case occurs when all the conditions of a skill instance match, in which case it is already applicable, but gradations can also occur. For instance, given a Blocks World state in which *(on A B)*, *(on B C)*, and *(not (on ?other A)* hold but *(not (holding ?any)* does not, Icarus would prefer the skill instance *(unstack A B)* over *(unstack B C)*, because the former has one unmet condition while the latter has two of them unsatisfied.

Naturally, choices arise when chaining backward over conceptual rules just as they do for skills, so Icarus also includes heuristics to guide their selection. The 'more effects' preference makes no sense for conceptual clauses, since their heads always involve single literals; thus, it does not come into play in concept chaining. However, the 'fewer conditions' preference remains relevant, so the system favors instances of conceptual clauses that have more satisfied conditions over those with fewer satisfied conditions.

Icarus can utilize these heuristics in two distinct modes. In one setting, the architecture invokes the 'more effects' heuristic as its primary criterion and uses the 'fewer conditions' preference only to break ties. This produces behavior similar to traditional means-ends analysis, although biased toward selecting skills that reduce more differences and that are more nearly applicable. In the other mode, Icarus uses the 'fewer conditions' predicate as its main criterion and resorts to 'more effects' for breaking ties. This results in forward-chaining behavior, since the architecture prefers skills that match against the current state over ones that do not. However, the system is still guided by goal information when this discriminates among candidate intentions. A complete account of

problem solving would explain when an agent operates in each mode, but the current scheme already offers greater flexibility than other frameworks.

4.4 Execution, Updating, and Solution

Recall that the purpose of problem solving is to transform the agent's current state into one that satisfies the problem goals. The selection of bindings, differences, and intentions, along with associated subproblems, are simply means to that end. When successful, this process eventually selects an intention with satisfied conditions, which means the agent can execute it in the environment. When this occurs, Icarus invokes its skill execution module on the intention. As discussed in the previous chapter, if the skill is primitive, the architecture uses its associated action to affect the environment. If not, then the system descends through the skill hierarchy until it reaches a primitive skill it can execute. When this skill has finished, Icarus moves on to later skills, moving up and down the skill hierarchy as necessary. This process continues until the beliefs match the effects of the top-level skill, at which point execution halts.

The problem solver creates an intention in hopes that it will achieve one or more goal. When this occurs, the system marks the goals as satisfied. Also, recall that different goals may share variables that must bind consistently against the agent's beliefs before it counts the problem as solved. For this reason, if the belief that matches a newly satisfied goal includes constants that are variables in the goal, the module also updates the bindings associated with the current problem. If other goals in the problem remain unsatisfied, they continue to influence the creation of future intentions. When this takes place, Icarus either executes this intention, if its conditions are met, or attempts to solve the subproblem of satisfying them, if they are not. Eventually, this can lead the system to achieve one or more current goals, which again it marks, after which it addresses the remainder of the problem.

When all goes well, this interleaving of binding and difference selection, intention selection, subproblem creation, and skill execution finally achieves all the goals associated with the current problem P. When this occurs, Icarus recognizes that it has solved P and can move on to other matters. If the system created P from the conditions of a skill that it intends to apply, then problem solution leads in turn to execution of that skill. If it instead created P from the conditions of an inference rule, then it leads to inference of that rule's consequent. When P is a top-level problem, the architecture simply marks the problem as solved and turns its attention to other top-level tasks.

4.5 Backtracking and Failure Recovery

Of course, problem solving does not always proceed as smoothly as the discussion above suggests. Failure can occur in three different ways. First, Icarus may notice that a generated subproblem is equivalent to, or more difficult than, a problem higher in the chain, in that its goal elements match those of the ancestor. Upon detecting such a loop, the system abandons both the subproblem and the intention that produced it. Second, the architecture may select an intention that is not immediately applicable when it has already reached the maximum depth for problem chains. Since creating a subproblem would exceed this depth limit, the system rejects the intention.

A final reason that Icarus may abandon a set of differences or an intention is that it cannot find any candidates it has not already rejected. This requires representing, storing, and accessing past choices that have failed in the past. To this end, the architecture associates a set of *failure contexts* with each problem. If one of the contexts contains only a set of differences D, this indicates that D did not work out. If a context contains both a set of differences D and an intention I, this denotes that the problem solver failed to execute I within D. When Icarus abandons – for whatever reason – an intention or difference set, it stores the failure context with the problem. The system then uses this information to eliminate candidates from consideration on future rounds of decision making.

Whenever the problem solver rejects an alternative in one of the above ways, it takes one step backward and considers other choices. For instance, when the system abandons an intention I, it stores this failure and attempts to select a different intention from the candidate set, provided one remains. If not, then Icarus also abandons the current difference set D, stores this fact, and turns to another alternative. If no differences remain, then the architecture fails on the problem itself and returns to its parent.

Recall that Icarus carries out an intention as soon as it becomes applicable, thus interlaving problem solving with execution. For this reason, the agent sometimes achieves goals in the environment that cause it difficulty later, when it cannot backtrack mentally because it has already taken action. This is an unavoidable side effect of the eager execution strategy that the architecture employs to avoid keeping entire plans in memory. In such cases, the system simply continues working on the problem from the new situation, backtracking in the environment if no other alternative will let it solve the problem. Many AI researchers would view this as a limitation, but, as noted earlier, we maintain this is generally consistent with how humans solve novel problems, at least ones above a certain level of complexity.

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.[7]

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 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

7. 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)))
```

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.[8] 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.

5.3 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

8. 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)))]
```

versions of them into the new skill clause's `:conditions` 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 `:conditions` 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 `:goals` 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.[9] 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.

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

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

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)))
```