Controlling Physical Agents Through Reactive Logic Programming

Daniel Shapiro
(dgs@leland.stanford.edu)
Engineering Economic Systems
& Operations Research Department
Stanford University, Stanford CA 94305

Pat Langley
(langley@rtna.daimlerbenz.com)
Adaptive Systems Group
Daimler-Benz Research Technology Center
1510 Page Mill Road, Palo Alto, CA

Abstract:

This paper describes Icarus, a language for specifying the behavior of agents that operate in physical domains. This language provides a novel metaphor of "reactive logic programming", which makes it convenient to express both extremely reactive control programs *and* programs with non-trivial deliberative elements. The key features of Icarus are the ability to express hierarchical objectives, requirements, and actions, the use of Prolog-like semantics across function calls, a merged concept of state and action, and a sequence primitive, all embedded in a reactive control loop that considers every relevant action on every cycle of the interpreter. We use a body of examples to illustrate these features, and justify several claims about the expressivity of Icarus relative to existing reactive languages.

1 Introduction: The space of reactive languages

Every artificial agent operating in a physical domain shares the need to integrate sensing and acting in service of objectives. However, since real environments are rarely predictable or even cooperative, agent programmers face a difficult problem; they need to merge goal-driven behavior with situation-relevant response. This design task calls for specialized agent architectures and languages. A well-targeted architecture makes agent programming easier by standardizing an encompassing framework (Albus, 1987; Simmons, 1997); a well-structured language standardizes primitives that make behavior easy to express. Thus, there is a role for research in each arena. This paper introduces Icarus, a language for expressing agent behavior tailored to a broad set of problems that require rich conceptual abstractions and non-trivial calculation in particularly dynamic environments. Icarus contains many novel features, viewed both individually and collectively.

The agent design problem has given rise to a number of languages and architectures that seem equally expressive in the Turing sense, but suited for distinct contexts in practice. Figure 1 characterizes these contexts by the features of the agent's environment and its tasks. We describe environments as static or dynamic, where dynamic environments change independent of the agent during the conduct of its task. We hold that tasks suggest particular coding styles, with *reactive* style employing an iterated situation → action map in the form of a structured lookup table, and *deliberative* style using a procedural encoding of reasoning that can include arbitrarily complex calculations, including search. While reactive programs emphasize fast situation response after prior, off-line computation, deliberative programs emphasize careful run-time thought. The entries in Figure 1 identify the "typical" use of particular programming metaphors; this is a comment more on historical trend than theoretical expressivity.

As an example, consider classical planners (Fikes & Nilsson, 1971; Veloso et al., 1995; Wilkins, 1988), where the classical problem is to plan the construction of a tower of blocks given the initial world state and a list of primitive action descriptors. These systems uniformly make the "STRIPS" assumption that nothing in the environment changes except by agent action, confining their relevance to static worlds. Of course, this is an overstatement, since classical planners can construct contingent plans that let control flow branch during execution. They can also be used in an iterated plan-act-replan cycle (Haigh & Veloso, 1996) that supports reactivity if the environment changes more slowly than the planner generates plans. Nevertheless, from an historical perspective, classical planning has been applied primarily to problems in the deliberative, static corner of the spectrum.

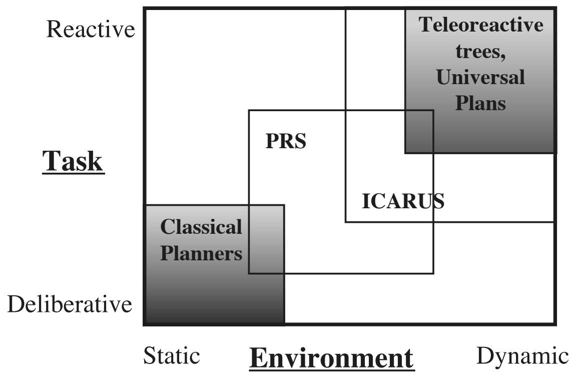


Figure 1: A comparison of agent architectures by task domain.

At the opposite extreme, universal plans (Schoppers, 1987, 1995) and teleoreactive trees (Nilsson, 1994) focus on reaction in the face of extremely dynamic environments. The interpreters for these languages employ an iterated sense-lookup-act cycle that lets applications respond to a spontaneous (and instantaneous) change between any two world states recognized by the plan. This represents the logical endpoint of reactivity, which we call a *fully reactive* design. There are many fully reactive agent architectures (Agre, 1988; Brooks, 1986; Nilsson, 1990), but since universal plans and teleoreactive trees are closest in spirit to the design we will introduce, we will use these systems as exemplars throughout this paper. Note that it is possible (though inconvenient) to define more deliberative calculations within an action lookup formalism, for example, by computing, storing, and then reacting to new information state. As a result, universal plans and teleoreactive trees are most suited to a range of problems that occupy the dynamic, reactive portion of the spectrum.

The PRS language (Georgeff, Lansky, & Bessiere, 1985) is best suited for applications lying between these two extremes because it focuses on procedure execution in the presence of certain reactive constructs. Firby (1989) and Gat (1992) describe other reactive designs with a similar thrust. In particular, PRS acts by processing a directed graph of program statements (e.g., tests, branch points, and goals) to advance the locus of control from a start node towards an end. When encountered, goals can be spontaneously true (obviating the need to invoke achievement procedures) and branches can fail, causing the interpreter to backtrack to the next untried option. However, in contrast with universal plans and teleoreactive trees, PRS never retries a failed option (unless part of an explicit iteration), which means that it assumes a more static world.

From the point of view of agent programming, the upper right hand quadrant of Figure 1 encompasses the most interesting and important problems. This is the region of the task/environment space where significant cognitive loads meet significantly uncooperative domains. What is missing is an agent design tool that is more reactive than PRS but also more deliberative than existing fully reactive designs. In theoretical terms, we would like a language that integrates full reactivity with procedural reasoning. We are now ready to consider one solution.

2 The Icarus language

Icarus is a language for specifying the behavior of agents that perform cognitively rich, reactive tasks in highly dynamic environments. The language unites a broad vocabulary for expressing abstract plans with a procedural framework based on Prolog-like semantics, under a fully reactive control loop. Because of these features, we say that Icarus supplies the metaphor of "reactive logic programming".

This section introduces the Icarus language. We begin with a simple example, and then discuss the key knowledge representations, the Icarus interpreter, the full set of language primitives, and a BNF for Icarus. We continue this development in Section 3, which illustrates the benefits of Icarus via multiple examples.

2.1 A simple Icarus program

Table 1 contains an Icarus program for controlling a bumper car at an amusement park. The code expresses a goal-directed plan for producing a collision with a car identified by an input parameter. The top-level plan contains three parts: an objective, a set of requirements (or preconditions) and a means field that identifies a method of producing the goal. In English, this program reads, "In order to be in contact with the car, accelerate hard when you are both facing and near it". The Icarus interpreter mirrors these steps, but in a more reactive way. It begins by asking if the objective is already true (InContactWith is a sensory test); if so, the plan terminates with success. If the objective is false, the interpreter evaluates the list of statements in the :requires field, and if they are all true (after recursive descent to examine the code for Face and Near), the system evaluates the :means clause, which contains the primitive action to Accelerate. The interpreter returns the Accelerate act, with instantiated parameters, to an external system that executes it in the physical world.

Table 1. An Icarus program for controlling a bumper car.

```
(CollideWithCar (?car)
  :objective ({InContactWithCar ?car})
  :requires  ((Face ?car) (Near ?car))
  :means     ((Accelerate *Hard*)))

(Accelerate (?n) :action ({SetPedal ?n}))

(Face (?car)
  :objective ({Facing ?car})
  :means     ((Turn)))

(Near (?car)
  :objective ((Distance ?car ?d)
              {< ?d *OneCarLength*})
  :means     ((GoForward)))
```

Like other reactive languages, Icarus applies this evaluation process in an infinite loop, starting at the top node of the plan on *every* execution cycle. The top node is the one referenced when invoking the interpreter, as in "(CollideWithCar car54)". Iterated interpretation has a profound impact on the meaning of Icarus programs. First, it lends clauses in the language a temporal scope. If the agent is accelerating towards the target on cycle N and nothing else changes, the interpreter will retrieve the same action on cycle N+1 and continue the acceleration. However, if the target car ever moves in a way that makes Facing false, Icarus will retrieve the Turn action. This transforms Face into a maintenance goal for the Accelerate action. The second consequence of top-down iterated interpretation is that the system can return a relevant action even if the environment undergoes a rapid and radical change between two successive evaluation cycles. The only limitation is that the new world situation must be recognized by the plan. Thus, the bumper car agent might Accelerate, then Turn, GoForward, Turn, Accelerate, and so forth, all in response to a changing world. This is the core competence of fully reactive languages.

2.2 Knowledge structures

Icarus contains a small number of key knowledge structures. The first, called a *skill*, encodes agent behavior. Skills can represent a situation, an action, a rule, or a process for accomplishing an objective that is relevant so long as certain conditions apply. Skills are represented by frames with name and parameter fields followed by several other slots. We have already seen several examples; CollideWithCar, Face, Near, and Accelerate are all skills from Table 1.

Skills with the fields :objective, :requires, or :means specify logical structure and conditional control flow. An :objective slot contains an ordered conjunction of goals, a :means slot contains a parallel disjunction of methods for accomplishing the objectives, and a :requires slot contains an ordered conjunction of maintenance conditions for application of the :means. The :objective slot defaults to False, the :requires to True, and the :means to False. This guarantees that a slot will be evaluated if the others are not specified.

A skill with the :sequence keyword contains Icarus statements that should be evaluated in sequence as each succeeds, but not rechecked thereafter. A skill with an :action slot contains code that alters the physical world. Finally, a skill with a :sensor slot holds functions that examine the world and bind the results to Icarus variables. We summarize these structures in Table 2.

Table 2. The four types of Icarus skills.

Keyword	Content
:action	code that effects the world
:sensor	code that senses the world and binds variables
:sequence	sequential control flow
:objective :requires :means	logical structure and control flow

A *plan* in Icarus is the tree obtained by threading the references to skills that descend from a top-level call. For example, the invocation "(CollideWithCar car54)" generates a plan where CollideWithCar points to Face, Near, and Accelerate, while Face points to Turn, and Near points to Distance and GoForward. In this document, the plan will often be obvious from context.

Icarus provides one built-in data representation called the *short-term memory*, which holds knowledge that must persist across evaluation cycles. The short-term memory contains a list of expressions, and is accessed via a unification procedure like that in Prolog. For example, if short term memory is the list "((car car54) (car54 (10 20) 1:00:04))", the form "(car54 ?loc ?time)" binds ?loc to the location (10 20) and ?time to 1:00:04.

2.3 The Icarus Interpreter

The Icarus interpreter has a simple but unusual organization based on a three-valued logic where every statement in the language evaluates to one of True, False, or an Act. These values take on special meaning if the clause names a skill (or, recursively, a plan). 'True' means the statement was true in the world, 'False' means the plan did not apply, and an 'Act' return identifies a piece of code for controlling actuators that addresses the objectives of the plan. We represent a return value of True as a collection of variable bindings that satisfy the clause, and an Act return as code with instantiated variables.

The interpreter processes the tree of skills in an endless loop, beginning with the top-level node every cycle. Within each cycle Icarus behaves much like Prolog; it returns all binding environments that render the plan's value True, but in addition it generates the resulting Acts. Icarus can be configured to return all of these Acts, the first Act found, or the single best Act as determined by a value calculation (not described here). We use the "first act found" interpretation in this paper. The system can also be configured to selectively expand the plan by considering a subset of the skills in each :means field. For example, it can expand the first *n*, or choose on the basis of a value calculation.

Table 3. Pseudocode for EvaluatePlan.

```
Function EvaluatePlan (skill, binding environment)
 Returns [list of actions, list of binding environments]

 Declare temp, result := [list of actions, list of binding environments]

 result ← EvaluateConjunction(Objective (skill), binding environment)
 If empty(result) Then
   Begin
     temp ← EvaluateConjunction(Requires(skill), binding environment)
     If empty(temp)
        Then return False
         Else result ← EvaluateDisjunction(Means(plan  element), temp)
   End

 Return [BestAct(result), NewBindings(result, skill)]
```

Table 3 describes the process of evaluating the objective/requires/means skill. The main point of this pseudo-code is that Icarus enforces a logical interpretation of skill slots. If the objective field produces a non-empty result, the work is essentially done. If the objective fails, the interpreter will look at the :requires field, and then the :means if evaluation of the :requires succeeds. The only complexity in this code comes from handling simultaneous Act and True returns, since these must be accumulated as evaluation progresses. For example, if the first clause of the :requires field is the match form "(bumper-car ?b)", any number of alternate bindings for ?b may result. Two or three might lead to plausible "BumpIt" Acts. Before returning, EvaluatePlan filters the set of Acts and True returns by selecting the one best action and isolating any new bindings discovered for input parameters; this implements a Prolog-like variable passing discipline.

The algorithm for evaluating :sequence states is a bit more arcane, since reactive languages typically eschew the concept of saved process state; they treat every cycle of the interpreter as an independent procedure. This gives rise to some unusual problems: in order to track progress through a sequence we need a definition for the equality of situations presented at two different interpreter cycles. Similarly, to know when to abort a sequence, we need a definition of continuity. Our solution maps situation equivalence onto the equality of binding environments, which lets the interpreter track progress via variable bindings. For example, if BumpIt(?car) is a three-step sequence, the agent can be in the third step with respect to the red car and the first step with respect to the blue one. The definition of continuity is subtler. It is tempting to abort all sequences after a fixed period of time (measured by interpreter cycles or elapsed time), but the appropriate number would be context sensitive. It seemed inelegant to define sequences with such a parameter. Instead, we map continuity onto the constancy of the binding environment. Thus, a sequence will abort unless the same situation is present every time the sequence skill is called. This lets the agent begin a sequence, shift its attention to event processed by an entirely different portion of the plan, and then return to the sequence an arbitrary amount of time later. If the interpreter receives a binding environment that matches what the sequence last saw, the agent will pick up exactly where it left off. We discuss an example of the sequence mechanism at the end of Section 3.

2.4 Icarus primitives

Icarus provides several primitives for composing skills. In particular, the { } syntax allows access to embedded Lisp functions that encode numeric predicates and sensory tests. For example, the predicate {Facing ?car} returns True or False after sensing the world. Icarus also supports an explicit Bind operator and two versions of a three-valued negation; *not* maps True to False, False to True, and preserves Acts, while *notA* is the same except it maps Acts to False. The final primitive is a match request that accesses data stored in short-term memory. Icarus treats a clause as a match request if it has no other interpretation. The Lisp functions *update*, *store*, and *forget* write to short-term memory; sensor skills use them implicitly to record sensed objects in persistent memory.

2.5 A BNF for Icarus

Table 4 provides a BNF for Icarus programs. We supply it primarily in the interests of precision, and to remove any lingering ambiguity about Icarus expressions. This BNF shows that skills are structured objects with names and parameters, and that there are four different types. The first produces a plan hierarchy, as it support objectives, requires and means fields. The second also supports hierarchy and implements the Icarus concept of sequence. The last two implement primitive actions and sensors, which contain calls to Lisp functions. The BNF also identifies five primitive clause types that can fill any objective, requires, or means slot. Lisp calls represent escapes to Lisp. The bind function causes Icaurs variables to take on values. Match-forms match onto short term memory, and the two forms of negation manipulate return values, as discussed above.

Table 4. A BNF for Icarus programs.

```
<skill> := (<name> (<parameter>*) <body>)
<body> :=   [:objective (<clause>*)] [:requires (<clause>*)] [:means (<clause>*)]
           | :sequence (<clause>*)
           | :action (<Lisp-call>*)
           | :sensor (<Lisp-call>*)

<clause> := <Lisp-call> | (bind <var> <Lisp-call>) | <match-form>
           | (*not* <clause>) | (*notA* <clause>) | <skill>
<Lisp-call> := {<Lisp-form>}
<match-form> := <Lisp-form>
<var> := ?<symbol>
```

Now that we have described the syntax and semantics of the Icarus language, we are in a position to discuss its implications for expressing agent behavior. In particular, we will make a number of claims to the effect that Icarus increases the scope of fully reactive programs by supplying novel programming abstractions. We explore these claims by example in the next section.

3 The benefits of Icarus

We maintain that Icarus is a facile and expressive programming language, which will support a new class of agents that are fully reactive but can also deliberate. This type of claim is somewhat difficult to support since it calls for a fundamentally aesthetic evaluation. In response, we will at least phrase our claims succinctly, provide numerous examples, and give a basis for comparison with other languages. In particular, we describe the benefits of Icarus in the context of its two nearest neighbors: universal plans and teleoreactive trees.

Claim 1: Icarus is at least as expressive as teleoreactive trees and universal plans.

Although we will show that Icarus has many novel features, we begin by demonstrating it preserves existing representational capabilities. We do this by defining exact translation rules between its representations and those used in other languages.

We begin with teleoreactive trees, which are ordered lists of situation-action rules, interpreted via a top-down, iterated loop that selects the first matching rule. They are constructed such that the leading predicates are produced (if possible) by actions deeper in the tree, although there is no explicit goal-action pairing. Table 5 contains a teleoreactive tree for the Bumper Car problem seen above. It has a natural English translation: if the agent is already in contact with the car, return True, else if Facing and Near are true, return the BumpIt action, and so forth (through the nested if). As in Icarus, these conditions can change from one time step to the next without effecting the validity of the plan.

The right half of Table 5 illustrates the Icarus representation for the same control flow. We can map teleoreactive trees into Icarus plans by employing two transformations:

(1) Create an Icarus skill for each rule in the teleoreactive tree and place the trees' predicates in the Icarus :requires clause.
(2) Construct a single Icarus skill that lists the transformed rules as negated :requires clauses.

The Icarus representation employs the three-valued negation construct, *not*, to preserve Act returns but transform rule failure into True to enable the appropriate control flow. In English, "if the first rule does not hold, try the second, then the third, etc.". By construction, Icarus returns the Act associated with the first relevant rule. Teleoreactive trees also allow ordered disjunctions of subtrees; Icarus captures this control flow via a nested use of the same transformation.

Table 5. A teleoreactive tree and the corresponding Icarus plan.

Teleoreactive Tree

```
InContactWith(car) → T
Facing(car) ∧ Near(car) → BumpIt
Facing(car) → GoForward
T → Turn
```

Icarus plan

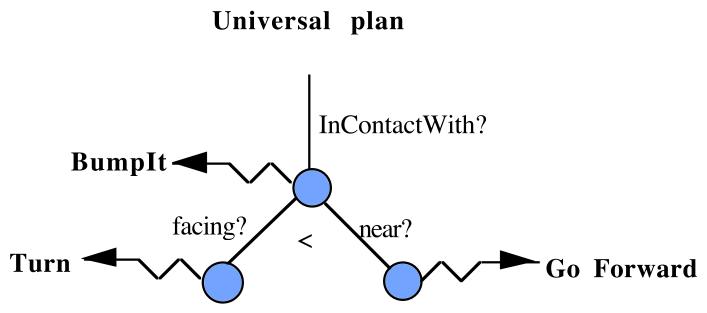
```
(CollideWithCar (?car)
  :requires
    ((*not* (TrueWhenInContact ?car))
      (*not* (BumpItWhenNearAndFacing ?car))
      (*not* (GoForwardWhenFacing ?car))
      (*not* (TurnWhenTrue)))

(BumpItWhenNearAndFacing (?car)
  :requires ({Near ?car} {Facing ?car})
  :means ((BumpIt)))
```

The correspondence between an Icarus plan and Universal Plan is similarly easy to state, as shown in Figure 2. The left-hand side of this diagram provides a graphical representation of a Universal Plan, where the arcs are predicates (interpreted as goals above a node and preconditions when below it), the jagged lines identify actions, and the "<" symbol indicates an ordered conjunction of the outgoing arcs. The Universal Plan interpreter iteratively processes this structure in a top-down left-to-right fashion using the rule: if Not(goal) and Precondition then return Action. This leads to the English translation, "if the agent is InContactWith the car then return True, else if Facing and Near hold, return the BumpIt action in order to make InContactWith true". Note that universal plans make the relation between actions and goals explicit. We can translate a Universal Plan into an Icarus plan with three transfomations:

(1) Define an Icarus skill for every node in the universal plan diagram.
(2) Place the goal of that node in the :objective field and the action in the :means field.
(3) Map every precondition onto a recursively defined skill and place its name in the :requires field; a True precondition corresponds to an empty :requires field.

We note that universal plans support parallel preconditions, which are not yet implemented within Icarus. However, our plans to incorporate them only require localized changes to the interpreter, specifically to represent parallel return values and execute multiple actions. Icarus already has the ability to evaluate multiple means fields in the same pseudo-parallel fashion found in universal plans.

In summary, universal plans and teleoreactive trees map easily and directly into Icarus plans. This not surprising, since all three representations are based on predicate logic and were all designed for reactive control. The remainder of this section highlights their differences, some of which are profound.



Icarus plan

```
(CollideWith (?car)
  :objective ({InContactWith ?car})
  :requires  ((Face ?car) (At ?car))
  :means     ((BumpIt ?car))

(Face (?car)
  :objective ({Facing ?car})
  :means     ((Turn)))

(At (?car)
  :objective ({Near ?car})
  :means     ((GoForward)))
```

Figure 2. A universal plan and its Icarus equivalent.

Claim 2: Icarus can encode fully reactive plans that employ novel abstraction principles. Like all programming languages, reactive systems support abstraction methods. Universal plans and teleoreactive trees allow recursion down precondition links, though actions must be primitive. Icarus provides additional methods; it supports abstract objectives and means, in addition to abstract requirements, because it lets each of these fields contain subplans. This increases the overall flexibility of Icarus programs.

Table 6 illustrates the concept of an abstract action by transforming the BumpIt primitive of Figure 2 into a hierarchical plan. This subplan has some complexity; it lets the agent Ram enemies and Tap friends, and it includes a further reactive subplan for acquiring electrical power. The key feature, however, is that BumpIt is a means for causing a collision in Figure 2, but a plan for doing so in Table 6. Thus, Icarus supports abstract actions. Note that universal plans and teleoreactive trees lack this mechanism. The nearest Universal Plan to the BumpIt skill in Table 6 would read, in Icarus, as:

```
(BumpIt (?car)
  :objective ((InContactWith ?car))
  :requires ((HavePower)(Driver ?car ?person) (Enemy ?person))
  :means  ((SetPedal 100)))
```

This encoding associates the most specific action with the most global objective, and therefor elevates details in an awkward way.

Table 6. An Icarus action defined by a subplan.

```
(BumpIt (?car)                          (Ram (?car)
  :objective ((InContactWith ?car))       :requires ((Driver ?car ?person){Enemy ?person})
  :requires  ((HavePower))                :means  ((SetPedal 100)))
  :means     ((Ram ?car) (Tap ?car)))
                                        (Tap (?car)
(HavePower (?car)                         :requires ((Driver ?car ?person){Friend ?person})
  :objective ({Power ?car ON})            :means    ((SetPedal 20))
  :means     ((HitPowerPole ?car)
```

Table 7 illustrates an abstract objective encountered in the expansion of the Near predicate from Figure 2. Here, we focus on KnowLocation (a knowledge goal), which we view as an ordered sibling to the goal of computing distance. The plan for KnowLocation asks if short-term memory contains a recent record of the target car, and returns its location if true. Otherwise, it calls on LookForCar, which either observes the car (CarsInView is a sensor skill) or invokes a physical search procedure. Even though this search might involve significant effort, Near treats KnowLocation as an abstract objective.[1]

[1] In universal plans, primitive predicates lacking a truth value implicitly expand into knowledge acquisition plans. In contrast, Icarus lets any clause in the :objective field contain a subplan.

Table 7. An Icarus subplan for an abstract objective.

```
(Near (?car ?d)                              (LookForCar (?car ?loc)
  :objective ((KnowLocation ?car ?loc)         :objective ((CarsInView  ?c)
              (Distance-to ?loc ?d)                        {member ?car ?c}
              {< ?d *OneCarLength*})                       (Location ?car ?loc))
                                               :means ((Turn))
(KnowLocation (?car ?loc)
  :objective ((?car ?loc ?time)
              {recent ?time})
  :means     ((LookForCar ?car ?loc))
```

In summary, Icarus can express hierarchical, reactive plans for actions and objectives that are not possible in universal plans and teleoreactive trees. This provides the programmer with a powerful vocabulary for expressing behavior and for engaging in hierarchical design.

Claim 3: Icarus can encode fully reactive plans with novel cognitive elements.

Although universal plans and teleoreactive trees provide logically structured maps from situation to action, neither system provides primitives for reasoning about the world during action selection. That work is relegated to the Prolog or Lisp implementations of primitive predicates and actions. This makes it awkward to design agents that need to deliberate to some degree to select a reaction. Icarus incorporates general-purpose applicative programming constructs to address this problem.

Table 8 contains an example of this interstitial cogitation. The code implements an abstract action that identifies and moves behind the fastest bumper car in a crowded arena. This behavior requires a relatively simple calculation to sort three cars by speed, although the domain problem (and the requirement on Icarus) can obviously generalize. Icarus supports this computation via a logic-programming metaphor using function calls and Prolog-like parameter passing. The Icarus language is general enough to support recursive programs.

Table 8. Calculation and recursion within Icarus.

```
(MoveToFastestLane ()                         (Fastest (?c1 ?s1 ?c2 ?s2 ?Car ?Speed)
 :requires                                     :objective ({>= ?s1 ?s2}
   ((CarAheadLeft ?car1 ?speed1)                           (bind ?Car ?c1)
    (CarAheadCenter ?car2 ?speed2)                         (bind ?Speed ?s1))
    (CarAheadRight ?car3 ?speed3)              :means
                                                 ((fastest ?c2 ?s2 ?c1 ?s1 ?Car ?Speed)))
    ;;bubble sort the three cars by speed
    (Fastest ?car1 ?speed1 ?car2 ?speed2 ?c ?s)
    (Fastest ?c ?s ?car3 ?speed3 ?fastest-car))
 :means ((MoveBehind ?fastest-car)))
```

Note that Icarus does not support the full generality of Prolog since literals can match against the contents of short-term memory but not against program memory. Moreover, all function calls must be explicit. We adopted this limitation to help ensure a fast reactive cycle, since it transforms hierarchical plan interpretation into a simple tree walk.

Claim 4: Icarus can encode plans with processes as goals.

One of the most unusual features of Icarus is that it treats situations and processes in a fundamentally symmetric way. In a mechanical sense, this property arises from the fact that any clause in an :objective, :requires, or :means slot can be a predicate that represents a situation or a skill that represents a process (a situation in which a particular action may be repetitively applied). This duality lets us define behaviors with processes as objectives.

Table 9 contains an example. The left-hand side of this table gives the top-level plan for Bumper Cars with processes as objectives. It says, "the goal of Bumper Cars is to cause collisions, and to drive both recklessly and impolitely. If none of these behaviors are accessible, then drive laps until they are." This encoding makes intuitive sense for this domain, since riding bumper cars is all about engaging in an activity. Contrast this with the situation-oriented version of the plan on the right-hand side of the table, which is the equivalent of the closest corresponding Universal Plan or teleoreactive tree. This code is in some sense forced to conjure up a metric for spills and thrills to give a focus for behavior. It also has an unwanted side effect; the agent will stop playing the game once the target thrill count is achieved.

Table 9. Process and situation oriented versions of a plan.

```
(RideTheBumperCars ()                    (GenerateSpills ()
  :objective ((CollideWithSomeCar)         :objective ((?spill-count ?sc){> ?sc 10})
              (DriveRecklessly)            :means ((CollideWithSomeCar)))
              (DriveImpolitely))
  :means ((DriveLaps)))                  (GenerateThrills  ()
                                           :objective ((?thrill-count ?tc){> ?tc 10})
                                           :means    ((DriveRecklessly)))
```

The BumperCar example also raises a more general point that domain problems can have state-like or process-like nature. State-like problems focus on achievement or maintenance goals, while process-like problems emphasize agent actions over their consequences. We submit that the technology of planning and reactive programming is entirely too biased towards state-like approaches, to the detriment of process-like models. As a result, systems like Icarus which deliberately blur the state/process boundary may deserve some special attention (Drabble, 1988; Earl & Firby, 1997).

Claim 5: Icarus can encode behavior composed of sequential reactive contexts.

Although languages based on an iterated action map support extremely reactive behavior, they have a down side; they *require* the programmer to write a fully reactive plan that can respond to any event at any time. This is clearly not true of the world; some events occur in sequence because they are constrained by the laws of physics, while others occur in sequence because the agent is the root cause. Icarus addresses this issue by supplying a sequence construct in addition to its reactive ones.

Table 10 gives an example that compares a sequential and fully reactive encoding of a plan for playing BumperCars. The right-hand side shows a reactive plan of the kind expressible with universal plans or teleoreactive trees; it states that the objective of PlayBumperCars is to Leave the arena but arranges for leaving to require the precondition “Ride”. This pattern repeats, connecting the action plans in a seemingly causal way. This structure can produce the behaviors Pay, Ride, and Leave in sequence, but it also lets the agent Pay, discover the ride is magically over, and then Leave. Alternatively it might Pay, then be teleported outside the arena making PlayBumperCars evaluate to “already true”. This unwanted generality comes at a price, since the interpreter must ask *all* the relevant context-setting questions on every decision cycle. This interaction with the frame problem is destined to be extremely inefficient as plan complexity grows.

In contrast, Icarus also supports the more constrained, sequential version of PlayBumperCars. Here, the programmer simply lists the behaviors that must occur in sequence, while the interpreter internally tracks process state. Note that Pay, Ride, and Leave can be reactive plans in their own right, which return True when done. The parameter ?day, which identifies the particular instance of the sequence, is used by the interpreter to index the appropriate process state and to delete saved state when a sequence is interrupted, as we described Section 2.3.

Table 10. Sequential and reactive versions of PlayBumperCars.

```
(PlayBumperCars (?day)          (PlayBumperCars () :objective ((Leave)))
  :sequence ((Pay)
             (Ride)             (Leave () :requires ((Ride))
             (Leave)))                    :means (<plan for leaving>))

                                (Ride () :requires ((Pay))
                                         :means (<plan for riding>))
```

In summary, Icarus' sequence construct increases the expressivity and simplicity of fully reactive languages in a novel way. Specifically, it lets programmers define reactions to a sequence of events in the external world, and it lets the interpreter avoid redundant testing.

4 Related work

We have already attempted to characterize Icarus' position in the space of architectures for controlling physical agents, but given the recent activity in this area, a few more words seem in order. The classical approach to agent control exemplified by STRIPS (Fikes & Nilsson, 1971) used deliberative planning methods followed by open-loop execution, which clearly encounters problems in dynamic environments. Early responses to these problems (Agre, 1988; Brooks, 1986; Schoppers, 1987) relied on purely reactive methods that operated entirely in closed-loop fashion. But, clearly, both paradigms contain valuable characteristics that would benefit an intelligent agent.

The Icarus architecture, which augments a reactive core with limited deliberative and sequence-handling abilities, aims for the best of both worlds.[2] However, other recent research on agent languages with similar motivations has produced somewhat different results. Work by Reddy and Tadepalli (1997) on reactive hierarchical task networks comes closest to our own, in that it represents control knowledge in hierarchical structures and uses explicit requirements to determine when it can execute a plan. One key difference is that this approach assumes sequential execution within each plan, whereas Icarus assumes reactive control. The RAMA system (Earl & Firby, 1997) also encodes control knowledge as process-oriented plans, but again assumes that plans, once initiated, run to completion rather than operating in reactive mode, as in Icarus.

Many recent architectures for robotic agents have a number of distinct levels that rely on different representations and control paradigms. One example comes from Simmons et al. (1997), who describe a robotic architecture for office-delivery tasks. Their Xavier system uses a means-ends planner to select tasks, a forward-chaining heuristic search to find paths, a Markov decision process for navigation, and constrained optimization for avoiding obstacles. Yamauchi et al. (1998) report another multi-level architecture, Magellan, that uses heuristic search for path planning and a reactive controller for obstacle avoidance. In contrast, Icarus represents all control knowledge in the same formalism, from high-level activities like *go for a drive* to low-level ones like *step on the brake*.

Another approach to combining deliberation with reactivity invokes learning to transform the former into the latter. For instance, the Soar architecture (Laird & Rosenbloom, 1990) supports deliberative methods like means-ends analysis, but also includes a chunking process that caches generalized results from deliberation into more efficient stimulus-response rules. When they applied this framework to the task of controlling a robot arm, Soar learned control rules that let it manipulate blocks in a reactive manner. Similarly, Benson (1995) describes an extension of Nilsson's framework that learns durative operators from experience, then uses deliberative planning with these operators to generate teleoreactive trees for the task of controlling a simulated aircraft. Icarus, at least in its current incarnation, assumes that its reactive knowledge is provided by a programmer rather than learned from experience.

---

[2] An earlier version of Icarus (Langley, 1997) supported a similar reactive formalism that merged states and process, but did not support hierarchical skills, extended calculation, or the notion of reactive logic programming.

Before closing, we should expand on the distinction introduced earlier between architectures and languages for physical intelligent agents. Briefly, an architecture specifies a set of memories and processes that one can use to implement an agent, along with communication channels among them, whereas a language indicates a specific syntax and semantics for instantiating these memories and processes. Much work on reactive controllers (Brooks, 1986; Hayes-Roth, 1991) focuses on architectural issues but places no constraints on the language used to implement behaviors. SCHEMER (Fehling, personal communication, 1996) has a similar flavor. Some research on multi-level architectures for robot control take an analogous position, although COLBERT (Konolige, 1997) is instead a robotic language with few architectural constraints. In contrast, Icarus constitutes an architecture and a language, making it more akin to frameworks like Soar and ACT (Anderson, 1983), which also address both issues.

5 Concluding remarks

This paper has introduced Icarus, a tool for constructing artificial agents that perform tasks involving significant cognitive effort in extremely dynamic environments. We demonstrated, by example, that the language has novel expressive properties which generalize existing reactive languages. In particular, Icarus can encode abstract actions and hierarchical objectives, it supports a logic-programming metaphor, it can represent both state-like and process-like plans, and it can express sequential behavior. These features hold promise to make Icarus a language of choice for building reactive plans.

We have tested Icarus' ability to express agent behavior by constructing disparate applications: pole balancing (with process objectives), a reactive version of the Tower of Hanoi, a control program for a simulated cleaning robot, and a watering system for the author's garden that grows competitive pumpkins for size. We are currently building our largest system; an agent that controls a vehicle in an extensive traffic simulation. Our experience with each of these projects has reinforced our belief that Icarus is a powerful and convenient language for specifying reactive behavior.

We are also extending Icarus in several directions. First, we are taking advantage of the interpreter's simplicity to transform Icarus from a behavior generator into a plan recognizer. In particular, we hope to interpret traces of human driving behavior as instances of contingent plans. A second extension builds on Icarus' ability to represent abstract plans; we have attached value estimates to skills at multiple levels in the plan hierarchy, and are developing algorithms for hierarchical credit assignment and convergent value learning. A third effort examines the impact on system performance when we use these estimates to prune exploration of the Icarus plan tree. This work is aimed at giving Icarus a decision-theoretic semantics that will enhance its ability to model deliberate thought.

Acknowledgements

We thank Ross Shachter, Derek Ayers, and John Hawkins for conversations that helped clarify this work, and Michael Fehling for discussions that influenced our framing of these issues. We owe special thanks to Marcel Schoppers for many discussions of reactive agent design.

References

Agre, P. (1988). *The dynamic structure of everyday life*. Tech Report AI-TR-1085, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Albus, J. S., McCain, H. G., & Lumia, R. (1987). *NASA/NBS standard reference model for telerobot control system architecture (NASREM).* NBS Technical Note 1235, National Bureau of Standards, Gaithersburg, MD.

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Benson, S. (1995). Inductive learning of reactive action models. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 47-54).

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation, 2,1.*

Drabble, B. (1988). *Planning and reasoning with processes.* Tech Report TR-56, Artificial Intelligence Applications Institute, University of Edinburgh, Scotland.

Earl, C., & Firby, J. (1997). Combined execution and monitoring for control of autonomous agents. *Proceedings of the First International Conference on Autonomous Agents* (pp. 88-95).

Fikes, R., & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2,* 189-208.

Firby, J. (1989). *Adaptive execution in complex dynamic worlds.* PhD Thesis, Department of Computer Science, Yale University, New Haven, CT.

Gat, E. (1992). Integrated planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp 809-815).

Georgeff, M., Lansky, A., & Bessiere, P. (1985). A procedural logic. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.* Morgan Kaufmann.

Haigh, K., & Veloso, M. (1996). Interleaving planning and robot execution for asynchronous user requests. *Proceedings of the International Conference on Intelligent Robots and Systems.*

Hayes-Roth, B. (1991). An integrated archtiecture for intelligent agents. *Sigart Bulletin, 2*, 79-81.

Konolidge, K. (1997). COLBERT: A language for reactive control in Sapphira. In Brewka, G., Habel, C., and Nebel, B. (Eds.), *KI-97: Advances in artificial intelligence.* Springer.

Laird, J., & Rosenbloom, P. (1990). Integrating execution, planning, and learning in Soar for external environments, *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 1022-1029).

Langley, P. (1997). Learning to sense selectively in physical domains. *Proceedings of the First International Conference on Autonomous Agents* (pp. 217-226).

Nilsson, N., Moore, R., & Torrance, M. (1990). *ACTNET: An action-network language and its interpreter*. Tech. Report, Department of Computer Science, Stanford University, Stanford, CA.

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

Reddy, C., & Tadepalli, P. (1997). Learning goal decomposition rules using exercises, *Proceedings of the Fourteenth International Conference on Machine Learning* (pp. 278-286).

Schoppers, M. (1995). The use of dynamics in an intelligent controller for a space faring rescue robot. *Artificial Intelligence, 73,* 175-230.

Schoppers, M. (1987). Universal plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 1039-1046). Morgan Kaufmann.

Simmons, R., Goodwin, R., Haigh, K., Koenig, S., & O'Sullivan, J. (1997). A layered architecture for office delivery robots. *Proceedings of the First International Conference on Autonomous Agents* (pp. 245-252).

Veloso, M., Carbonell, J., Perez, M., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, *7,* 81-120.

Wilkins, D.E. (1988). *Practical planning: Extending the classical AI planning paradigm*, Morgan Kaufmann.

Yamauchi, B., Langley, P., Schultz, A. C., Grefenstette, J., & Adams, W. (1998). *Magellan: An integrated adaptive architecture for mobile robotics*. Tech. Report 98-2, Institute for the Study of Learning and Expertise, Palo Alto, CA.