Controlling Gaming Agents via Reactive Programs

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

Abstract

This paper introduces Icarus, a language for specifying the behavior of artificial agents that require extremely reactive, yet goal-focused control programs which include non-trivial cognitive elements. The key features of Icarus are the ability to express hierarchical objectives, actions, and plans, the use of function call semantics, and the ability to express processes as objectives, all embedded in a reactive control loop than considers every relevant action on every cycle of the interpreter. Icarus has a small footprint and executes fast. We use a body of examples to illustrate these features, and establish Icarus' relevance to the problem of encoding purposive, but reactive artificial agents for computer games.

Reactivity Meets Computer Games

Every artificial agent operating in a physical domain shares the need to integrate sensing and acting in service of objectives. However, since real and simulated 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 is a key issue in applications as diverse as planetary rovers which have to collect rocks while avoiding dust storms, and computer games where synthetic characters have to pursue their own agendas while responding to player actions and interference.

Several technologies from Artificial Intelligence address this problem. Interleaved plan-execution systems generate plans from goals and action descriptors, then replan in response to observations (Haigh and Veloso 1996). Reactive execution systems employ structured situation-action maps to represent highly contingent behavior (Agre 1988, Brooks 1986, Firby 1989). Universal plans (Schoppers 1987) and teleoreactive trees (Nilsson 1994) use predicate logic representations and are perhaps the best known. Layered architectures merge goal and situation driven response by segregating functionality, e.g., into layers for mission planning, behavior selection, and mobility control, or scheduling, path selection, and navigation (Simmons et al. 1997). One can also invoke learning to transform goal oriented deliberation into situation-action rules (Laird and Rosenbloom 1990).

While all of these AI technologies are relevant to agent design for computer games, the gaming context imposes special constraints. In particular, any tool suite must be *scaleable, "fungible", and efficient*. Scaleable techniques support the creation of many agents, and agents of increased complexity which strategize in addition to react. Fungible techniques (to borrow an economic term) make it easy to specialize agent behavior and add the nuances which are in some sense the saleable product of computer games. (This is generally accomplished via hands-on procedural programming.) The need for efficiency is a given; useful agent design tools must have low overhead measured in both size and speed, even for exploratory games which require second vs fractional second response.

AI techniques and gaming constraints mismatch to a significant degree. Replanning systems require plan generators that seldom provide sub-second turn-around. Reactive techniques may not scale well as the space of contingencies grows unless they have access to planning engines or more general data and control flow constructs. However, designs with such structures have large footprints, e.g., PRS (Georgeff, Lansky, and Bessiere 1985). Reactive approaches also tend to be non-fungible because they lack rich methods of expressing plan hierarchy which can be used to encode behavioral nuances. Layered architectures are clearly large-footprint items, as are learning technologies which begin with compute intensive representations of behavior. What is missing is a time and memory efficient agent design tool that supports reactivity, cognitive (or strategic) calculation, and a convenient means of specializing behavior.

Icarus by Example

Icarus is a language for specifying the behavior of agents that perform cognitively rich, reactive tasks in highly dynamic environments. Icarus provides several novel features which make it a powerful programming tool. In particular, it unites a broad vocabulary for expressing abstract plans with a procedural framework based on Prolog-like semantics under a fully reactive control loop. This section illustrates Icarus via example. We discuss the full syntax and semantics of Icarus and relate it to prior art in (Shapiro and Langley 1998).

Table 1 contains a simple 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 element contains three parts: an objective, a set of requirements (or preconditions) and a means field which 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 (InContactWithCar is a sensory test); if so, the plan terminates with success. If the objective is false, the interpreter evaluates the statements in the :requires field in order, 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, BumpIt. The interpreter returns BumpIt to an external execution system that applies the code in its action clause within the physical world.

Table 1: An Icarus program for driving a bumper car.

```
(CollideWithCar (?car)
  :objective ((InContactWith ?car))
  :requires  ((Face ?car) (Near ?car))
  :means     ((BumpIt ?car)))

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

(Near (?car)
  :objective ((Distance ?car ?d)
                {< ?d 10})
  :means     ((GoForward)))

(BumpIt (?car) :action ({SetPedal 100}))
```

Like other reactive languages, Icarus applies this evaluation process in an infinite loop, starting at the top node of the plan 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 retrieves {SetPedal 100} on cycle N and nothing else changes, the interpreter will select 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 BumpIt 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 BumpIt, then Turn, GoForward, Turn, BumpIt, and so forth, all in response to a changing world. This is the core competence of fully reactive languages.

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. An Act is a body of code with instantiated variables meant to effect the world. True, represented by a collection of variable bindings, means the clause is true under those bindings. These values take on special meaning if the clause names a plan element (or recursively, a plan): an Act return means “take this action to pursue the plan’s purpose”, True means “the purpose has already been accomplished in the world”, and False means the plan does not apply. Icarus processes the tree of plan elements in an endless loop, beginning with the top level node every cycle. Within each cycle Icarus behaves much like Prolog; it returns all of the binding environments which render the plan’s value True, but in addition it generates all of 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.

Novel Icarus Features

Like all programming languages, reactive systems support abstraction methods. Icarus provides a particularly rich set by allowing each of the objective, requirement, and means fields to contain whole sublans (for comparison, universal plans support recursion down precondition links but its actions are primitive). This increases the flexibility of Icarus programs.

Table 2 illustrates the concept of an abstract action by transforming the BumpIt primitive of Table 1 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 point, however, is that BumpIt is a means for causing a collision in Table 1, but a plan for doing so in Table 2. Thus, Icarus supports abstract actions.

Table 2: An Icarus action defined by a subplan.

```
(BumpIt (?car)
  :objective ((InContactWith ?car))
  :requires   ((HavePower))
  :means    ((Ram ?car) (Tap ?car)))

(HavePower (?car)
  :objective ((Power ?car ON))
  :means     ((HitPowerPole ?car)))

 (Ram (?car)
  :requires ((Driver ?car ?person)(Enemy ?person))
  :means   ((Accelerate 100)))

(Tap (?car)
  :requires ((Driver ?car ?person)(Friend ?person))
  :means   ((Accelerate 20)))

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

Icarus also supports abstract objectives, as shown in Table 3 which expands the Near predicate of Table 1. Here, we focus on KnowLocation (a knowledge goal) viewed as an ordered sibling 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 (not shown) which might observe the car or invoke a physical search procedure. In either case KnowLocation is an abstract objective from the perspective of Near.

Table 3: A subplan for an abstract objective.

```
(Near (?car ?d)
  :objective ((KnowLocation ?car ?loc)
              (Distance-to ?loc ?d)
              {< ?d 20}))

 (KnowLocation (?car ?loc)
  :objective ((?car ?loc ?time)
              (recent ?time))
  :means     ((LookForCar ?car ?loc)))
```

Another novel aspect of Icarus is that it provides primitives for reasoning about the world during action selection. (Other fully reactive designs relegate that task to primitive predicates and actions coded in an external language.) Table 4 contains an example from an abstract action for moving 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. Icarus is general enough to support recursive programs.

Table 4: Calculation and recursion within Icarus.

```
(MoveToFastestLane ()
  :requires
    ((CarAheadLeft ?car1 ?speed1)
     (CarAheadCenter ?car2 ?speed2)
     (CarAheadRight ?car3 ?speed3)
     (Fastest ?car1 ?speed1 ?car2 ?speed2 ?c ?s)
     (Fastest ?c ?s ?car3 ?speed3 ?fastest-car))
  :means ((MoveBehind ?fastest-car)))

 (Fastest (?c1 ?s1 ?c2 ?s2 ?c ?s)
  :objective ({>= ?s1 ?s2}
              (bind ?c ?c1)
              (bind ?s ?s1))
  :means
    ((Fastest ?c2 ?s2 ?c1 ?s1 ?c ?s)))
```

Icarus' most unusual feature is that it treats situations and processes in a fundamentally symmetric way. Consider the top-level game plan called RideTheBumberCars in Table 5, which casts processes as objectives. It says, "the goal of Bumper Cars is to cause collisions, and to drive both recklessly and impolitely. If these behaviors aren't accessible, drive laps until they are." This encoding makes intuitive sense since riding bumper cars is all about engaging in an activity. Contrast this with the situation-oriented version of the plan presented in the remainder of Table 5 which is essentially forced to conjure up a metric for spills and thrills to focus agent behavior. This approach also has an unwanted side-effect; the agent will stop playing the game once the target thrill count is achieved.

Table 5: Process and situation oriented versions of a plan.

```
(RideTheBumperCars ()
  :objective ((CollideWithSomeCar)
              (DriveRecklessly)
              (DriveImpolitely))
  :means ((DriveLaps)))

 (GenerateSpills ()
  :objective ((?spill-count ?sc) {> ?sc 10})
  :means     ((CollideWithSomeCar)))

(GenerateThrills ()
  :objective ((?thrill-count ?tc) {> ?tc 10})
  :means     ((DriveRecklessly)))
```

These BumperCar examples raise 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 problem of encoding artificial agents for computer games is more process than state-like in nature, since the behavior of the game agent vs. its goal is paramount.

Concluding Remarks

This paper has introduced Icarus, a tool for constructing artificial agents that pursue cognitively rich tasks in extremely dynamic environments. Icarus provides a rich vocabulary for representing plans that should address currently unmet needs of computer gaming: the hierarchical nature of Icarus plans provides scaleability, while the function call semantics gives Icarus plans a familiar procedural flavor. These two features support the ability to encode nuanced behavior (contrast this with a coding model that employs a flat list of production rules). Finally, Icarus' footprint is quite small; Icarus programs quasi-compile into fixed tree structures, and the interpreter which walks that tree only occupies a few pages of code. Concerning speed, a non-trivial program for controlling a simulated robot executes ~30 times per second on 200 MHz hardware running Icarus in Lisp under Unix (implemented with only a modicum of attention to efficiency). This number includes simulation, display, and inter-process communication delays. A second implementation will improve this picture.

Icarus is, however, a research prototype that has only been tested in a limited number of domains. These include toy problems (pole balancing with process objectives, a reactive version of the Tower of Hanoi) and non-trivial control programs such as 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 application; an agent that controls a vehicle in an extensive traffic simulation. Each of these projects has reinforced our belief that Icarus is a powerful and convenient behavior specification language.

Icarus has the potential to address other key problems in game design. One of our current efforts transforms Icarus from a behavior generator into a recognizer for highly contingent user plans. As a result, Icarus could provide a method of analyzing player intent and increasing the sophistication of automated adversaries. More boldly, we can generalize the notion of an Icarus agent into a game-conductor whose "sensors" detect game events, "actuators" introduce characters, puzzles, or widgets, and whose objective is to orchestrate the story-line towards some conclusion. Icarus' ability to represent a process as an objective provides language level support for this task; the developer can express the game-conductor in terms of actions that enable whole scenarios, and subplots that compose story lines. Icarus may be unique in this regard.

Acknowledgements

We thank Pat Langley, Ross Shachter, and Derek Ayers for their assistance in clarifying this work, and Marcel Schoppers for many discussions of reactive execution.

References

Agre, P. 1988: The Dynamic Structure of Everyday Life, Technical Report, AI-TR-1085, MIT Artificial Intelligence Laboratory.

Brooks, R. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2:1.

Firby, J. 1989. Adaptive Execution in Complex Dynamic Worlds, PhD. diss., Dept. of Computer Science, Yale University.

Georgeff, M., Lansky, A., and Bessiere, P. 1985. A Procedural Logic. In Proc. 9th IJCAI, pp 516ff.

Haigh, K., and Veloso, M. 1996. Interleaving planning and robot execution for asynchronous user requests. In Proceedings of the International Conference on Intelligent Robots and Systems, 148-155.

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

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

Schoppers, M. 1987. Universal Plans for reactive robots in unpredictable environments. In Proc. 10th IJCAI, 1039-1046.

Shapiro, D., and Langley, P. 1999. Controlling physical agents through reactive logic programming. Third International Conference on Autonomous Agents. To appear.

Simmons, R., Goodwin, R., Haigh, K., Koenig, S., and O'Sullivan, J. 1997. A layered architecture for office delivery robots. In Proceedings First International Conference on Autonomous Agents, 245-252.