Very Fast Action Selection for Parameterized Behaviors

Ian Horswill
Northwestern University
2133 Sheridan Road
Evanston, IL
+1 847-612-2765
ian@northwestern.edu

ABSTRACT
In this paper, I will discuss a set of techniques for supporting limited variable binding in behavior-based systems. This adds additional useful expressivity while preserving the property of selecting actions in $O(1)$ time and space. An unusual feature of the method is that it allows inference operations to be reduced to bit-parallel and short-vector operations, making it particularly well suited to real-time operation on modern superscalar and SIMD architectures. I also describe an implementation of the technique in the *Twig* procedural animation system.

Categories and Subject Descriptors
I.2.m [**Artificial Intelligence**]: Miscellaneous

General Terms
Algorithms

Keywords
Behavior-based control; agent architectures; reactive planning.

1. INTRODUCTION
NPC control is one of the fundamental problems in AI for computer games. A number of techniques have been proposed, including finite state machines, decision trees [19], generative planning [37], reactive planning [27], behavior-based systems [22], cognitive simulation [25], and various hybrids. One of the fundamental trade-offs in these systems is between, ***expressiveness*** – the ability to represent goals, actions, and properties of the domain as concisely as possible, ***reactivity*** – the ability to respond quickly to unanticipated changes in the environment, and ***computational complexity***. In general, systems that support more expressive representations tend to be more expensive, and consequently, less reactive, whereas systems that are efficient enough to be rerun on every clock tick tend to be relatively inexpressive.

In this paper, I will discuss the application of role passing [16], a technique originally developed for real-time embedded control on robots, to NPC control in game systems. Role passing is a technique that allows a limited, but useful, subset of Prolog-like reasoning to be compiled to straight-line code (code without branches or subroutine calls) consisting only of loads, stores, and bitmask instructions. The code pipelines well, and so has good performance on modern superscalar architectures. A ruleset consisting of 1000 inference rules containing 5 conjuncts each can be completely rerun to deductive closure on every clock tick at 60Hz using less than 0.1% of a 500MHz Pentium III. Thus, although the technique is limited, any inference expressible in it can effectively be done for free, and kept up to date continuously. While a more general reasoning system may still be desirable, the general system can be made faster by off-loading as much work as possible to role passing.

I will also describe how to generalize the technique to support flexible reasoning about multiple, dynamically prioritized goals by compiling to short-vector operations rather than bitmask instructions. Although these short vector operations are considerably slower than bitmask instructions, they are still fast in comparison to operations such as unification.

Finally, I will describe an implementation in Twig [17], an open-source procedural character animation system designed for interactive narrative applications.

This work takes place in the context of a body of work from the 1980s and 1990s on mapping inference to feed-forward logic networks. Since this work is not well known in the game AI community, I'll begin by reviewing this work, which will hopefully help motivate the approach. However, the reader wishing to skip this background can jump to section 4.

2. BACKGROUND
Suppose we want to have an NPC sit down in a chair. This would be implemented either by playing a fixed animation or using a procedural animation system. In either case, the character would already have to be in an appropriate position. To get to an appropriate position, the character may need to turn around, and/or walk to the chair if they're not already nearby. These would, in turn, be implemented as separate operations. Again, these might be implemented as procedural animations, key framed animations, or mo-cap.

We therefore have three actions that we need to sequence appropriately: going to the chair, facing away from the chair, and sitting down. The important action in this plan is sitting in the chair. The other actions are there only to establish the preconditions of being near the chair and facing away from it.

ICFDG 2009, April 26–30, 2009, Orlando, FL, USA.

We can express this system of goals and preconditions as a STRIPS [12] planning problem as follows:[1]

Action **Sit(x)**
Preconditions: Near(x), FacingAwayFrom(x)
Adds: SittingOn(x)

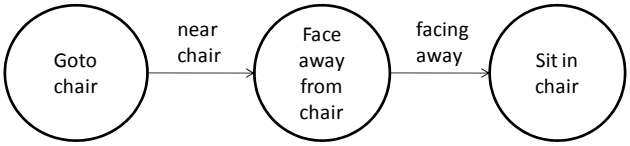
Action **Goto(x)**
Adds: Near(x)
Deletes: FacingAwayFrom(x)

Action **FaceAwayFrom(x)**
Adds: FacingAwayFrom(x)

A solution to the goal SittingOn(chair) would then be the plan:

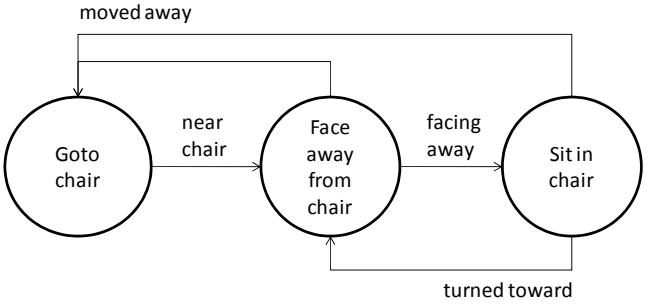
Goto(chair), FaceAwayFrom(chair), Sit(chair)

Which is equivalent to the state machine:



State machines are a common way of sequencing behaviors in games, although they are most commonly hand-coded rather than automatically generated by a planner. They're simple to implement, relatively efficient, and can be authored by non-programmers given the right tools.

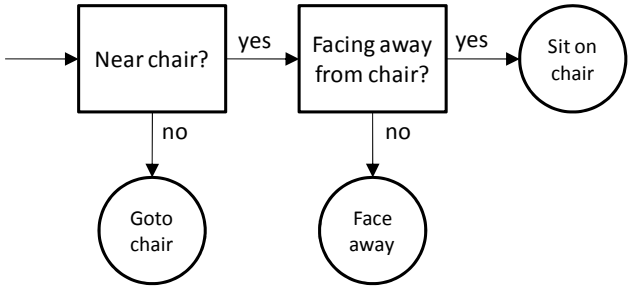
One problem with state machines is that they don't make their underlying goal-and-precondition logic explicit. As a consequence, if an unexpected event happens during execution, such as the character being drawn away from the chair after first moving toward it, the state machine above will still run the turn and sit actions, even though the character is nowhere near the chair. We can solve that by introducing new transitions to initiate recovery actions:



But this leads to cumbersome state graphs, since the number of recovery transitions is potentially quadratic in the number of states. It is also error-prone when done by hand.

[1] In STRIPS planning, the effects of actions are specified by add and delete lists. Any proposition in the add list of an action is guaranteed to be true after execution of the action. Propositions in the delete list may be made false by the execution of the action, although they're not guaranteed to be false. Any other propositions are assumed to be unchanged by the execution of the action.

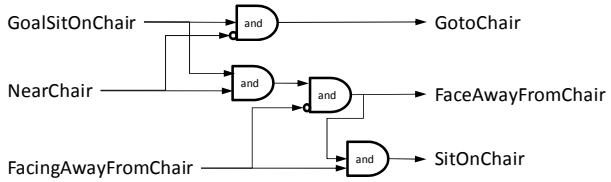
A common solution is to implement the plan as a decision tree:



This technique was first used in Shakey [13], which automatically extracted the equivalent of the decision tree (called a "triangle table") from the plans generated by STRIPS. However, similar approaches can be found in [1, 31, 33]. Production systems such as Soar [23, 24] also have similar behavior. More recently, Halo 2's behavior trees [19], although more general, can also be used in this way.

The advantage of the decision tree approach is that we can reevaluate it on each decision cycle. If a previously established precondition, such as being within range of the chair, gets violated after being initially established, the system will automatically abort its current action and re-fire the goto action to reestablish the precondition, without having to rerun a planner or requiring authors to add explicit recovery transitions.

Another common approach is to achieve the same effect with some type of (simulated) network, the simplest version of which is a logic network [1, 21]. The inputs to the network are the goals of the system (whether we want to sit on the chair) and the relevant percepts (whether we are sitting in the chair, whether we're facing it, etc.), and the outputs are the enable controls for the individual behaviors:



(Note: the small circles at the inputs denote inverters, i.e. "not" gates). Again, this has the advantage of changing its outputs, and hence the selected behaviors, in real time, as conditions change. It also has the advantage of allowing other networks implementing other goals to be run in parallel, and their outputs OR'ed together. In fact, a relatively large subset of propositional STRIPS planning problems can be compiled into logic networks [20, 21], allowing continuous re-planning at frame rate.

Logic networks also have the additional advantage of being fast to simulate, since they can be compiled to straight-line code. The network above can be simulated using the code fragment:

```
GotoChair =
     GoalSitOnChair && !NearChair;
FaceAwayFromChair =
     GoalSitOnChair && NearChair
     && !FacingAwayFromChair;
SitOnChair = GoalSitOnChair && NearChair
     && FacingAwayFromChair;
```

which, again, is cheap enough to rerun on every clock tick of the game.

While I've focused here on the particular example of sitting in a chair, the general problem of selecting behaviors based on goals and preconditions is pervasive. Networks can be extended to handle multiple, prioritized goals by using analog networks, although these are somewhat more expensive to simulate. In that case, the network computes a set of activation strengths for each behavior based on a set of input strengths for each goal, and the system fires the most activated behavior at any given moment. Maes' behavior nets [26], for example, are essentially an analog approximation to propositional STRIPS planning.

3. VARIABLE BINDING

The problem with the network approach above is that it handles the problem of sequencing actions for sitting in one particular chair, rather than for sitting in chairs in general. If we have to worry about multiple chairs in the room, or other kinds of goals or objects, we have to have separate copies of the network for every possible goal and object. What we want is something with the performance properties of a network representation, but the expressiveness of a traditional symbolic reasoning system. To take Prolog [7] as a simple example, we'd like to be able to say something like the Prolog rule:

```
Sit(X) :-
   Near(X), FaceAwayFrom(X), SitDown(X).
```

Meaning that for any X, to sit on X, first get near it, then face away from it, and then sit down on it. If we issue the goal Sit(chair7), then Prolog will search the available rules for ones that match the goal. When it matches the goal to the rule above, it will bind the variable X to chair7, then recursively try to achieve the goals Near(chair7), FaceAwayFrom(chair7), and SitDown(chair7), in order.

Unfortunately, this kind of inference is significantly more expensive than updating an FSM or a logic network. We could try extending logic networks, but networks are notoriously bad at implementing variable binding [35]. Unification, the matching algorithm used in Prolog, is P-complete [11], meaning that it is highly unlikely to be computable by either a realistically sized feed-forward network or by straight-line code.[2]

There are two ways of trying to solve this problem. One is to use a general inference engine for high-level problem solving, but then cache the results in a dependency network. Soar [23] works this way, for example. When Soar first considers a subgoal, it matches rules and binds variables, then caches the results in a *rete* network [14]. Subsequent updates go through the network, with the matcher being reinvoked only when novel subgoals (e.g. new argument values) are encountered. That new matching then grows the *rete* network. Agre's LIFE system [2] worked in a similar manner, but used a Truth Maintenance Systems [10] to build the dependency network. Or the network can be explicitly built as part of the execution of a more general programming language, as with Nilsson's Teleo-Reactive Programming [31]. Finally, one can simply build the network manually. Maes [26], for example, implemented blocks-world stacking in behavior nets by making separate network nodes for every possible set of arguments for every possible predicate. In effect, she built manually the network that a *rete* or TMS system would have eventually built incrementally.

The problem with these approaches is that the networks can get very large in the general case. Maes' blocks-world network, for example required $O(n^3)$ nodes connected by $O(n^4)$ links to represent an n-block world. Hasegawa et al.'s behavior-based dialog system required separate nodes for every possible utterance [15].

The other approach is to look for ways to fit limited versions of variable binding into network formalisms. Generally, these involve placing some set of registers outside the network that the network makes implicit reference to. Agre and Chapman's "deictic representation" (a.k.a. "indexical-functional representation") [1, 2] uses the fact that the human perceptual system can only track a limited number of objects at one time. Each of these trackers then effectively operates as a distinct variable that can be "bound" to an object simply by telling the perceptual to track that object rather than another. They argued that by exporting *all* variable binding to the perceptual system, the central (reasoning) system can be reduced to combinational logic and allowed to update itself on every clock tick. Using this approach they built systems that played the arcade games Pengo [1] and Gauntlet [6] fluently, in real-time, using neurophysiologically plausible (simulated) visual processing on mid-1980s computing hardware (!).

Minsky's Society of Mind uses a similar mechanism in which variable binding is limited to a relatively small number of global variables, which he called "c-lines" [29] and later "pronomes" [30]. Rhodes [5, 32] used a modified version of behavior nets that incorporated pronomes to control characters in a virtual story-telling application.

Finally, there has been considerable interest in modeling the synchronous firing of neurons in the primate cortex as a kind of variable-binding mechanism [9, 34]. To a first approximation, this can be thought of as a time-slicing mechanism in which the same inference network is used for inference about different possible values of a variable at different times. Although this only allows for binding a small number of variables, this is consistent with observations that human working memory is extremely limited [8, 28].

4. FAST VARIABLE BINDING

All the techniques for adding variable binding to logic networks rely on the fact that human working memory is highly limited and so relatively few items need to be kept track of. Although the issue of what kinds of working memory exist in humans and what their capacities are remains controversial, a common rule of thumb is that people can keep roughly seven items in a given memory system at a time [28], although it has been argued that the actual number is much smaller [8].

There's no reason why synthetic characters should necessarily simulate the limitations human working memory, whatever they may be. However, there are many reasoning tasks that have a kind of "locality" property in that they only involve a relatively small set of objects at a given time, regardless of how many objects are actually in the world. Not all tasks have this locality

[2] In fairness, this is for the full version of unification that includes the "occurs check". However, the general point still stands that networks are highly limited in their ability to handle variable binding.

property, but for those that do, we can perform an interesting optimization. We represent the extensions of unary predicates (the set of objects for which the predicate is true) directly as bitsets, then implement inference using bitwise logic operations.

Formally, let U be the set of objects in working memory. Then for any unary predicate, P, we define the set:

$$E_P = \{x \mid P(x) \wedge x \in U\}$$

to be the restriction of P's extension to the set U. We can now convert any set of Horn clause inference rules over unary predicates:

$$\forall x.\left(\bigwedge_{i=1}^{n_1} SL_{i,1}(x)\right) \Rightarrow P(x)$$
$$\forall x.\left(\bigwedge_{i=1}^{n_1} L_{i,2}(x)\right) \Rightarrow P(x)$$
$$\vdots$$
$$\forall x.\left(\bigwedge_{i=1}^{n_k} L_{i,k}(x)\right) \Rightarrow P(x)$$

into an equivalent set operation:

$$E_P = \bigcup_{j=1}^{k}\left(\bigcap_{i=1}^{n_j} E_{L_{i,j}}\right)$$

that computes the entire extension of P at once.

Since the math may be somewhat obscure, let me say what this means at the level of implementation. We represent a unary predicate such as Near(X) using a single machine word in which the *i*th bit is set iff Near is true for the *i*th object in working memory, whatever it might be. We can then implement an inference rule such as the Prolog rule:

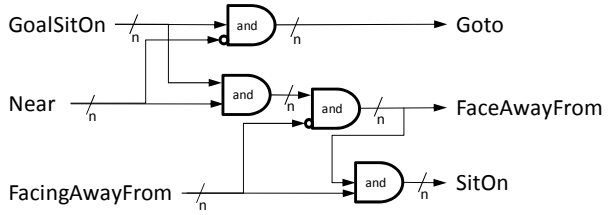
```
P(X):- Q(X), ¬W(X).
```

(i.e. P(X) is true if Q(X) is true and W(X) isn't), as one line of C code:

```
P = Q & ~W;
```

This runs dramatically faster than the Prolog rule, and also solves P(X) for all possible values of X simultaneously. To find the value of P for any particular object, such as chair7, we simply find out which slot chair7 occupies in working memory, and check the corresponding bit in the variable `P`.

Using this technique, we can extend our control network to handle parameterized versions of Near, Goto, etc., by making the inputs and outputs word-sized busses and performing bitwise logic operations:

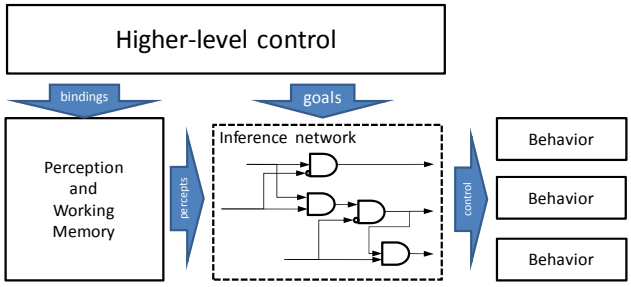


This can then be simulated by the C code:

```
Goto = GoalSitOn & ~Near;
FaceAwayFrom = GoalSitOn & Near
               & ~FacingAwayFrom;
SitOn = GoalSitOn & Near
        & FacingAwayFrom;
```

The behaviors Goto, FaceAwayFrom, and SitOn can then determine whether to run by checking whether their respective variables are non-zero. If so, then they can determine which object is their target by checking which bit is set and finding the object in the associated slot of working memory. We will discuss how to handle the case where multiple bits are set in section 6.

5. CONTROL ARCHITECTURE



As mentioned above, this is a simplified form of role-passing [16], a technique originally developed for robotics applications. It provides us with a modest, but useful, subset of quantified inference that we can implement cheaply; so cheaply that it's effectively free, and so we can afford to rerun the rule set to deductive closure at video frame rate.

In the original version of role passing, variable binding is distributed through a set of different perceptual and memory systems. For games applications, we can abstract these to a single "working memory," whose only purpose is to keep track of which game objects the character's inference network is paying attention to at the moment.

Since the inference network is limited, we assume that some higher level system drives the inference network by loading objects into working memory and asserting the relevant input goals as inputs to the inference network, although for many applications, this may be unnecessary. The inference network then generates as output control signals for enabling or disabling the different primitive behaviors.

6. PRIORITIZED GOALS

One of the advantages of this technique is that it allows the inference network to update all goals and subgoals represented in the network simultaneously. Unfortunately, this raises the possibility that multiple, inconsistent goals may be signaled simultaneously. One case of this is when a single low-level behavior is asked to target two different objects. For example, if the Goto behavior is given the simultaneous goals of going to two different objects at once. Another case is where two different behaviors compete for control of the same resource. For example, if two characters are having a conversation while walking, the Goto behavior will want its character to look forward, while the dialog system will want it to look at the other character. While it's acceptable to switch the gaze between

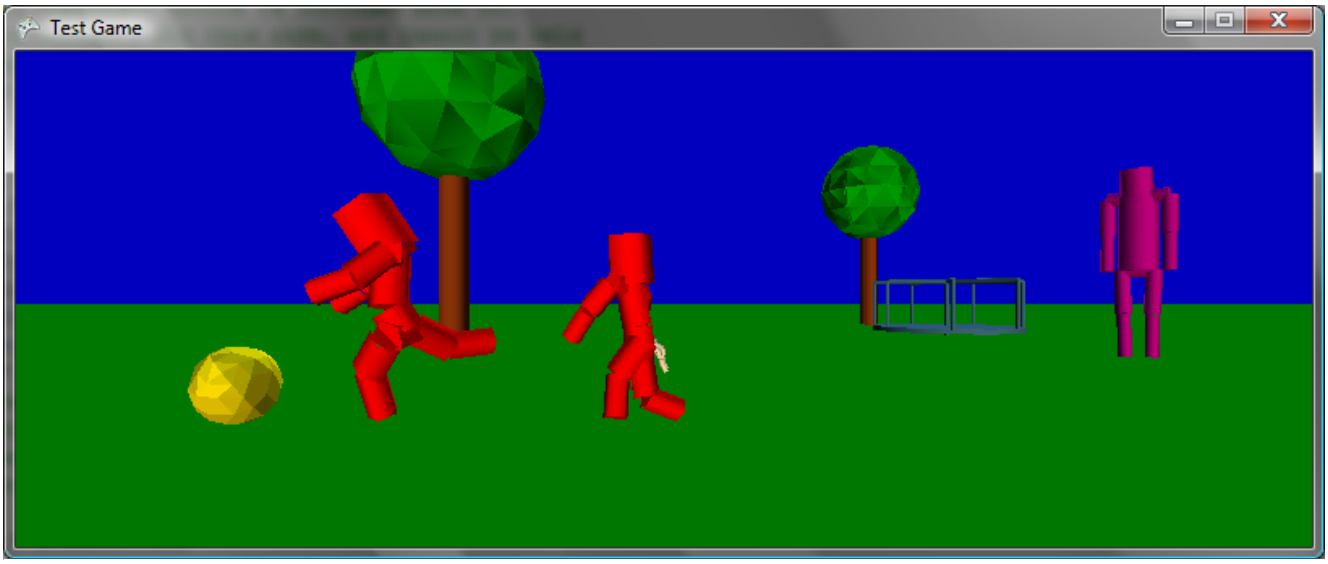


Figure 1: Still frame from Twig

these two targets, neither goal can be allowed to starve out the other. This issue will be discussed in more detail in section 8.2.2.

Because of these issues, it's useful to allow the system to explicitly represent and reason about the relative priorities of different goals. We can do this by representing goals not as bit vectors, but as vectors of floats or small integers. A non-zero value in the *i*th slot of the vector then means that it's a goal to achieve the predicate (or to fire the behavior) on the *i*th object in working memory. The exact number represents the relative strength of the goal. The update code for Goto, FaceAwayFrom, and SitOn is then:

```
for (int i=0; i<WMSIZE; i++) {
  int mask = 1<<i;
  priority_t p = GoalSitOn[i];
  Goto[i] = (mask&~Near)?p:0;
  FaceAwayFrom[i] =
    (mask&Near&~FacingAwayFrom)?p:0;
  SitOn[i] =
    (mask&Near&FacingAwayFrom)?p:0;
}
```

Low-level behaviors like Goto then can then control themselves by finding the object that has the highest priority for them. If no object has non-zero priority for them, they remain inactive.

7. PERFORMANCE

The point of including C code fragments is to make clear that the update rules can run very, very fast. Ordinary inference, such as the code for the non-prioritized goal case of the sit controller compiles to 12 machine instructions on a generic load/store architecture (4 AND instructions, 3 loads, 3 stores, and 2 NOT instructions). And since it contains no branch instructions, it will pipeline so long as the loads don't trigger cache misses. Thus the whole update will cost less than the entry/exit sequence for a typical procedure call.

Updates for priority vectors for prioritized goals will be slower, but still relatively fast, assuming working memory is relatively small. They have good cache coherency, and in a few special cases, such as setting the priority of one goal to be the sum of the priorities of two others, they can be implementing using SIMD instructions.

8. ROLE-PASSING IN TWIG

Twig [17] is an open-source procedural animation system intended for interactive narrative applications. Built on XNA and C#, it provides simple physics, perception, and behavioral capabilities for stylized humanoid characters. Characters can be driven either through behavior-based AI techniques [3] running inside the *Twig* engine, or they can be controlled by an external planner or reactive planner using a remote-procedure call interface.

Figure 1 show an example of three characters, two children and a parent from a simulation of attachment behavior in human children [4]. The children play with a ball and merry-go-round, and periodically fight if they get in one another's way. The smaller child gets anxious when too far away from the parent. The child will then periodically pause to glance back at the parent, and will eventually run back to hug the parent to relieve the anxiety, after which the child can return to play.

8.1 Implementation

Previous implementations of role passing were implemented with a custom compiler [18] that could do elaborate inlining tricks and other optimizations. However, both because the XNA build system only supports C#, and also to protect prospective Twig users from having to learn an obscure research language, role passing in Twig is implemented directly in C#. This makes it less efficient, but more accessible to causal users.

8.1.1 Working memory

Each Twig character maintains a working memory of world objects. The current implement has 7 slots in working memory. In principle, if a large number of tasks involving different objects were active concurrently, the working memory could be overloaded, however this hasn't yet been an issue. Working memory could also easily be increased, since role passing doesn't rely on any particular bound on memory size, just that there be a bound.

Objects can be loaded into working memory by the perceptual system, the RPC system, or the script interpreter. Most users will use the system via RPC control or scripting. The working memory and role passing are then invisible to the user. When the user issues a command such as "goto table" the interpreter loads its arguments into working memory and asserts the relevant goals in the network. Although loading an object into working memory requires a linear search of working memory, it only occurs once per RPC call, and in any case is negligible compared to the overhead of an RPC call.

Objects can also be loaded into working memory by the perceptual system. The perceptual system does a regular scan of the environment, performing a simple appraisal (currently valence and monitoring priority) of each object in view based on the character's current goals and affective state, similar in some ways to the motive and advertisement systems in *The Sims* [36]. For example, in the scenario of figure 1, the child characters are interested in playing with the ball, parent character is interested in the children, and the smaller child's interest in the parent varies depending on its anxiety level.

Working memory is currently managed with a least-recently-used replacement policy. When new items arrive, the item least recently used as the target of a primitive behavior (gaze, locomotion target, etc.) is removed and replaced by the new item. More elaborate strategies are possible, but LRU has been effective so far.

8.1.2 Inference and control

Twig provides a set of three C# classes, `Predicate`, which implements a unary predicate over working memory objects, `Function<T>`, which is generic type implementing a function from working memory objects to type `T`, and `GoalAssertion`, a subclass of `Function<float>` that tracks the goal strengths of a predicate.

Predicates and functions both have update procedures that are automatically run at the start of a clock tick. Sensory predicates build their bit-masks by iterating over the objects in working memory and testing them for whatever property the predicate represents. Other predicates can compute their values by applying logical operators to other predicates using overloaded versions of `&`, `|`, and `~`. Predicates also have two special fields, `Goal`, which contains the GoalAssertion for this predicate, and `Preconditions`, which contains an ordered list of other predicates that are preconditions to achieving the predicate as a goal.

GoalAssertions provide operations for specifying the predicates goal strength for a given object, and for querying to find the object in working memory with maximal goal strength. Once a goal is asserted for a predicate, it automatically passes activation to the GoalAssertions of its unsatisfied preconditions. When a predicate is a precondition for more than one other predicate, its goal strength is summed across the predicates for which it's a precondition.

Low-level (leaf) behaviors then determine whether to run and what object to target by finding the maximum-strength goal object for the predicate they achieve. If the strength is non-zero, then it tries to run, although in some cases it may be overridden by a competing behavior with higher strength.

8.2 Principal uses in Twig

Role passing is an inference mechanism. It allows a system to quickly compute the values of unary predicates for objects in working memory based on the values of other predicates. In practice, this is most useful in Twig for keeping track of goals within the system and choosing which goals to solve when there are conflicts.

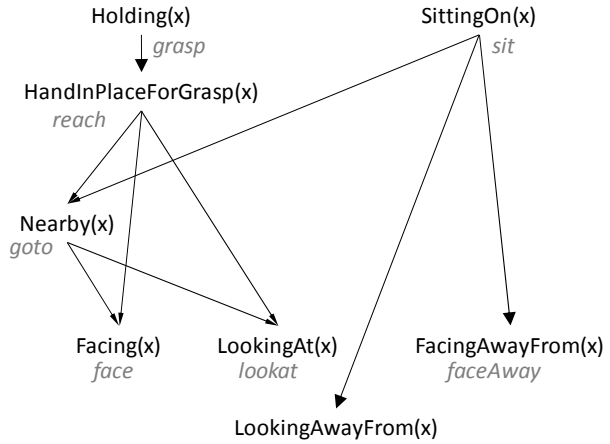


Figure 2: Simplified subgoal structure for holding and sitting, with associated task-achieving behaviors labeled in *gray italics*. Note that LookingAwayFrom has no task-achieving behavior per se, but is implemented by propagating a negative goal strength for LookingAt.

8.2.1 Subgoal sequencing

The simplest case of this is the kind of subgoal tracking discussed in section 2. Most actions – sitting in a chair, grasping an object, throwing a ball, etc., have preconditions that must be satisfied before the action is fired: grasping requires the hand be in position, but reaching requires the body be in position, which requires facing the object, etc. (see Figure 2). Sitting, as discussed above, requires first establishing some preconditions (facing, looking at), and then later establishing their negations – it's kinematically impossible to sit in the chair while facing it, and the character looks silly if it stares at the chair while sitting in it. Role passing is an inexpensive mechanism for tracking preconditions, firing behaviors, and automatically re-firing them as necessary when preconditions are clobbered.

8.2.2 Resource allocation

Sequencing behaviors in Twig is complicated when there are multiple goals running concurrently. A Twig character can control the facing direction of its head, torso, and pelvis independently, modulo the kinematic limits of the body. When approaching an object, a character needs its pelvis to face the object. It also needs to have its gaze *mostly* directed toward the object. However, the character can and should continue to switch its gaze to other objects of interest, such as nearby obstacles, so long as its gaze isn't turned from the target for too long. If the character's gaze does move from the target for too long, the character should slow down or stop until the gaze returns. In point of fact, the system doesn't have a LookAt action *per se*; the gaze controller is an autonomous process that

runs in parallel with the other behaviors, arbitrating their gaze requests, trying to insure that each object of interest gets enough attention at the right times. Thus gaze direction, posture, and so on are effectively shared resources that must be dynamically allocated to competing behaviors.

Preplanning this allocation is difficult because tasks and contingencies are highly dynamic. However, simple reactive schemes can lead to deadlock or livelock. For example, when a character is being chased, the locomotion behavior wants to look at the destination it's running to, but the gaze system also wants to look back at the chaser periodically. Unfortunately, looking all the way behind the character requires rotating he torso, which in turn requires a small pelvic rotation. If gaze control and locomotion are not properly coordinated, the locomotion controller won't release the pelvis and the gaze controller won't be able to complete the look back. But then the gaze controller will never return the gaze forward, and the locomotion controller will eventually stop running. The result is deadlock; the character stands motionless, not completing either operation.

A better way to handle the situation is to have the relevant behaviors make prioritized requests for bodily resources to a higher-level arbiter that that can make intelligent decisions based on all the requests. Role passing essentially provides a convenient bookkeeping mechanism for tracking the different demands on a resource and updating them at frame rate (60Hz).

8.2.3 Implementation status

Twig was originally implemented without role passing, and so the low-level behaviors had to rely on some higher-level system to track and establish preconditions. Because this was inconvenient and because of the resource contention and deadlock issues discussed above, we decided to retrofit role passing into the system. Since the system was already organized around a working memory, it was a natural fit. As of this writing, the locomotion, gaze control, facing, and sitting systems have been converted to role passing. The role-passing control for reaching and grasping is in place, but the low-level reaching and grasping behaviors themselves are in the process of being refactored, after which the control logic will likely need to be revised.

9. DISCUSSION

Role passing allows a limited, but useful, subset of Prolog-like reasoning to be compiled to extremely efficient code and constantly updated with each clock tick. It provides a useful intermediate architecture between classical behavior-based systems and reactive planners, extending the expressiveness of the former without significant performance penalties.

That said, role passing does have a number of limitations. The most obvious is the limitation to reasoning only over the objects in short-term memory. Although for worlds limited to a few hundred objects, one could theoretically remove this restriction and just use larger bit vectors, in practice it is probably more reasonable to use a different technique for more general reasoning and save role passing for reactive reasoning about the objects in the character's working set.

Another limitation is the restriction to unary predicates (predicates with a single argument). While in principle it could be extended to handle binary or ternary predicates, representing the extension of an n-ary predicate requires s^n bits, where s is the size of short-term memory, which is infeasible for large values of s. The last main restriction is that it doesn't handle term expressions. However, since these are more commonly used in conjunction with binary or ternary predicates, this a less common complaint than the lack of binary predicates.

Role passing is not a panacea. However, it can be used to significantly improve the performance of characters by allowing low-level behaviors perform more sophisticated run-time coordination, while still maintaining reactivity and efficiency. More general systems, such as planners, reactive planners, and production systems, can still be used in conjunction with role passing. And at least in the case of Twig, most users will likely elect to do so.

Again, the argument here is not that "smart" systems like generative planners are bad; far from it. But to the extent that even the smart systems generally rely on a network of "dumb" behaviors for moment-to-moment control, there's no reason not to make those behaviors smarter if we can do so at little or no cost. Doing so can only make the system more reliable and responsive.

10. REFERENCES

[1] Agre, P. and Chapman, D. 1987. PENGI: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)* (Seattle, WA, 1987). AAAI Press.

[2] Agre, P. E. 1988. *The dynamic structure of everyday life*. Technical Report 1085, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[3] Arkin, R. 1998. *Behavior-Based Robotics*. MIT Press, Cambridge, 1998.

[4] Bowlby, J. 1969. *Attachment and Loss*. Basic Books, New York,, 1969.

[5] Bradley, R. 1997. PHISH Nets: planning heuristically in situated hybrid networks. In *Proceedings of the Proceedings of the first international conference on Autonomous agents* (Marina del Rey, California, United States, 1997). ACM.

[6] Chapman, D. 1990. *Vision, Instruction, and Action*. Massachusetts Institute of Technology, 1990.

[7] Clocksin, W. F. and Mellish, C. S. 2003. *Programming in Prolog: Using the ISO Standard*. Springer, New York, NY, 2003.

[8] Cowan, N. 2001. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24 (2001), 87-185.

[9] Crick, F. and Koch, C. 1990. Towards a neurobiological theory of consciousness. *Semin. Neurosci.*, 2 (1990), 263–275.

[10] Doyle, J. 1979. A Truth Maintenance System. *Artificial Intelligence*, 12, 3 (1979), 251-272.

[11] Dwork, C., Kanellakis, P. and Mitchell, J. C. 1984. On the sequential nature of unication. *Journal of Logic Programming*, 1, 1 (1984), 35-50.

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

[13] Fikes, R., Hart, P. and Nilsson, N. 1972. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3 (1972), 251--288.

[14] Forgy, C. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19 (1982), 17-37.

[15] Hasegawa, T., Nakano, Y. I. and Kato, T. 1997. A collaborative dialogue model based on interaction between reactivity and deliberation. In *Proceedings of the Proceedings of the first international conference on Autonomous agents* (Marina del Rey, California, United States, 1997). ACM.

[16] Horswill, I. 1998. Grounding Mundane Inference in Perception. *Autonomous Robots*, 5 (1998), 63-77.

[17] Horswill, I. 2008. Lightweight Procedural Animation with Believable Physical Interactions. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment* (Stanford, CA, 2008).

[18] Horswill, I. D. 2000. Functional Programming of Behavior-Based Systems. *Auton. Robots*, 9, 1 (2000), 83-93.

[19] Isla, D. 2005. Handling Complexity in the Halo 2 AI. In *Proceedings of the Game Developer's Conference 2005* (San Francisco, CA, USA, 2005). CMP, Inc.

[20] Kaelbling, L. P. 1988. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)* (St. Paul, MN, USA, 1988). AAAI Press.

[21] Kaelbling, L. P. and Rosenschein, S. J. 1990. Action and planning in embedded agents. In *Designing Autonomous Agents*, P. Maes, Ed. MIT Press, Cambridge, MA, 1990.

[22] Khoo, A., Dunham, G., Trienens, N. and Sood, S. 2002. Efficient, Realistic NPC Control Systems using Behavior-Based Techniques. In *Proceedings of the AAAI Spring Symposium Series: Artificial Intelligence and Interactive Entertainment* (Stanford, CA, 2002). AAAI Press.

[23] Laird, J. E., Newell, A. and Rosenbloom, P. S. 1987. Soar: An Architecture for General Intelligence. *Artificial Intelligence*, 33 (1987), 1-65.

[24] Laird, J. E. 2001. It knows what you're going to do: adding anticipation to a Quakebot. In *Proceedings of the Fifth International Conference on Autonomous Agents* (Montreal, Quebec, Canada, 2001). ACM.

[25] Laird, J. E. and Lent, M. v. 2001. Human-level AI's killer application: interactive computer games. *AI Magazine*, Summer 2001 (2001), 15-25.

[26] Maes, P. 1989. How to do the right thing. *Connection Science Journal*, 1 (1989), 291-323.

[27] Mateas, M. and Stern, A. 2002. A Behavior Language for Story-Based Agents. *IEEE Intelligent Systems*, 17, 4 (2002), 39-47.

[28] Miller, G. A. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63 (1956), 81-97.

[29] Minsky, M. 1977. Plain talk on neurodevelopmental epistemology. In *Proceedings of the Fifth International Joint Conference on Articial Intelligence,* (Cambridge, MA, USA, August, 1977).

[30] Minsky, M. 1986. *The Society of Mind*. Simon and Schuster, New York, NY, 1986.

[31] Nilsson, N. 1994. Teleo-Reactive Programs for Agent Control. *Journal of Artificial Intelligence Research*, 1 (1994), 139-158.

[32] Rhodes, B. 1995. *Pronomes in Behavior Nets*. Technical Report 95-01, MIT Media Laboratory, Cambridge, MA, USA, 1995.

[33] Schoppers, M. J. 1987. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of the International Joint Conference on Artificial Intelligence* (Milan, Italy, 1987). Morgan Kaufmann.

[34] Shastri, L. and Ajjanagadde, V. 1993. From simple associations to systematic reasoning: A connectionist representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioral and Brain Sciences*, 16 (1993), 417-494.

[35] Smolensky, P. 1990. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46, 1-2 (1990), 159-216.

[36] Wright, W. 2000. *The Sims*. MAXIS/Electronic Arts, City, 2000.

[37] Young, R. M. and Riedl, M. O. 2005. Integrating plan-based behavior generation with game environments. In *Proceedings of the Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology* (Valencia, Spain, 2005). ACM.