Tagged Behavior-Based Systems: Integrating Cognition with Embodied Activity

Ian Horswill, *Northwestern University*

This article focuses on tagged behavior-based systems—supporting a large subset of classical AI architectures while allowing object representations to remain distributed across multiple sensory and representational modalities.

Automated reasoning systems are typically built on a transaction-oriented model of computation. Knowledge of the world is stored in a database of assertions in some logical language, indexed perhaps by predicate name.[1] When the system is given a query such as "Are there any blood disorders with symptoms that affect the gastrointestinal tract?" the system might translate the query from natural language into a logical assertion such as "There exist X and Y such that X is a blood disorder, X has symptom Y, and Y involves the gastrointestinal tract." It then answers the question by proving or disproving the assertion. The backtracking control structure used by a logic-programming engine to check the assertion amounts to a series of nested loops:

```
for each X such that blood-disorder(X) appears in
      the database
   for each Y such that symptom(X, Y) appears
         in the database
      for each Z such that involves(Y, Z) appears
            in the database
         if Z=gastrointestinal-track
               then return true
return false
```

Of course, this is a simple case of automated reasoning; it doesn't really use much in the way of inference rules. However, even simple examples like this can be extremely problematic to implement in a robot. Suppose we ask the robot, "Are there any predators here that could eat you?" (ignoring the issue that robots aren't particularly digestible). That query would translate into the sentence "There exists X such that X is nearby, X is a predator, and X can eat me." Again, a reasoning system attempting to verify this sentence would need to perform a search that would amount to a set of nested loops:

```
for each X such that nearby (X) appears in the
      database
   if predator(X) appears in the database
      then for each Y such can can-eat(X,Y)
            appears in the database
         if Y=me, then return true
return false
```

The problem occurs when we ask how the database is filled to begin with. Unlike blood disorders, the set of objects near the robot is in continual flux. The only way the robot can know about them is to direct its sensors and sensory processes toward the objects and measure relevant properties. However, automated reasoning systems typically have no good way of directing perceptual attention. They either assume that all relevant information is already stored in the database or they provide a set of epistemic actions that fire task-specific perceptual operators to update specific parts of the database. The former approach requires that the perceptual systems run some expensive loops in parallel with the reasoning system:

```
for each object X in view
   for each property P of X
      measure P(X)
      retract the old value of P(X) from
         the database
      assert the new value
   for each other object Y in view
      for each spatial relation R
         determine whether R(X,Y) holds
            update the database
               accordingly
            … etc…
```


1094-7167/01/$10.00 © 2001 IEEE

The epistemic action approach requires that the programmer design the rule base to ensure that the appropriate actions are fired at the appropriate times. This is more complicated than it might seem. Epistemic actions not only need to be fired when information in the database is missing but also when the information is out of date. In effect, the database is functioning as a cache for sensory data. As with any cache system, it must be kept coherent with the rest of the system—in this case, the sensory systems—and more importantly, the external world.

This cache coherence problem, which I call the *model coherence problem*, is an instance of a more general problem in robot design. Real robots consist of a large number of sensory, motor, and reasoning processes operating in parallel, on separate processors, and often on simple processors with little or no operating system. Each of these processors has, in some sense, its own limited model of the world and the robot's current task. All these models must be kept consistent with one another and the external world. This cannot be done by fiat nor, I would argue, can it be done as an afterthought. It must be designed as a central architectural tenet of the system.

In this particular case, however, we can solve the problem by making the perceptual system emulate the database. A typical automated reasoning system implements two functions, **Ask** and **Tell**, which query and modify the database by performing the appropriate database searches. The database typically implements simpler versions of these operations. For queries, these operations often amount to enumerating the set of variable bindings that match a given literal—an application of a predicate to a set of argument expressions. These argument expressions may or may not contain variables. When the literal contains no variables, the database need only answer whether it appears in the database. When it does contain variables, the database needs to act as an abstract iterator, generating a series of values for the variables until either a set of values is found that make the sentence true or all possible values are exhausted, in which case the sentence is false.

Fortunately, these sorts of enumeration operations are relatively straightforward to implement directly in the perceptual system. When the reasoning system asks to enumerate the variable bindings of **red**(X), the database can defer to the perceptual system. The perceptual system allocates a color tracker, records the fact that it is "bound" to the variable X, and looks for a red object. When the reasoning system asks for the next binding of X, the perceptual system redirects the tracker to a new red object.

The Bertrand system

The Bertrand system uses exactly this technique. Bertrand is a database-free logic programming system that answers blocks–world queries using real blocks and a real-time vision system.[2] Bertrand uses an implementation of current theories of human intermediate-level vision to search the scene for specified configurations of colored blocks.[3] Enumeration operations in Bertrand are handled by a *visual routine processor*, a specialized vision computer whose registers were object trackers and whose instruction set consisted of these operations:

- Bind a specified tracker to an object of a specified color.
- Bind a specified tracker to a *different* object of its specified color that has not already been searched.
- Test a specified tracker to determine its color.
- Bind a specified tracker to an object beneath–beside–above the object tracked by another specified tracker.
- Bind a specified tracker to a different object beneath–beside–above that has not already been searched.
- Test whether the objects tracked by two trackers lie in a specified spatial relationship.

A conventional compiler for a logic programming language might compile the query "**blue**(X), **above**(X,Y), **red**(Y)" (that is, "Is there a blue object above a red object?") into

```
find the first instance of blue() in the
    database
bind X to its argument
repeat
    find the first instance of above(X, …)
        in the database
    bind Y to its second argument
    repeat
        if red(Y) appears in the database,
            return true
        find the next instance of above
            (X, …) in the database
        rebind Y to its second argument
    until no more instances of above
        (X, …)
    find the next instance of blue(…) in
        the database
    rebind X to its argument
until no more instances of blue(…)
```

Bertrand compiled the same query into a series of *visual* operations:

```
find a blue patch in the image and bind
        tracker 1 to it
repeat
    find a colored patch under the region
        tracked by tracker 1
    bind tracker 2 to it
    repeat
        if tracker 2 is tracking a red patch,
            return true
        find the next colored patch beneath
            tracker 1
        rebind tracker 2 to it
    until the bottom of the image is reached
    rebind tracker 1 to another blue patch
until no more blue patches
```

This allowed Bertrand to execute queries without the need for separate epistemic actions because the inference operations *were* epistemic actions. Figure 1 shows the execution of this query under Bertrand.

The Ludwig system

Ludwig is a simple natural-language question answering system based on the same approach.[4] It can answer simple queries involving colors and spatial relations, such as "Is there a blue block on a red block on a yellow block?" or "Is the block on the yellow block blue?" Although syntactically and semantically simple-minded—the only words it pays attention to in "Where is the blue block?" are "where," "is," and "blue"—it's unusual in that it's directly grounded in real-time vision.

What makes Ludwig most unusual is that it uses a behavior-based architecture—it consists of a parallel network of communicating finite-state machines similar to Brooks' subsumption architecture.[4] Instead of having a single, centralized representation, information about an object was distributed between different specialized representational mechanisms (see Figure 2).

Ludwig's parser consisted of a pipeline of finite-state machines built from (simulated) logic gates and latches. Each word from the user's query is presented to the parser as it is typed, one word per cycle. It compiles an adjective or preposition into a visual routine processor instruction; "blue" compiled into

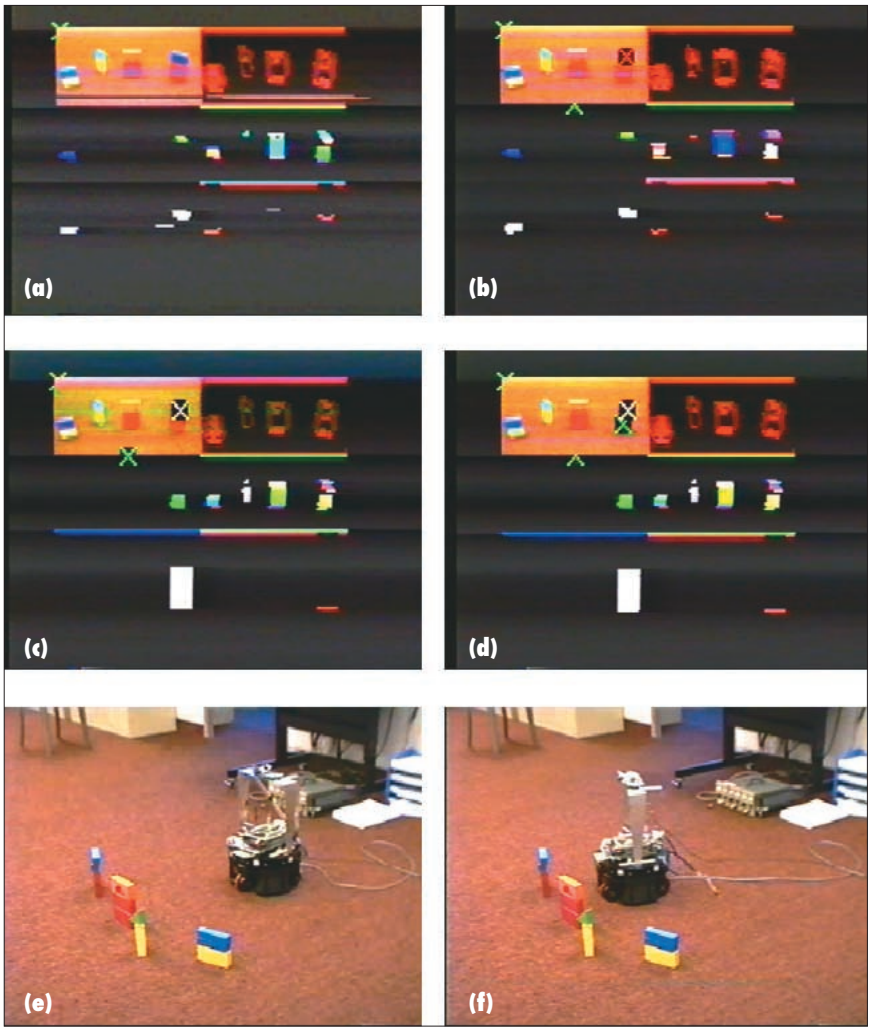

Figure 1. The Bertrand system searches for a blue block stacked on a red block using the query blue(X), on(X,Y), red(Y). Each screen image shows the original NTSC camera image, the edges detected in the image, the color blobs extracted from the image, the salient pixels given the current search criteria, and the selected most salient blob. In frame (a), the system searches for a blue block and finds two blue objects. In frame (b), it happens to select the correct blue object and binds a tracker to it (the red X). In frame (c), it searches for an object underneath the selected object by highlighting all points immediately beneath it as salient. Only one object is in the salient region. In frame (d), another tracker is bound to it (green X). If the system had chosen the wrong blue object in the first step, it would have backtracked when the search for a red object underneath it failed. In frames (e) and (f), the robot's external view is shown as it executes the query and drives to the chosen blocks.

"place tracker N on a blue object," for example, while "on" was compiled into "place tracker M on the object immediately below tracker N in the image." The parser latches the instruction into a shift register, gradually accumulating a visual routine processor program (represented within the shift register) that searches for the referent of the noun phrase.

In parallel with this semantic analysis, another set of finite-state machines performed a simple syntactic analysis to find phrasal boundaries. Verbs (of which Ludwig understands only "is") and the end of an utterance signified phrasal boundaries. However, a noun followed by an adjective, as in "is the block on the blue block • red?" also signals a boundary between two noun phrases. When the parser encounters a phrasal boundary, it marks the current shift register completed and switched to a new shift register. As the new noun phrase is parsed, the visual routine processor automatically begins searching for the referent of the previous phrase using the completed shift register. Thus, semantic analysis, syntactic analysis, and visual processing all occur in a pipelined, parallel fashion.

Ludwig keeps track of the relationship between the programs, stored in shift registers, that represent the description of an object, and the visual trackers that represented its position and identity, using a system of associative *tagging*. Object representations, both descriptions (programs) and trackers, are tagged with the names of the objects they represent. For example, in the command "face the green block on the blue block" (Ludwig has a very limited motor control capability), the theme of the sentence is "the green block on the blue block." The description of the theme,

```
green(Theme), on(Theme, X), blue(X)
```

is stored in a shift register tagged Theme. When the control system needs the referent of the theme, it tells the visual routine processor to run the Theme program, knowing that the referent of the description will be tracked by whatever tracker is bound to Theme. The VRP executes the program tagged Theme, which involves allocating trackers and tagging them Theme and X, and directing them toward their intended referents. On completion of the program, the top-level control system then tells the motor control system to servo toward the Theme. To do this, it sends the tag Theme to the turn behavior. The behavior then forwards it to the trackers, and whichever tracker is tagged Theme responds with the coordinates of the object.

Tagging provides an alternative mechanism for coordinating the different representations of an object. Rather than copying all the data to a centralized database or passing complicated symbolic expressions between system components, components communicate by passing simple tags. Tagged behavior-based systems preserve the simplicity, parallelism, and efficiency of traditional behavior-based systems while providing additional flexibility and programmability.

Forward-chaining inference for control

While the Bertrand–Ludwig architecture works well for question answering, it doesn't work as well for controlling action. Being

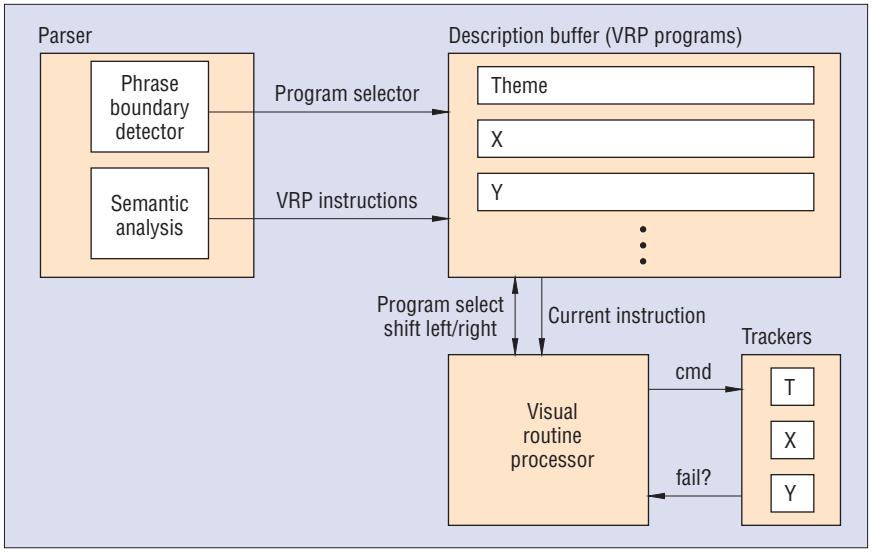

Figure 2. The Ludwig system's architecture. The parser compiles noun phrases to visual routine processor programs to find their referents and loads them into shift registers in the description buffer. The VRP opportunistically runs these programs in parallel with the parsing process to bind trackers to the referents of the query's noun phrases.

a backward-chaining inference engine, its control structure is necessarily top-down and serial. If the scene changes as the robot scans it, the robot won't notice because it only attends to the particular property or spatial relation it's measuring at the moment, and it doesn't keep any sort of dependency graph to link premises to their conclusions. Without a dependency network, conclusions can't be retracted when their premises change; the model doesn't *track* the changes in the environment. This is a problem for control applications. If we build a package delivery system using serial backward-chaining, the system can infer that it needs to pick the package up before driving to the destination, but if the package falls out of the gripper in route, it might not even notice. Current reactive planners often require the programmer to explicitly program when to check these sorts of sensory conditions. When unanticipated changes occur, the robot's behavior can be quite pathological.

A forward-chaining inference system is needed, one where any computed inference is continually recomputed, or at least recomputed whenever its antecedents change. Compiling forward-chaining propositional inference into parallel networks is relatively simple.[5] For every proposition in the system, we have a wire that holds its truth-value. An axiom such as facingObject and objectNear ⇒ objectGraspable, is implemented by connecting an And gate to the facingObject and the objectNear wires and using its output to drive the objectGraspable wire.

This approach breaks down when implementing quantified inference (inference with variables). To implement an inference rule such as facing(X) and near(X) ⇒ graspable(X), we need an infinite number of wires, one for each possible value of X, and an infinite number of And gates. This is not a purely theoretical problem; it is a general problem of connectionist and behavior-based control systems that represent predicate–argument structure in fixed, parallel networks. In this situation, behavior-based systems often use multiple copies of behaviors to handle all their possible arguments. One behavior-based dialog system, for example, uses a separate behavior for each possible utterance of each speaker.

Fortunately, tagging gives us a partial solution to this problem. In a tagged behavior-based architecture, there is a fixed set of tags that is used to identify the objects referred to by a given tracker or memory representation. If a system's inferences are restricted to the set of objects actually represented in working memory, then we can represent a predicate graspable with the set of tags for which it's true. Assuming the set of tags is relatively small, we can represent a tag set using a fixed vector of bits, one bit per tag. The predicate is true of a given object if and only if the bit position corresponding to its tag is set to one. If the number of tags is manageable (we presently use about 20), then this still allows for a relatively compact network. We simply take the original propositional network and replace individual wires with 20-bit busses, or time-multiplex a single network.

Of course, no one actually builds parallel hardware like this. Real robots are built from loosely coupled networks of microcontrollers, PCs, and other serial processors, not from custom parallel hardware. Fortunately, tagging is also very efficient on conventional serial processors. Instead of using a 20-bit bus to represent a predicate, we use a single 32-bit machine word. The graspable rule above can then be compiled to a simple assignment statement, written here in C++:

```
graspable_tags = facing_tags&near_tags;
```

If, for example, all rules are Horn sentences (implications with conjunctions on the left), then we can compile all the rules into a series of such assignment statements and reexecute them on every cycle of the system's decision loop, effectively recomputing the entire knowledge base on every cycle. While this sounds expensive, it is cheap enough that it is effectively free. Because each rule is compiled to a small number of load, store, and bit-mask instructions, it's possible to run 1,000 rules at 100 Hz and still use less than one percent of a modern CPU.

Tagging is only a partial solution to the problem of representing predicate–argument structure. It doesn't support term expressions. It's limited to reasoning over the set of objects to which the robot is presently attending. And it is also effectively limited to unary (signal argument) predicates; if there are n tags, then binary predicates would require n^2 wires, ternary relations, n^3 wires, and so forth. However, tagged architectures are sufficient to do most of the kinds of reactive reasoning that robots do today, only considerably more efficiently.

Role passing

We've used this feed-forward tagging scheme to program a robot, Kluge, to follow simple natural language commands, such as "get the green ball," "go to the blue ball," and "follow me."[6] Kluge uses a custom 25-MIP DSP board with an attached frame grabber (the DIdeas Cheap Vision Machine) and video camera. The electronics are housed on a commercial synchrodrive base (a Real World Interface B14S). The robot can track up to three colored objects simultaneously and performs visual navigation at about 1m/s. The vision system runs at 10 frames per second, and the inference engine and motor control behaviors completely update themselves on every frame.

Kluge can represent objects in three different modalities (see Figure 3). It can remem-

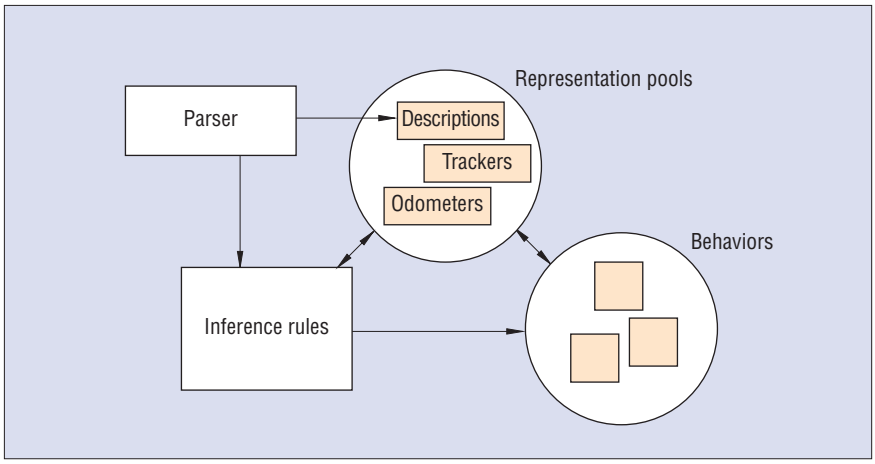

Figure 3. Role-passing architecture used on the Kluge robot. Nearly all data passed between subsystems is in the format of tag sets (roles).

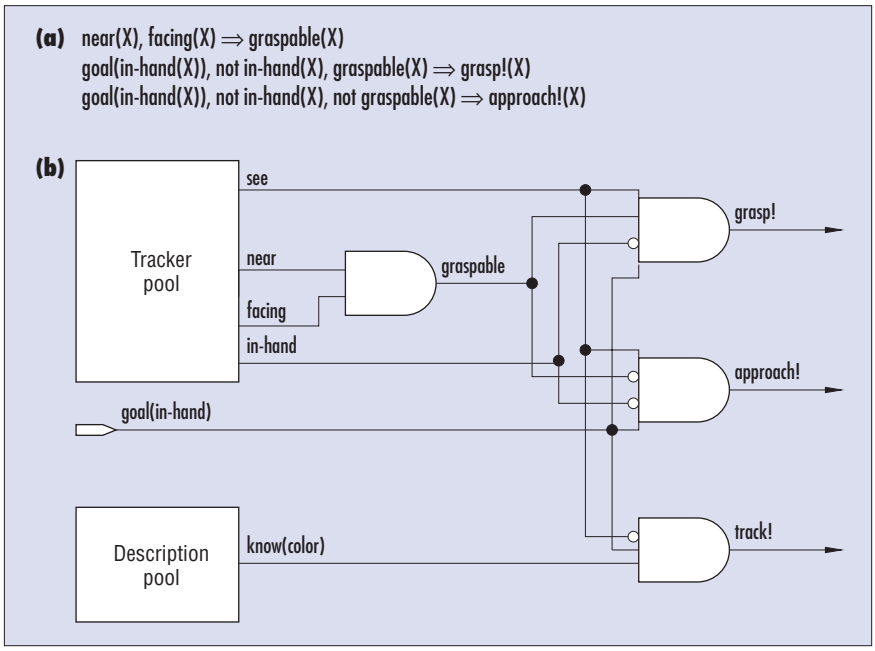

Figure 4. (a) Simplified set of inference rules to control the firing of different grasping behaviors. (b) An equivalent control circuit. Note that each wire is really a bus with one bit per role and that logic operations are bitwise Ands and Nots on these bit-vectors. Data flows from the goal input, the description pool, and the vision system, and its outputs drive the grasp, approach, and visual tracking behaviors. Inferences are recomputed with each successive frame of the video stream.

ber what color it is by tagging one of a set of color representations held in its description pool. Given the color, it can allocate a visual tracker from its tracker pool, assigning it an appropriate tag, and set it searching for that color. And when the object goes out of view, it can still keep a rough idea of its location using an odometric tracker pool.

As with Ludwig, Kluge uses a finite-state parser whose output is a set of tag bindings in the other components of the system, specifically the representation pools. In this system and subsequent systems, we have adopted the practice of using linguistic role names such as Agent or Patient as our tags. So when the user types "get the green ball," the parser binds the role **Object** to the color green in its pool of color descriptions and asserts that **in-hand(Object)** is a goal.

Most of the communication between the parser, inference engine, and peripheral systems consists of passing sets of linguistic roles in this bit-vector representation. On every control cycle, the vision system grabs a new frame and processes it. The different tracking systems independently determine whether their targets are nearby. The vision system Ors together the roles (represented as bit-vectors) of all the trackers with nearby targets. The result is the extension of the predicate near, which it passes to the inference engine. Similarly, it Ors together the roles of all tracked objects it is facing to form the facing predicate. In parallel, the odometry system, which tracks the locations of objects using dead reckoning, determines which of its targets are nearby or faced by the robot, and Ors their tags into the bit-vectors representing near and facing. These bit-vectors are cheap to compute compared to the cost of doing the tracking in the first place, so they can be recomputed on every decision cycle.

Kluge then uses a set of inference rules to control the firing of different behaviors. A simplified version of the rules used to control grasping and approaching objects is shown in Figure 4.

These rules are transformed into the feedforward logic network shown in Figure 4. (In the real system, **grasp!** isn't an atomic behavior. Separate rules are used to open and close the gripper, to drive toward the object, and son on.) The inference engine Ands the near and facing bit-vectors to form the bit-vector for the graspable predicate. It Ands this with the bit-vector of objects it wants to have in its hand and with the complement of the bit-vector of objects it already has in its hand (as determined by the vision system). The result is a bit-vector specifying the object to try and grasp, if any. By feeding this bit-vector to the **grasp!** behavior, the inference engine can control grasping. When the bit-vector is nonzero, the behavior fires and tries to grab the specified object. When the behavior succeeds in grasping the object, the in-hand predicate will change, automatically deactivating the behavior. Because these bit-vector inference operations are cheap to perform, Kluge can recompute them on every cycle of its decision loop, continually updating its knowledge base. So, if the object slips out of the gripper after being grasped, the grasp behavior immediately refires.

For its part, the **grasp!** behavior doesn't need to know which tracker is tracking the object,

or even which sensory modality is tracking it. It simply passes the tag on to the tracking systems. All trackers match the tag to their own tags in parallel and the matching tracker drives the behavior's input bus with the object's coordinates. Again, this matching and transmission is performed on every cycle, so the behavior can close a feedback loop around the tracked position of the object. We call this type of tagged architecture *role passing* because nearly all the data coordination between the various subsystems is performed by passing sets of linguistic roles.

Unfortunately, the inference rules in Figure 4 are useless unless the vision system is tracking the object. When the robot is told "get the green ball," the parser only binds the role **object** to the description pool; it doesn't do anything to the vision system. To even know whether it's facing the object, Kluge has to direct its visual attention toward the relevant objects. It does this using the inference rules

$$\mathbf{see}(X) \Rightarrow \mathbf{know}(\mathbf{near}(X))$$
$$\mathbf{see}(X) \Rightarrow \mathbf{know}(\mathbf{facing}(X))$$
$$\mathbf{see}(X) \Rightarrow \mathbf{know}(\mathbf{in\text{-}hand}(X))$$
$$\mathbf{goal}(\mathbf{see}(X)), \mathbf{not}\ \mathbf{see}(X), \mathbf{know}(\mathbf{color}(X)) \Rightarrow \mathbf{track!}(X)$$

The first three rules state that it knows whether objects are nearby and so on when it can see them—that is, when the vision system is tracking them. The last rule says that if it wants to see something but doesn't, and it knows what color the object is, then it should fire the **track!** behavior on it. The **track!** behavior allocates a tracker, binds it to the specified role, and sets it searching for the right color. These rules are also compiled into the control network shown in Figure 4. Figure 5 shows a screen capture from the running robot as it tries to deliver a green ball to a blue target while a mischievous human repeatedly steals the ball from it.

The Cerebus Project

The unifying theme of all these systems is to import the useful features of traditional symbolic AI systems into behavior-based systems without also importing the model-tracking and model-coherence problems. I've argued that you can implement forward- and backward-chaining inference for useful subsets of predicate logic using the kinds of parallel, distributed computations commonly employed in behavior-based systems.

My group's current effort in this direction is the Cerebus system (see Figure 6), an attempt to build, within a nominally behavior-based architecture, a "self-demoing robot." (The name "Cerebus" is not a reference to the cerebrum of the brain, but rather to a barbarian aardvark in a satirical comic book series written by artists Dave Sim and Gerhard. Like our robot, Cerebus is short, brutish, and stupid and only speaks of itself in the third person.)

The goal is for Cerebus to be capable not only of interacting with the world, but of using reflective knowledge about its own capabilities to interactively describe and demonstrate those capabilities. Cerebus combines a set of perceptual–motor systems with reflective knowledge about those systems, allowing it to perform limited reasoning about its own capabilities. Cerebus distributes its representations of itself and its world through a number of semiautonomous representational systems linked by role-passing:

- Tracker and description pools, as in Kluge
- A pool of place nodes in a topological map. Places can be bound to roles and the map reports the role of the robot's current location (if any) and the direction of the next point on the path from the current location to the goal location.
- A *lexicon pool*, entries of which are automatically bound to roles by the parser as the user types an utterance.

Cerebus also includes a set of reflective pools that give the robot access to its own internal state.

The *behavior pool* holds bindings between tags and specific robot behaviors. Each behavior continually compares its tag to the set of tags on a global **call** signal. Whenever a behavior detects a match, it activates itself. (Of course, behaviors can also be activated in all the normal ways, including bottom-up self-activation.) Active behaviors also drive a global **running?** signal with the bit-vector of their tags. The signal therefore holds the tags of all running behaviors, allowing any part of the system monitoring the signal to determine whether the behavior bound to a given tag is running.

The *proposition pool* holds bindings between tags and specific binary-valued signals in the system. The pool generates a **true?** signal comprised of the set of all tags bound to propositions that are presently true. This allows one component of the system to "pass" a signal to another component by binding it to a tag that has been agreed upon in advance.

Figure 5. Kluge follows the command "continually bring green to blue." The parser binds the object role to the color green and the destination role to blue, then asserts the goals in-hand(object) and near (destination). The robot grasps the green ball then drives toward the blue trash can.

Figure 6. The hardware used by Cerebus. A commodity laptop with a USB frame grabber and an NTSC board camera mounted on a Real world Interface Magellan robot base.

The receiving component can then monitor the signal by inspecting the appropriate bit of the **true?** signal.

The *predicate pool* holds bindings between tags and unary predicates. The predicate pool generates vector of signals, indexed by role, whose elements hold the extensions of all bound predicates—role 0 in element 0, role 1 in element 1, and so on. Again, this provides an indirection facility for passing signals between components.

Cerebus also has a marker-passing semantic net. Nodes within the net can be bound

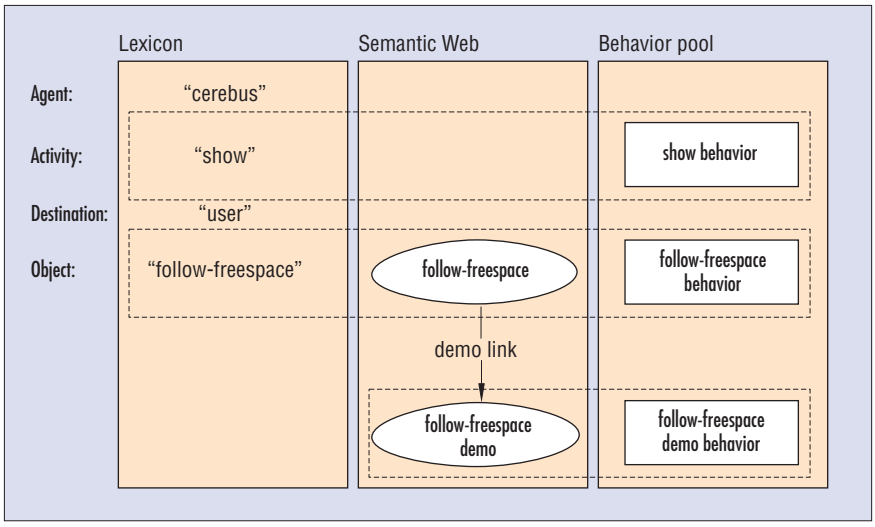

Figure 7. Tag bindings in the task "show me freespace following." Tags are shown on the left, and their respective bindings within pools are on the right. Objects that share a single tag register are linked in dashed rectangles. The parser established the initial bindings of agent, activity, destination, and object in the lexicon. Activity and object are then implicitly bound in the semantic net and behavior pools by virtue of register sharing. The show behavior, then binds follow-freespace-demo to the scratch role X by performing a marker-passing operation in the semantic net.

to role tags and then propagated as markers along links in the net to perform retrieval and inference from long-term memory.

It is important to understand that a given object or concept might be represented in several of these pools simultaneously, with each pool representing different aspects of the object. This is supported in part by allowing elements of different pools to share a single tag register. For example, the lexicon pool entry for the word "show," the behavior **show**, and the semantic net node representing information about the behavior all share a common tag register. Therefore, when the parser binds "show" to a role, the behavior that can implement the verb is automatically bound to the same role at the same time. Conversely, if some other process binds the behavior to a role, the lexical entry is automatically bound, thereby insuring that the robot will be able to name the behavior verbally, should it be necessary.

Allowing system components such as behaviors and signals to be tagged gives the system the ability to *reify* those components—to make them data objects that can be manipulated, inspected, and passed as parameters in their own right. Allowing them to be associated with nodes in the semantic net gives the system the ability to store reflective knowledge about its own structure and capabilities.

Reification of behaviors allows Cerebus to implement higher-order behaviors—behaviors that are parameterized by other behaviors. One example of a higher-order behavior is **handle-imperative**. When the user types the command "show me freespace following," Cerebus' parser reads key presses in real time as it continues with its other activities and breaks it into words. Cerebus finds the verb of the sentence, "show," binds it to the **activity** role, and identifies objects "me" and "follow-freespace," the latter is treated as a single word. Based on the verb's lexical entry, it knows that the first object should be bound to **destination**, the second bound to **object**, and the subject bound to **agent**. Since the subject is absent the parser defaults it to "Cerebus." Cerebus has behaviors by the names **show** and **follow-freespace**, each of which shares its tag register with its respective entry in the lexicon. Therefore, on completion of the parse, the **show** behavior is bound to **activity** and the **follow-freespace** behavior is bound to **object**.

The **handle-imperative** behavior automatically activates itself whenever **agent** is bound to the word "Cerebus" and **activity** is bound to a behavior. It responds by driving the **call** bus with the bit-vector representing **activity**, thereby activating the **show** behavior. It then waits until the **running?** signal no longer includes the **activity** role, the signal that **show**—or whatever other behavior was specified by the user's command—has terminated, at which point **handle-imperative** deactivates itself.

For its part, **show**, another higher-order behavior, works by driving the **call** bus with the **object** tag, thereby activating whatever behavior was bound to that tag by the parser, in this case, **follow-freespace**. It then waits for 60 seconds and stops the behavior by driving another global bus, the **stop** bus, with the **object** tag, which stops the freespace follower.

However, a much more interesting case is when **show** uses information from the semantic net to determine how to demonstrate the behavior (see Figure 7). In reality, the freespace follower also shares a binding register with a node in the semantic net, which can hold reflective information about the behavior. When that node is linked to another node by a **demo** link, the other node is assumed to contain information about how to demonstrate the behavior. In particular, **show** assumes the other node will be linked to a special demo behavior used to **show** off the freespace follower. When the **show** behavior detects that the **object** role is bound to a semantic net node with a demo link, it binds the linked node to another role. Because the new net node shares a tag register with the actual demoing behavior it represents, the behavior is tagged at the same time, allowing **show** to activate it using the **call** bus.

Cerebus is still a work in progress. Its parser can resolve references to objects in its various pools, and its primitive natural language generator can produce descriptions of the object bound to a given role, regardless of the representational systems tracking it. However, most of the work on Cerebus has gone into building infrastructure—the parser–generator, the reification system, the semantic net, and so on—rather than filling that infrastructure up with content. Thus, there is a limit to what Cerebus can actually do with a user. A typical interaction with Cerebus might be

Cerebus: Hello.
User: What can you do?
Cerebus: cerebus can do follow-freespace, follow-color, follow-human, and answer-question.
User: Show me freespace following.
Cerebus: ok. cerebus show user freespace-following.
… (Cerebus shows off freespace following) …
User: (Interrupting) Stop.
Cerebus: ok. cerebus stop.
User: Give a talk.
Cerebus: ok. cerebus give user talk.
… (Cerebus does a simple tree-walk of its

semantic net to generate a largely prescripted talk)

User: (Interrupting) Stop.

Cerebus: ok. cerebus stop.

User: Die.

Cerebus: ok. cerebus die.

… (the Cerebus program exits).

Note that Cerebus is pointedly dysfluent. This is partly because it generates sentences by walking the bindings of its internal representations (semantic net, lexicon, reified behaviors, and so on), which are fairly simple to begin with. It cannot generate complex noun phrases or even noun phrases with determiners ("a," "the," "some"). However, we have also intentionally exaggerated that dysfluence—for example, by having it refer to itself in the third person. If the robot only generates simple sentences, hopefully the user will only give it simple sentences.

It is common to see behavior-based systems and symbolic reasoning systems presented as distinct, or even incompatible, approaches to intelligent control. This view has left behavior-based systems representationally impoverished and therefore, highly limited. However, not only is symbolic reasoning compatible with behavior-based systems, it's *implementable* by behavior-based systems. The relevant distinction is not between symbolic and nonsymbolic computation but rather between a transaction-based model of computation and a circuit-based model of computation. Although nearly always implemented in conventional programming languages, behavior-based systems have traditionally used styles of computation that are analogous to parallel circuits. This style of computation is easier to interface to sensors and effectors and, because it recomputes everything on every clock tick, is highly reactive. Symbolic reasoning systems have traditionally been implemented in Lisp, or more recently, Java. However, the same I/O behavior can often be implemented in circuit-style computation, allowing greater reactivity and easier interface to sensory systems.

Of course, just because something is possible doesn't necessarily mean it's a good idea. Hand-engineered rules in straight logic have many limitations. Learning approaches, as well as probabilistic or fuzzy reasoning systems, have also shown great promise. However, these systems, such as behavior-based systems, have traditionally been representationally impoverished. Hopefully the techniques described here can be used to extend these systems to more powerful representations. And, of course, many things are not implementable in this framework and probably never will be. Full-blown, domain-independent planning, for example, involves unbounded search and cannot be done with a fixed parallel network. When such techniques are necessary, a tagged architecture would need either to make out-calls to a more traditional Lisp program or effectively emulate a Lisp interpreter. Both approaches are worth investigating.

These caveats aside, I believe that there is a rich and largely untouched area of research in trying to extend parallel distributed control architectures to support more expressive representations. The most common approach to improving the reactivity of symbolic reasoning systems is to build a hybrid architecture in which a symbolic planner generates plans to be executed by a behavior-based system. Even if such hybrid architectures do ultimately prove necessary, their performance can be improved dramatically by pushing as much work as possible down from the deliberative components into the behavior-based components. There is a great deal of interesting work still to be done. What other techniques besides tagging can be used to extend distributed representations? How can exception handling and meta-level reasoning be incorporated into these architectures? Can online statistical learning techniques be extended to more expressive representations? If so, can it leverage those representations to improve performance? The answers are by no means obvious, but any advances that can be made will have a high payoff.

One of the reasons that behavior-based systems have lagged behind traditional symbolic systems is because we haven't yet found the right set of tools for building them. While symbolic reasoning systems have very advanced languages, compilers, and development environments for them, most behavior-based systems are still written in procedural languages like C++ that don't have built-in notions of circuits or finite-state machines, much less of reified behaviors. Programmers therefore must in some sense program in two languages at once. They first conceive of their program within some higher-level behavior-based architecture, then hand-compile it to C or Java code. As modifications are made, they must solve problems at both these levels and incrementally recompile manually. The process tends to be painful and error-prone. Debugging has to be done at the C level rather than at the architectural level—assuming some kind of thread-safe parallel debugger is available at all. In my own work, I have used a functional language for circuit layout.[8] While it's been a big help—we couldn't have built Cerebus without it—it's still far too limited. What's needed is some equivalent of a Lisp or Prolog integrated development environment for behavior-based systems. ■

References

1. S. Russell, and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.
2. I. Horswill, "Visual Routines and Visual Search: A Real-Time Implementation and an Automata-Theoretic Analysis," *Proc. 1995 Int'l Joint Conf. Artificial Intelligence*, Morgan Kaufmann, San Francisco, 1995.
3. S. Ullman, "Visual Routines," *Visual Cognition*, MIT Press, Cambridge, Mass., 1984.
4. R. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE J. Robotics and Automation*, vol. RA-2, no. 1, 1986, pp. 14–23.
5. L. Kaelbling and S. Rosenschein, "Action and Planning in Embedded Agents," *Designing Autonomous Agents*, P. Maes, ed., MIT Press, Cambridge, Mass., 1991, pp. 35-48.
6. I. Horswill, "Grounding Mundane Inference in Perception," *Autonomous Robots*, vol. 5, Kluwer Academic Publishers, Netherlands, 1998, pp. 63–77.
7. I. Horswill, "Functional Programming of Behavior-Based Systems," *Autonomous Robots,* vol. 9, Kluwer Academic Publishers, Netherlands, 2000, pp. 83–93.

The Author

Ian Horswill is an assistant professor of computer science at Northwestern University where he holds the Lisa Wissner-Slivka and Benjamin Slivka Junior Professorship in Computer Science. His research focuses on integrating high-level reasoning systems with low-level sensory-motor systems on autonomous robots. He received his BSc from the University of Minnesota and his PhD in computer science from the Massachusetts Institute of Technology. Contact him at the Computer Science Dept., Northwestern Univ., 1890 Maple Ave., Evanston, IL 60201; ian@cs.northwestern.edu.