Center for Cognitive Architecture
University of Michigan
2260 Hayward St
Ann Arbor, Michigan 48109-2121

TECHNICAL REPORT
CCA-TR-2009-01

An Overview of Spatial Processing in Soar/SVS

Investigator

Samuel Wintermute

23 June 2009

Table of Contents

1 Introduction

For the last several years, work has been ongoing to augment the Soar cognitive architecture with specialized processing for spatial and visual information (Lathrop, 2006, 2008; Lathrop & Laird, 2007, 2009; Wintermute, 2009, forthcoming; Wintermute & Laird, 2007, 2008, 2009). This work originally took the form of two different modules, Soar Visual Imagery (SVI, developed by Scott Lathrop), and Spatial Reasoning for Soar (SRS, developed by Sam Wintermute). However, recently the functionality of both systems has been combined into a new system, the Spatial and Visual System (SVS, Wintermute & Lathrop, 2008).

As the architecture has been evolving throughout the course of this work, previous documents so put together do not provide a totally satisfactory account of the current system. The purpose of this document is to provide a comprehensive overview of how the current version of the architecture works.

SVS is based in part on theories of human visual imagery (Kosslyn et al., 2006), but with a strong emphasis on increasing agent functionality. The architecture includes short-term and long-term memories for both spatial and visual information, along with mechanisms to access and modify those memories. The discussion here will focus mainly on the spatial aspects of this. The visual system of SVI, upon which that in SVS is based, was covered in detail by Lathrop (2008), although some aspects have changed since then.

Parts of this report are adapted from previous (and pending) publications. The beginning of section 2 was adapted from (Wintermute, forthcoming). Section 2.6.1 was adapted from (Wintermute & Laird, 2007), section 2.6.3 from (Wintermute & Laird, 2008), section 3.1 from (Wintermute & Laird, 2009) and (Wintermute, forthcoming), and section 3.2 from (Wintermute, 2009).

2 System Design

To understand the design of SVS, we must first look at imagery processing in the context of an intelligent agent (Figure 1).

Many types of AI systems fit the basic pattern that perceptions are mapped to a high-level problem state, and decisions are made in terms of that problem state. Call the direct output of the agent's sensors $\boldsymbol{P}_l$, for low-level perception. This signal is transformed by the perception and imagery systems to create a higher-level perception signal, called $\boldsymbol{P}_h$. We will call the highest-level part of the agent that receives this signal and decides what to do next the "decision system" (although it could be argued that decisions are made at many levels in the system). The internal problem state in the decision system can directly be $\boldsymbol{P}_h$, or some more complicated function of $\boldsymbol{P}_h$ taking into account past observations and background knowledge. The decision system also typically uses a high-level representation of actions: it is rare that actions are considered in terms like "set motor voltage to .236", even though that may be the final output of an embodied agent. So, even in a simple system, there are typically distinct high- and low-level action signals, $\boldsymbol{A}_h$ and $\boldsymbol{A}_l$, and motor processing that creates $\boldsymbol{A}_l$ from $\boldsymbol{A}_h$.

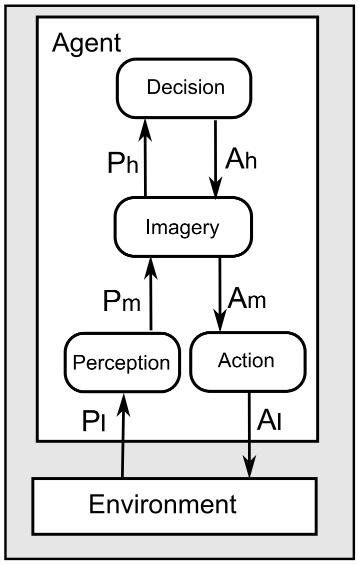


Figure 1: Imagery with the context of an agent.

A box for the imagery system lies between high-level decision processing and low-level perception and action. Like the boxes for perception, action, and decision, it might incorporate arbitrary processing and memory. The imagery system provides an additional level of perceptual and action processing, above the lower-level modules. The output of low-level perception is provided to the imagery system, so it is called $\boldsymbol{P}_m$, for mid-level perception. Processing in the imagery system transforms $\boldsymbol{P}_m$ into $\boldsymbol{P}_h$, which is the perception signal provided to the decision system. Note that this happens independently of whether the state inside the imagery system is real or imagined: the form of $\boldsymbol{P}_h$ is the same, just possibly annotated as real or imagined. That is, the imagery system performs the

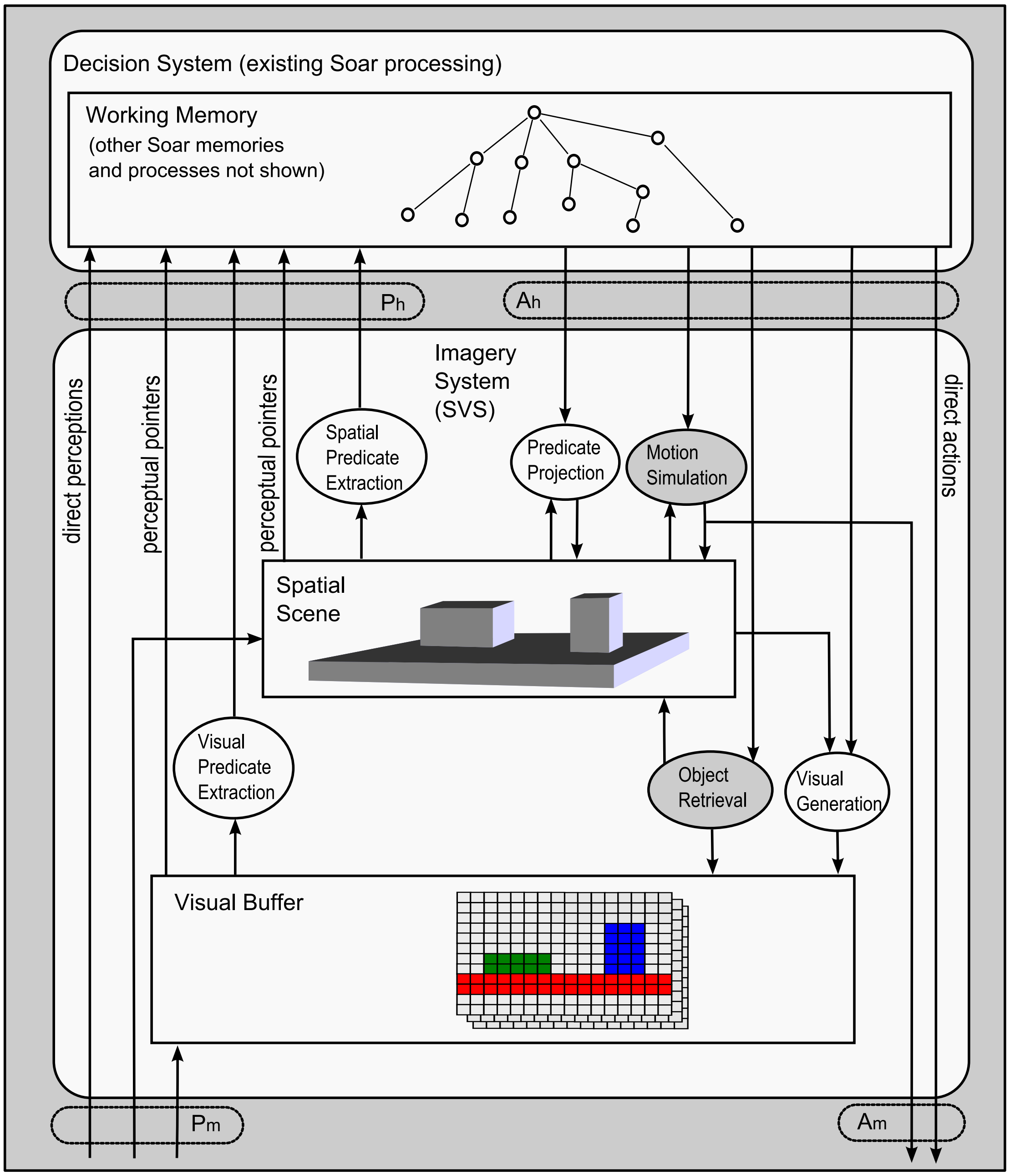


Figure 2: SVS System design. Boxes are short-term memories, circles are processes. Grey circles involve access to information in long-term memory (knowledge). There are implicit control lines (not shown as arrows) between working memory and all of the processes shown.

same high-level perception over both real and imagined data.

Agents can thus be built where the decision system can issue actions that either cause actual action in the world, or simulate the results of that action in the imagery system. Moreover, the system can now use actions that cause changes to the imagery system, but do not have a corresponding external action. These imagery actions can be used for many different things, for example, memories could be retrieved, or geometric reasoning could be performed.

SVS is an example of this sort of imagery system. The overall design of SVS is shown in Figure 2. The existing Soar architecture (Laird, 2008) constitutes the decision system in this case. Soar contains a symbolic working memory, through which different processes in memories in Soar communicate. This is where SVS connects to the system, high-level perception (***P***$_h$) adds elements to a special area of working memory, and high-level actions (***A***$_h$) are similarly formulated in a special area of working memory. The ***P***$_h$ and ***A***$_h$ signals actually have many meaningful components (there are many "forms" of perception and action), as shown in the diagram, and will be explained later. Other cognitive architectures, such as ACT-R (Anderson et al., 2004) similarly use symbolic structures to connect to the outside world, the concepts behind SVS (if not the software itself) could be adapted to any system with such an interface.

As an imagery system, SVS sits between symbolic processing in Soar and the world. A complete embodied agent also requires lower-level perception and action system to handle the actual output of sensors and input to effectors. These systems are the source of the ***P***$_m$ signals and receiver of the ***A***$_m$ signals, respectively. SVS processes visual and spatial data, but other forms of perceptions and actions may be necessary, especially in virtual environments. For example, an agent playing a computer game might need a perception of its score or health, and an action to reset the game. These are simply passed through SVS ("direct" perception and action in the Figure)[1].

The memories inside SVS are influenced by Kosslyn's theory of visual imagery (Kosslyn et al., 2006). There are two short-term memories in the system for spatial and visual information. The visual buffer represents information in 2D arrays of pixels, analogous to the part of the human visual system active during classical imagery tasks. This represents strictly visual information, including precise shapes and colors. In contrast, the spatial scene contains 3D information that is (theoretically) derived from multiple senses, quantitatively represented as continuous coordinates describing polyhedrons in 3D space[2]. In addition to the two short-term memories, there is a long-term memory in SVS for visual, spatial, and motion data, called perceptual LTM. To simplify the diagram, this memory is not explicitly shown, but is accessed by the object retrieval and motion simulation processes. While these memories all contain chiefly non-symbolic information, they also are partly symbolic—unique objects are given identifying symbols, through which the symbolic aspects of the system can refer to them. These identifying symbols, here called *perceptual pointers*, are similar to the visual indices described by Pylyshyn (2001) for short-term visual memory, since the system "... picks out a small number of individuals, keeps track of them, and provides a means by which the cognitive system can further examine them in order to encode their properties ... or to carry out a motor command in relation to them".

This system operates both during perception and imagery, and much functionality is shared by those processes. For real-world agents, perception is a major challenge. Theoretically, the perceptions provided to SVS (***P***$_m$) should be raw pixels from a camera, or something analogous to the lowest cohesive representation in the human visual system. This is the component of ***P***$_m$ which feeds directly to the visual buffer. As will be discussed, imagery in terms of this basic representation is useful, so it should not be abstracted out by the low-level perception system (which then has only a very minor role to play,

[1] These signals could equivalently be drawn as connecting directly to the environment. They are drawn this way both to follow the convention established in the previous figure, and to reflect the actual underlying software implementation.

[2] This is the equivalent of the Object Map in (Kosslyn et al., 2006), and in (Lathrop, 2008), which inherits the terminology.

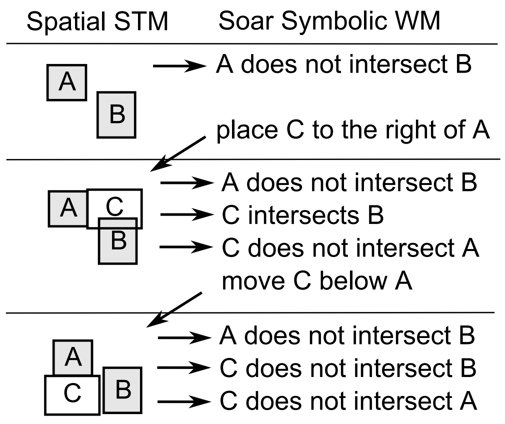


Figure 3: Simple problem solving using imagery. The problem is to determine a position for box C, such that it does not intersect box A or B, but is adjacent to A.

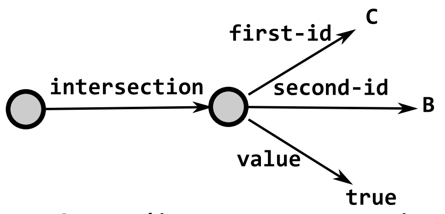
if any, since $\boldsymbol{P}_\mathrm{m}$ will probably be very similar to $\boldsymbol{P}_\mathrm{l}$). Theoretically, memories and processes inside SVS, with influence from symbolic processing in Soar, should segment and recognize objects, and estimate 3D spatial structure based on 2D visual information, but we will not discuss this problem here.

However, for many virtual environments (and some limited problems in real environments), it is possible for a separate perception system to provide information at a high-enough level that SVS can be used without a complete visual system. Many simulated environments directly represent the world as labeled 3D polyhedrons; in that case, those objects and labels can be directly fed into the spatial scene via $\boldsymbol{P}_\mathrm{m}$. In the current implementation, the $\boldsymbol{P}_\mathrm{m}$ component to the visual buffer isn't used, only the component to the spatial scene. This does not mean that the visual buffer isn't used at all, though, as visual information can be derived from the spatial scene, as will be discussed later.

From the point of view of the decision system, the only aspects of the underlying visual and spatial state available are what is encoded in $\boldsymbol{P}_\mathrm{h}$. Here, that is the object identifiers, along with information available through spatial and visual predicate extraction[3] processes. These processes, outlined fully in the next section, extract qualitative symbolic information from the underlying quantitative state. For example, this process can detect whether or not two objects intersect. Note that these processes do not involve access to task-specific knowledge: there is a fixed, architectural library of predicates that the system can extract. The exact visual and spatial details of the objects in the world (their coordinates in 3D space) are *not* provided in $\boldsymbol{P}_\mathrm{h}$.

Since the information available to the symbolic system is limited to object identities and simple qualitative properties, for complex reasoning tasks, imagery must be used. Figure 3 shows an example of this sort of reasoning process, in a simplified system. An *image* is an object in one of SVS's short term memories that is the result of top-down processing, it is something not present in $\boldsymbol{P}_\mathrm{m}$. In the example in the figure, symbolic processing creates a candidate image for the new object, which it refines based on feedback from imagery. Essentially, imagery allows the system to address the complexity of the problem through simple processing over time (repeatedly placing the image and detecting its properties) instead

[3] This terminology is inherited from work in diagrammatic reasoning (Chandrasekaran, 1997). The term "predicate" is perhaps overly formal, since it might imply that predicate logic is being used for inference in the system, which isn't the case, but it has the correct implication that we are dealing with symbolic properties of objects.



```
intersection.first-id C
            .second-id B
            .value true
```

Figure 4: Soar working memory representation. Working memory is a directed graph, which can be represented either graphically (left), or textually (right).

of through complicated single-step processing; in this case, directly interpreting a statement like "place `C` adjacent to `A`, not intersecting `A` or `B`".

To perform imagery, the system needs mechanisms through which visual and spatial images can be created and manipulated. In SVS, there are four such mechanisms, which will be outlined in detail in the following sections. These mechanisms include memory retrieval, which instantiates objects from long-term memory into either the visual or spatial STM, motion simulation, which moves images in the spatial scene in the same way a motor action or other motion in the world would, and predicate projection, which creates spatial objects based on qualitative descriptions created by the symbolic system (like "a line between `A` and `B`"). In addition, there is some implemented ability for new visual objects to be recognized, which happens as a result of top-down control from the symbolic system.

2.1 Perceptual Pointers

The most basic form of information passed between SVS and symbolic processing in Soar is the perceptual pointer. These are unique tokens, each of which refers to a specific underlying visual or spatial structure. These appear as `id`[4] attributes in Soar's working memory. In addition, if a structure is recognized as an instantiation of a structure in perceptual LTM, it is augmented with a `class-id`. Figure 5 shows an example of how a simple scene is presented to Soar's symbolic working memory, which will be discussed in more detail in the next section.

The types of structures that use this identifier system are spatial objects, spatial transformations, visual textures, and motion models, all of which will be discussed below. The perceptual pointer provides a simple means by which a symbolic working memory structure can refer to an underlying perceptual structure: the pointer id is generated by SVS, and every time symbolic processing in Soar uses that id in a context SVS can understand (e.g., an imagery specification to "imagine `car23` to the right of `house12`"), SVS uses that id to look up the underlying perceptual structure (e.g., the polyhedron describing the car) in its internal memories.

[4] Throughout this document, words in `fixed-width` font refer to tokens in Soar's working memory. Figure 4 shows how working memory structures can be represented either textually or graphically.

2.2 Spatial Scene Encoding

The spatial scene is SVS's short-term memory for spatial information. It normally contains the structure of the world around the agent (including parts it can't immediately see), or the structure of an imagined situation, or, more commonly, a mixture of both. Internally, the scene is a set of 3D polyhedrons grounded in continuous coordinates. This information is presented to Soar's working memory as a hierarchically-organized tree of objects, with the tree structure indicating part-of relationships. Each object node includes a perceptual pointer by which Soar can refer to it. Between each object node, a transformation node is present. Each transformation node contains a perceptual pointer to the relationship between the two objects. Figure 5 shows how a simple scene is presented to Soar.

Only the leafs of this tree correspond to primitive polyhedrons, but nodes at every level are considered objects. For example, in Figure 6, everything in the scene is part of one **scene** object (the root of the tree), and the **tree** object has children for its parts, the **trunk** and **canopy**. Much of the processing in SVS (e.g., detecting an intersection between two objects) relies on the assumption that objects are convex, so when objects are referred to, what is actually used is the convex hull of all of the parts below that object. If non-convex objects are to be used, the environment must encode them as a set of convex parts, each of which the system can reason with independently. Automating this convex decomposition is an area for future work.

Since the scene grounds objects at locations in 3D space, an appropriate concern might be how the coordinate frame chosen affects the rest of the system. Is the scene allocentric (encoded relative to a "world" frame of reference) or egocentric (encoded relative to the agent's point of view)? The answer is both and neither. It turns out that the actual coordinate frame of the scene is arbitrary, it can be changed without affecting the rest of the system. Intuitively, this is because the properties that the higher-level parts of the system care about are qualitative properties of objects relative to one another, properties which are preserved across translation, scaling, and rotation. However, it is possible for the agent to *interpret* the scene in allocentric or egocentric ways.

```
svs.spatial-scene.object.id scene
                       .transform-child tree-transform
                       .transform-child house-transform
                .transform.id tree-transform
                          .object-child tree
                .transform.id house-transform
                          .object-child house
                .object.id tree
                       .class-id tree-type-37
                .object.id house
                       .class-id house-type-24
```

Figure 5: A partial example of the symbolic encoding of the spatial scene in Soar's symbolic working memory. See also Figure 6, which shows a scene graph and spatial information for a similar scene.

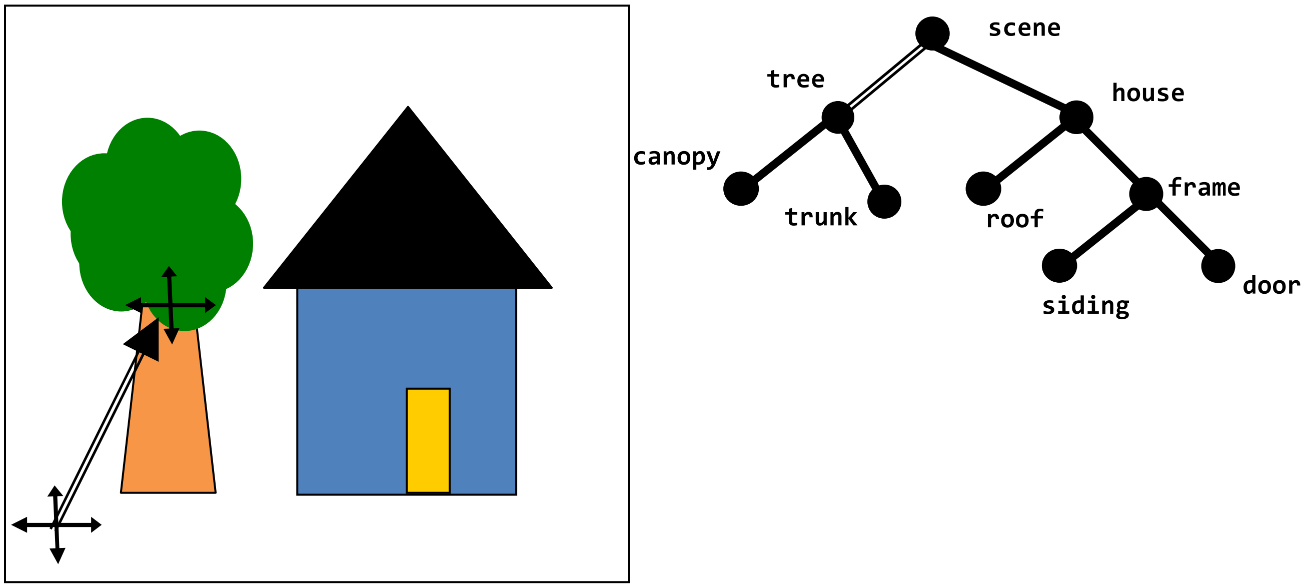


Figure 6: An example 2D scene and accompanying scene graph. The transformation from the scene frame of reference to the tree frame of reference is shown. The previous figure shows how a similar scene is encoded in Soar's symbolic working memory.

To understand what this means, it is important to examine the underlying implementation. Internally, the scene is encoded as a scene graph, a structure commonly used in computer game graphics engines (Eberly, 2004). For our purposes, we can consider this graph as a tree. This is (conceptually and as implemented) the same tree as the object hierarchy provided to Soar. Each node in this tree contains its own internal coordinate system, but it also includes a local transformation (a data structure encoding translation, rotation, and scaling), representing the relationship of the current node's coordinate system to that above it. The coordinate frame of the root of the tree is considered the global frame, this is the frame in which all objects are grounded. Local transformations can be chained together, so the grounded location of a polyhedron in the tree is calculated by applying all of the local transformations above it to the coordinates of the polyhedron. In Figure 6, a local transformation is shown, going from the frame of reference of the **scene** to that of the **tree** object.

In its native encoding (that is, in the encoding scheme commonly used in computer graphics), the scene graph represents transformations between objects in line with parent-child (whole-part) relationships. Conceptually, however, the scene contains transformations between every pair of objects, as there is enough information in the scene graph to calculate any of these relationships. For example, for the situation in the figure, the agent might wish to consider the relationship between the **canopy** and the **door**. The scene graph does not explicitly represent this relationship; it only encodes relationships between objects and their parts. The relationship can be efficiently computed, though, by calculating each object's coordinate frame relative to the **scene** (a product of local transformations), and determining the transformation from the **canopy**'s scene-relative coordinate frame to that of the **door**. Similarly, it is possible to compute the transformation from the agent's body (an object in the scene) and other objects which aren't parts of the agent—these are the egocentric transformations of those objects.

In the current implementation, symbolic processing in Soar has no way to directly reason with transformations in the scene other than those explicitly encoded by the underlying scene graph. That is, the transformation pointers passed from the scene to working memory refer to transformations between wholes and their parts. This is an area for future work, but is chiefly an engineering problem. Once this aspect is complete, the system should be able to better model the difference between egocentric and allocentric encodings in tasks like spatial navigation (as examined by models such as that in (Chown et al., 1995)). This will not be a reflected as a difference in the encoding of the underlying spatial scene itself, but as a difference in the way the scene is accessed by the symbolic system.

It should be emphasized that the perceptual pointers that make up the scene graph in Soar's working memory refer to the actual objects and transformations in the spatial scene. In Figure 6, if the agent refers to the **`id`** of the scene-tree transformation shown, it is referring to a specific quantitative transformation in 3D space, *not* a more generic qualitative relationship like "towards the upper right" of "northeast of". If the agent retrieves a transformation from perceptual long-term memory (by referring to its **`class-id`**), that transformation is similarly quantitative. In general, everything in perceptual LTM and in the STMs of SVS is quantitative. However, symbolic processing in Soar can only *refer* to quantitative perceptual information, it cannot directly access the actual quantities involved. Rather, it accesses that information only indirectly, by querying the scene for qualitative information through the predicate extraction process (to be covered shortly).

2.3 Visual Buffer Encoding

The internal encoding of the visual buffer is relatively straightforward. It is a set of pixel arrays, at a fixed resolution. Each array in the buffer is called a *depiction*[5], and has a perceptual pointer presented to working memory. Pixels in a depiction can be set to a specific color, or set to a special value indicating emptiness.

Since the visual buffer is intended to correspond to the human visual system, it is important to address why this is a *set* of depictions, rather than a single pixel array. Having a set of depictions is useful for representing segmentation (all pixels belonging to the same object are in the same depiction), but this could also be done by annotating pixels within a single array. A set of depictions is more powerful though, since it allows multiple objects to exist at the same location.

It could be argued that, in humans, visual imagery and perception compete for the same resource (activation in the visual system), and that each is likely to override the other in certain circumstances. Using a set of depictions does not include this constraint— an unbounded number of real and imagined objects can be "seen" equally well, even if they are in the same visual position. This is chiefly an engineering benefit, but if the system is ever adapted for cognitive modeling, it will probably need to be addressed in more detail.[6]

[5] These were called "layers" by Lathrop (2008).

[6]A similar argument about the implausibility of overlaying real and imagined spatial perception without degradation could be made, or even real and imagined aspects of symbolic reasoning, but the case is particularly strong for low-level vision (e.g., Kosslyn et al., 2006).

2.4 Perceptual LTM Encoding

The internal representation of perceptual LTM is more heterogeneous than the other parts of SVS. It stores scene graphs which encode spatial information, but also textures (visual information) and motion patterns. Perceptual pointers (**`class-id`**s, in this case) are passed to Soar for all of these things. Conceptually, associations between different structures in perceptual LTM should also be encoded, for example, linking a car object with its corresponding motion pattern. This aspect is only partially implemented, however. In addition, this memory should eventually be integrated with Soar's semantic memory (Wang & Laird, 2006), allowing higher-level concepts in long-term memory to be associated with visual and spatial information.

2.5 Predicate Extraction

The predicate extraction processes serve to provide symbolic processing in Soar with qualitative properties of the contents of the spatial scene and visual buffer short-term memories in SVS. These processes are fixed parts of the architecture; there are no plans to enable new forms of predicate extraction to be learned by the agent itself. In contrast to perceptual pointers, qualitative predicates are presented to working memory only when requested by processing in Soar. There is a great deal of qualitative information implicit in the memories of SVS, each piece of which can take substantial calculation to derive, so some attention mechanism is needed to make the system computationally tractable. The process of requesting predicate extraction from working memory is called *querying* SVS.

For the spatial system, there are three important kinds of relationships between objects that might be queried for: topology, direction, and distance. Topological relationships describe how the surfaces of objects relate to one another. Work in qualitative spatial reasoning has explored these relationships (Cohn et al., 1997). While schemes exist that represent several possible cases of topological interaction (e.g., discrete, partially overlapping, proper part), the only topological information currently available in SVS is whether or not two objects are intersecting or not (that is, colliding). More relationships might be added in the future, but no tasks have yet been addressed that require more detailed types of topological information.

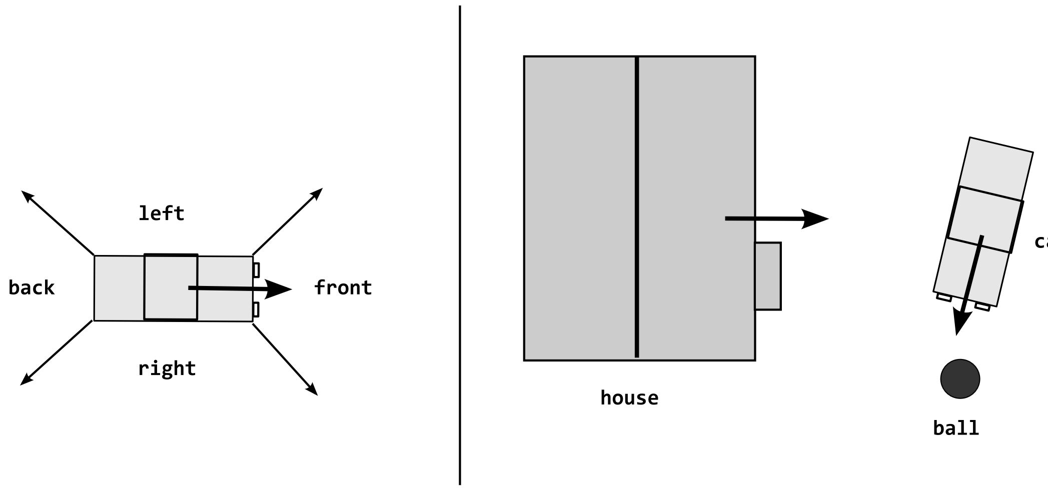


Figure 7: Orientation information in SVS (two-dimensional simplification).
Left: The intrinsic front vector of the car object, along with its shape, defines four acceptance areas, which are used to define qualitative orientation information. For example, an object in the region labeled "right" is considered to be to the right of the car.
Right: A scene showing how the intrinsic frames of reference for objects can vary (see text).

Distance is similarly simple, currently the system is able to query for the distance between any two objects in the scene, along the closest line connecting them. In the current implementation, this distance is simply extracted as a continuous number. This makes the information arguably non-qualitative, although it is certainly "less quantitative" than the contents of the spatial scene, as it reduces three-dimensional information to a scalar quantity. However, it is extremely useful in practice. The common use for distance information is simple distance comparison, which can be done easily with Soar's existing number-comparison functionality. The closest obstacle to the agent might be detected by extracting the distance from the agent to all of the obstacles, and comparing the distances to determine the closest. If we wanted to strictly evict all continuous numbers from Soar's symbolic processing, this could easily be replaced with a "closest" extractor, but that currently seems an overcomplication.

To support orientation relationships between objects, and determine information such as "object A is to the left of object B", a final class of orientation queries is implemented. Following the approach of (Hernández, 1994), for each object, a set of surrounding *acceptance areas* is defined. An acceptance area corresponds to a region of the world where all points in that region share a common orientation with the object in question. These regions roughly correspond to concepts like left, right, front, back, above, below, etc. An example set of acceptance areas for a two-dimensional object is shown in Figure 7 (left).

Orientation queries are often performed relative to the intrinsic frame of reference of the object, as discussed in Section 2.2. However, orientation queries can be performed in the frame of reference of any object in the scene[7]: that is, the frame of reference of a different object can be used to define the

[7] In the current implementation, this must be done through a multiple-step process (generating an image of the desired object with a new reference frame, and then extracting the relationship), but it should be eventually handled entirely in the predicate extraction system.

acceptance areas. Figure 7 (right) shows an example what this means: the ball is in front of the car if the car's native frame of reference is used, but to the right of the car if the house's frame of reference is used. For this second inference, the "front" vector of the house is mapped onto the car, and acceptance areas are calculated based on this vector, along with the shape of the car. In this way, allocentric orientations can be extracted by using the frame of reference of the entire scene (or of a compass object in the scene), and interpreting the results as cardinal directions (North, South, East, West, up, down).

For some spatial queries, the 3D nature of the spatial scene can result in undesirable consequences when the problem is inherently two-dimensional. For example, an agent might want to determine how long it would take to move beneath an object that is in front of and above itself. In this case, simply extracting the distance won't work, since the vertical dimension must be ignored. In addition, if the agent is attempting to determine the closest obstacle to itself, it might calculate the point-to-point distance from its centroid to the centroid of each obstacle, which can be done much more quickly than calculating the distance between polyhedrons. For these reasons, SVS supports spatial *object interpretations*. For most queries, the system can interpret each shape as a convex polyhedron (the default), a bounding box, a centroid, or two-dimensional versions of each of those types, where the projection of the object into its intrinsic xy-plane is used. Making shape interpretations available can increase the functionality of the system, and greatly speed it up when precise calculation over complex 3D objects is unnecessary.

All queries discussed so far have concerned the spatial system, but the visual system also supports predicate extraction. As implemented, though, this is very simple. There is one predicate that can be extracted, which simply reports whether or not a given depiction has any non-empty pixels. This is often used in conjunction with the visual generation and top-down visual recognition processes, which will be discussed shortly.

2.6 Imagery

The information provided to Soar through perceptual item pointers and predicate extraction about objects the agent can currently perceive is often not enough to allow general-purpose problem solving. Often, imagery processes like that in Figure 3 must be employed. While imagery has often been proposed in the past as a means for problem solving, the exact means by which images can be created in a problem-independent manner have rarely been specified. One of the contributions of this work is exploring this problem. The current implementation of SVS inherits much of its imagery functionality from a prior system, SRS (Spatial Reasoning for Soar). Two of the image creation processes in SVS, predicate projection and motion simulation, were previously explored in detail in SRS (Wintermute & Laird, 2007, 2008). Sections on those processes below are adapted from the papers cited.

Images, once created in the spatial scene or visual buffer, are thereafter treated identically to structures in those memories built by perception. In the discussion below, the term 'image' will be used to refer to the structure being created, as in "the image is placed adjacent to the object", but it should be noted that once the image is placed, it becomes an object itself.

2.6.1 Predicate Projection

Creating a new spatial image often involves translating a qualitative representation of the image (present in symbolic working memory) to a quantitative representation in the scene. This problem has previously not been as well studied as predicate extraction (Chandrasekaran, 1997). We call the qualitative representation of a new image the *description* of that image. Our goal is to create a system with broad applicability, which requires a qualitative language for describing new images that is as expressive and general as possible. Broadly, there are two kinds of possible descriptions: direct and indirect.

An image created with an *indirect description* inherits its shape from an existing object, in the scene or in LTM. This shape is placed in the scene based on a set of abstract predicates describing the position of the object.

In SVS, the supported predicates are **`on`**, **`at`**, **`adjacent`**, and **`facing`**. Placing an image **`on`** another object positions the image above and adjacent to that object, in the z-direction. This z-direction is currently that defined in the coordinate frame of the scene root, but the system could be improved to allow the coordinate frame to be specified. Placing an image **`at`** another object results in the image being centered at the center of that object. Placing an image **`adjacent`** to another object is similar to **`on`**, but does not constrain the direction of adjacency. An image can be specified **`facing`** another object, in this case the intrinsic frame of reference of the image is aligned to point towards that object.

An indirectly described object is not guaranteed to have a valid position (an image can't be both 'at' and 'on' the same object, for example). Underlying processing in SVS attempts to interpret the set of predicates, and will add the image to the scene if it is able to find a suitable position (or report an error otherwise). In many cases, indirect descriptions are underdefined, and the system arbitrarily chooses one of many images meeting the description. A previous implementation (Wintermute & Laird, 2007) included a much more comprehensive scheme for indirect imagery, but used a two-dimensional spatial representation. The underlying processing in that system solved complicated computational geometry problems in order to place the image, but this approach could not be easily extended to the three-dimensional case.

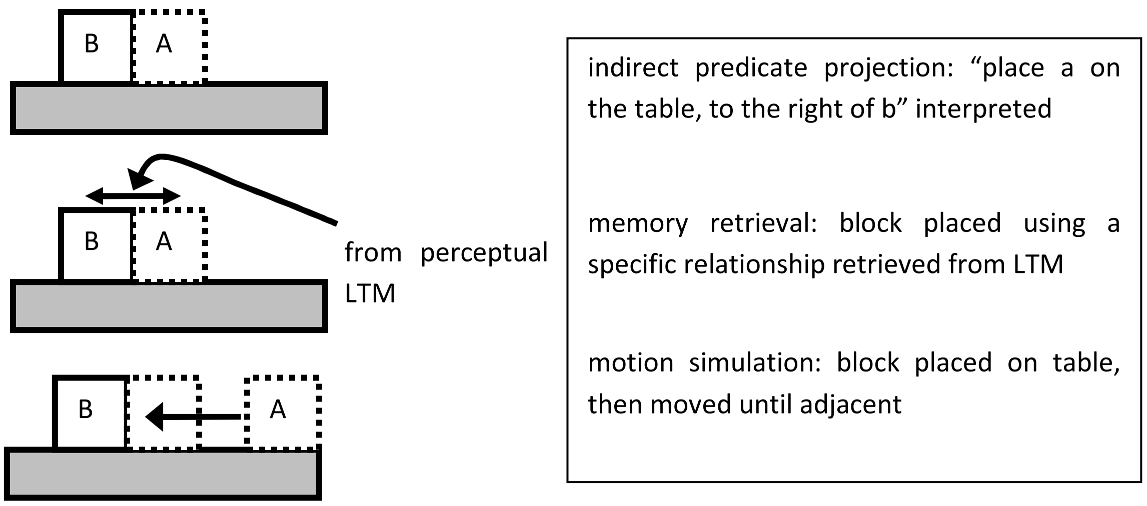

B
A
B
A
from perceptual LTM
B
A
indirect predicate projection: “place a on the table, to the right of b” interpreted
memory retrieval: block placed using a specific relationship retrieved from LTM
motion simulation: block placed on table, then moved until adjacent

Figure 8: Different approaches to adding a block to the scene.

Instead of extending SVS’s indirect imagery power, alternative approaches are being explored to allow similar capabilities in the system through different mechanisms. For example, consider the problem of placing an image of a block **`A`** on a table to the right of an existing block **`B`** (Figure 8). This could be done by allowing indirect imagery to interpret specifications like “**`A`** is **`on`** the table, **`adjacent`** to **`B`**, and to the **`right-of B`**”, by using long-term memory to recall an appropriate coordinate transformation between the blocks, or by first simply placing the block on the table, then simulating motion until it is in the appropriate position (an approach explored in Wintermute & Laird, 2008).

In addition to the indirect descriptions described above, SVS also supports *direct* descriptions. In contrast to indirect descriptions, a direct description is always unambiguous – it describes only one possible image. An example is ‘the image is the convex hull of objects **A** and **B**. For any diagram that has objects **A** and **B**, there is exactly one such shape. Other descriptions are those such as ‘the image is the intersection of objects **A** and **B**’. For any objects **A** and **B** in the scene, the image may or may not exist, depending on their positions in the scene, but if it does exist, there is only one. Thus, a direct description describes exactly one image, not a category of images.

In SVS, the supported direct imagery types include **`hull`**, **`intersection`**, **`centroid`**, **`random-point`**, **`clone`**, and **`scale`**. A **`hull`** image is the convex hull of two or more objects, and an **`intersection`** image is the region of intersection of two or more objects. **`centroid`** images simply create a point object at the center of mass of an existing object, while **`random-point`** images create a point object at a random location within an existing object[8]. A **`scale`** image is a version of an existing object, expanded in all directions by some amount, and a **`clone`** image simply copies an object.

[8] Random points are technically indirect images, since there are many possible images that could result from the description. As implemented, though, this predicate is not integrated with the rest of the indirect imagery system (it cannot be combined with other indirect predicates), and is instead implemented as a direct type.

The predicate projection processing described above is considered to be a fixed part of the architecture; there are currently no plans to allow new kinds of predicate projection commands to be learned by the agent or provided as knowledge. However, there is no strong commitment that the current library of available operations is appropriate or complete. It has served well for the tasks that have been addressed, but may change as the architecture evolves. From a theoretical point of view, the important aspect is that the system has this capability at all.

2.6.2 Memory Retrieval and Image Composition

Often, spatial images must be created based on objects in perceptual long-term memory. This involves the agent telling SVS to instantiate an object of a known type, at a known location. This is different than predicate projection—predicate projection works by the symbolic system *describing* the general required qualitative properties of the image, memory retrieval works by the symbolic system *referring to* specific items in long-term memory (note that some images are created through a combination of both processes). The difficult aspect of this retrieval is determining this location: what does it mean for the agent to refer to a known location?

Recall that the Spatial Scene is organized as a tree, of objects and their parts (as discussed in Section 2.2). Internally, a coordinate transformation is associated with each of these relationships. This is clear in Figure 6: the **`tree`** object is part of the **`scene`** object, and there is a transformation between them specifying how the **`scene`** frame of reference relates to the **`tree`** frame of reference. The most obvious way to implement transformation recall is to simply use these transformations, which are explicitly represented in the underlying scene graph data structure of the spatial scene. In the case in the figure, the agent could retrieve an object from LTM, and specify that it is related to the **`scene`** the same way the **`tree`** is: the transformation connecting **`tree`** to **`scene`** can simply be copied to relate the new object to the **`scene`**. This is the way the current implementation of SVS works. Images can be added by specifying a source object and transform (called **`object-source`** and **`transform-source`**), along with a reference to the parent object in the scene that the new object will be part of. The **`object-source`** and **`transform-source`** can refer to objects in Perceptual LTM, by specifying a **`class-id`**, or to objects currently in the scene, by specifying an **`id`**. The image is positioned in the scene by applying the specified transformation below the provided parent object.

In practice, it is frequently desirable to add multiple objects at once to the scene, rather than build it up by adding objects one at a time. In this case, SVS allows entire symbolic scene structures, similar to those passed from SVS to working memory (Figure 5), to be used as imagery commands. This ability is called *scene composition*. A possible use for this ability is episodic memory retrieval: if the agent remembers the symbolic structure of the scene, and it is built out of components present in Perceptual LTM (that is, every object and transformation has a **`class-id`**), it can re-imagine that scene.

As discussed in Section 2.2, the means for symbolically referring to transformations between objects is an area of active research. The current implementation only deals with transformations between wholes and parts, a scheme which is sometimes inadequate. For example, if the agent wants to place block B to the right of block A, where "right of" is a reference to a specific mathematical transformation in long-term memory (the middle case of Figure 8), this is not simple to do: the current scheme would require

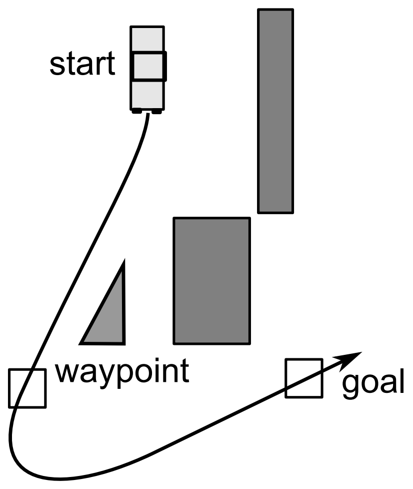


Figure 9: An example motion-related problem. The agent must determine whether or not the car can drive to the goal, via the waypoint, without colliding with an obstacle.

that the new block be added as a part of the existing block, which is undesirable. Partial support is present for adding images where non part-of transformations are specified, but more work is needed to enhance this capability.

2.6.3 Motion Simulation

While the previous approaches to image creation are powerful, it is difficult to see how they can encode the knowledge to solve spatial problems involving non-uniform motion. For example, consider the problem of predicting if a turning car will hit an obstacle. One approach to this problem might be to project an image into the scene that outlines the path that the car would follow around the corner, and check if this image intersects the obstacle. However, this presupposes that the system knows exactly what path the car will follow and has the ability to construct an image in that shape. For cases where the car is traveling in a straight line, this is feasible – the path image is a rectangle connecting the car to its destination, which can be created through predicate projection. In more complex cases, the path image may be difficult to construct and represent, such as when a car transcribes a path that is partially curved and partially straight. Moreover, a car with alignment problems might traverse a path that approximates a sine wave.

For a general agent that is capable of the same range of behavior as humans, there are many diverse types of motion that the system may need to predict. To plan its own behavior, a general agent needs to predict the outcome of actions it can perform in the environment, both relative to its own body ("What would happen if I reached for object X?"), and relative to objects it controls indirectly ("What would happen if I tried to back my car into this parking spot?"). In addition, it is useful for the agent to predict the motions of others and objects not under the control of any agent, such as the path of a bouncing ball. In general, the system should be able to internally reason about motions perceived in the environment.

However, it is very difficult (if not impossible) for an agent to reason about an arbitrary motion by generating an equivalent geometric shape, as can be done with a car moving in a straight line. This is roughly equivalent to requiring the system to have a closed-form definition of the result of any movement it might reason about. For example, in Figure 9, this would require the agent to be able to build the path shown from geometric primitives. A more straightforward way to solve this sort of problem is through continuous simulation: the system executes a forward model of the motion of a car, and observes whether or not the car hits the obstacle.

The general concept of combining qualitative symbolic reasoning with continuous motion simulation is not new (Funt, 1980 is an early example), but little work has been done to examine how, in general, continuous motion can be integrated into a complete system. In SVS, this information is encoded in the form of a continuous motion model, within the spatial level of the system. By transforming one continuous spatial state to another, these motion models provide the fine-grained quantitative resolution needed for detecting qualitative interactions between objects that is critical for correct reasoning in the symbolic level, which can "observe" the simulation through predicate extraction.

In SVS, patterns of motion are represented through structures called motion models inside perceptual LTM. Motion models can be applied to any object in the spatial scene, and are invoked with a step size. The model then steps forward the state of the object it is invoked for (called the primary object) a small amount of time. A motion simulation is then a sequence of steps, executed by the symbolic level. Between steps, qualitative information can be queried from the diagram and reasoned over. In the example in the figure, the agent can query for whether or not the car intersects any obstacles (or the goal or the waypoint) between each time step, gaining information about the simulation.

Note that this discretization provides a natural speed/accuracy tradeoff, with faster simulations using bigger step sizes that might miss intermediate states where queries temporarily match. Thus, the step size controls the effort an agent will expend on a given simulation.

A car motion model allows SVS to solve problems like that in the figure. As implemented, this motion model uses standard "simple car" equations (LaValle, 2006). Given a body angle, position, steering angle, and time step size, the next body angle and position can be easily calculated. As the body angle and position are present in the scene, the motion model must only determine the steering angle to use at each step, and feed it to the simple car equations to determine the next state.

A simple way for the symbolic level to provide the needed steering angle input to the model is to do it indirectly, by providing the object that is the target of the motion. The motion model can then determine the angle between the front of the car and the target, and steer in that direction, proportional to that difference, saturating at some maximum steering angle.

With this model, the symbolic system has the raw tools needed to plan a path: it determines what happens when a car tries to drive from one location to another. Then, path planning can be accomplished by decomposing the problem into a series of attempts to reach qualitatively specified waypoint images. An implemented version of this approach to path planning will be covered in Section 3.2.

Motion models can be used to simulate many kinds of motion (see Wintermute & Laird, 2008 for more examples). However, the agent's own effectors are an important class of moving objects that must be reasoned about. In a system incorporating real effectors, motion models could be intimately tied to their control. Recall that objects created in the scene as images are perceived by Soar (via predicate extraction) identically to objects fed into the scene from the environment, outside of an annotation to differentiate them. This allows hypothetical reasoning to occur identically to reasoning about the actual environment. Accordingly, hypothetical actions in the scene (motion simulations) and real actions can have the same representation in Soar, outside of an annotation.

This observation can help direct us toward how realistic actions should be integrated into a cognitive architecture using imagery. Mainly, at a high level, the actions should be controlled in the same way motion models are. In a system such as SVS, only relatively simple continuous models are needed to solve complex problems, since the qualitative system can easily take on the higher-level aspects of the problem that would be difficult to reason about in a purely-quantitative system. For example, in the car driving problem, the motion model itself needs only to know how to seek towards a location, not how to do a backtracking search for a path, which objects in the scene are obstacles and which are not, etc. An equivalent car controller, while still faced with the difficult task of reducing movement to actuator signals, would similarly not have to be concerned with those higher-level concepts.

The idea of tightly binding motor control and imagery was explored in detail by Grush (2004) in his emulation theory of representation. He points out that maintaining a prediction of the outcome of motor commands can aid the controller itself, shortening the feedback loop through the environment. If this predictor is present for the controller, it can also be used “offline” by itself to drive imagery – as a motion model.

In Figure 2, the motion simulation process accounts for this system. Based on symbolic commands, it instantiates motion models from perceptual long-term memory, creating images of moving objects in the scene. Conceptually, at least, this is the same system that controls the agent’s effectors, so it contributes to the $\boldsymbol{A}_{\mathrm{m}}$ signal, sending control commands to lower-level parts of the agent. It should be noted that no current implementation actually controls real effectors in this manner, however.

2.6.4 Visual Generation

All of the above processes have worked mostly with the spatial system of SVS, but the visual system can also be used during problem solving. If SVS were a more comprehensive model of human processing, perception would directly write to the visual buffer, and internal processing could derive the spatial scene from it. In imagery contexts, the visual buffer would be modified top-down, based on information in the spatial scene and symbolic working memory. This processes is called visual generation. Since SVS is currently used in virtual environments that provide the spatial scene directly, to use the visual system, its contents must first be derived from the spatial system through visual generation. That is, while SVS does not support visual perception, it does support visual imagery. To do this, the agent constructs a symbolic command, specifying which object(s) in the scene to render, along with parameters specifying the point of view to be used (the camera location). The result of this process is a new depiction being created in the visual buffer.

This process entails a conversion from 3D to 2D information, so it can be used directly to solve problems of the form “can object **A** be seen from the perspective of object **B**?”. To solve this sort of problem, a symbolic visual generation command is constructed, creating a depiction of object **A**, with the viewpoint set to the location of object **B** looking towards object **A**. If any pixels are present in this depiction (a property which can be detected through visual predicate extraction), the object is at least partially visible.

2.6.5 Top-Down Visual Recognition

Besides simple visibility problems, a use of the visual system in SVS is to allow recognition processes to be used that can leverage properties of the depictive representation there. Supporting visual recognition in general would be a major challenge, but allowing simple, limited recognition processes in certain domains can be very useful. This aspect was a focus of Lathrop's work with SVI (2008). He used "depictive manipulation" rules, encoding pixel transformations, to create new depictions based on existing depictions. These rules specified how to change pixel values based on surrounding values. With appropriate rules, the result of this process (the new "recognized" objects) can be meaningful depictions, such as an outline of the enclosed space in an object, or even likely paths for an agent to follow on a terrain map. Since the agent is creating symbolic commands to ask for these depictions to be created based on the problem, this is top-down control of visual recognition.

Some capabilities for this are present in SVS (inherited from SVI). However, this aspect (along with visual processing in general) will not be covered here in detail.

2.7 Interface to Symbolic Working Memory

If SVS is to be broadly applicable across many different problems, it is important that it be tightly integrated with symbolic processing in Soar. To understand how SVS has been designed to integrate with the existing parts of Soar, we must first outline some of the properties of processing in Soar.

Soar represents its state as a declarative structure, the working memory (or WM). An element of WM is called a WME (for Working Memory Element). Rules match on structures in WM, and cause changes within it. Reasoning in Soar is accomplished through a sequence of *decisions*, where an operator, representing a specific choice of an action, is selected. Different types of rules affect this decision process in different ways. Some rules, called state elaborations, create useful structures that remain in WM until the rule no longer matches, at which point the structure is removed. For example, suppose the agent's task is to pick up every red block it encounters. If block identities and object colors are encoded in separate locations of working memory (for example, **`^is-a-block block37`** and **`^is-red block37`**), an elaboration rule can bring this information together, and create a simpler structure which can be used to make a decision (e.g., **`^is-a-red-block block37`**). When any of the conditions that caused this structure to be created go away, the structure itself is automatically removed.

Other rules, called operator applications, cause changes in working memory that persist when the rule no longer matches. These are the results of decisions the agent has made. For example, the agent might wish to remember that it had encountered a block in a certain room, so it might create a structure like **`^block-in-room room45`** through an operator application at the time it sees the block. Even after the agent leaves the room, and it can no longer directly perceive the block, the structure remains until it is removed by another operator application. Other rules control which operator will be selected at a given time (and hence which operator application rules will fire). The details of this process are unimportant for this discussion, but it is important to understand that it is possible for the agent to create temporary structures that are automatically updated in response to changes in WM (through elaborations), and more permanent structures that must be deliberately added and removed from memory through operator applications. The former of these structures are called *i-supported* (for instantiation supported), and the latter *o-supported* (for operator supported).

Soar has built-in means for *subgoaling*. This is a process used to allow a task to be decomposed into smaller pieces. When an agent enters a subgoal, a special part of working memory is created (a substate), in which processing specialized for that subgoal can execute. Architectural mechanisms in Soar maintain these substates, and clean them up once the task of the subgoal has been completed. These substates give WM a hierarchical organization. The topmost state in this hierarchy is called **`top-state`**.

This overall design for Soar allows structures in working memory to be asserted as implications of other structures (elaborations), or as the result of decisions the agent has made (operator applications). Reasoning in Soar is guided by a fixed decision and subgoaling process, which controls when working memory structures are added and removed based both on rules encoding the assertions above, and on the overall role of the memory structure in the problem solving process. For example, the decision procedure applies operator application rules only after an operator has been selected, and automatically removes all of the state relating to a subgoal once that subgoal has been solved.

This means that structures in working memory are added and removed not solely based on what they represent, but also based on the role they serve in problem solving. Soar has many architectural elements that facilitate this automatic truth maintenance, which serves an important role in allowing functional agents to be built.

The overall motivating principle behind the design of the interface between SVS and working memory is that SVS processing should be considered an extension of symbolic processing in working memory. The agent should be able to use images in SVS in the same types of problem-solving roles that plain working memory structures can fulfill, such as structures elaborated inside a subgoal, or structures globally asserted at the top-state as a result of a decision. In addition, the agent should be able to make inferences about information in SVS in all of the same contexts that inferences can be made in working memory; for example, it should be able to infer that two particular objects intersect just as if there were a symbolic rule detecting the property.

SVS can be viewed as a process that works like a set of hidden productions, matching against structures at different places in working memory and producing new structures in response. Some of these structures (perceptual pointers) have a corresponding underlying sub-symbolic state in SVS. This is in contrast to the view that SVS is an external environment with respect to the agent that receives commands and gives back responses through Soar's I/O links. Although both viewpoints are technically correct, the latter does not bring to light the complex way SVS is integrated with existing Soar processing.

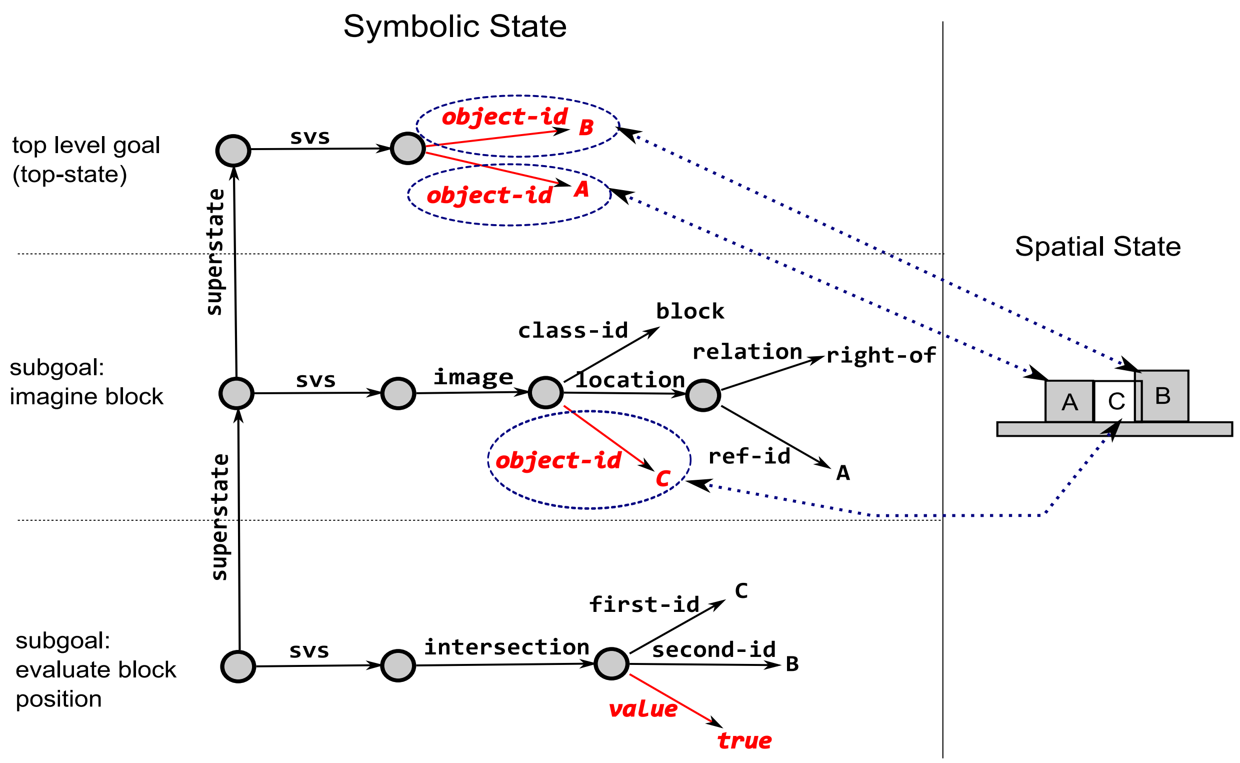


Figure 10: An example working memory state showing associations with spatial state and processing in SVS. The agent is performing a simple blocks world problem. It has subgoals to imagine a potential block placement, and to evaluate that placement. Structures in red italics are added to WM by SVS processing. Structures in blue circles have corresponding state in the spatial system. In this problem, SVS has provided perceptions of two real blocks to the symbolic system (A and B). Soar creates a subgoal, in which a new block (C) is imagined to the right of A. SVS creates this block in response to the symbolic structure on the svs link in that subgoal, and will remove it once that structure goes away. After imagining the block, another subgoal is created to evaluate if the potential block placement is good. SVS processing matches on the query structure in this subgoal, and provides the result.

This is a simplification to show the concepts, implemented SVS WME names and structures differ slightly.

Figure 10 shows a comprehensive example of this interface. Some structures in working memory (those in red) are added by SVS, either through bottom-up perception (the top-level objects), or as a result of matching certain structures in memory (imagined objects, and extracted properties). Perceptual pointers are present for objects perceived by bottom-up perception or created through imagery. Note that other working memory structures refer to underlying state in SVS that aren't shown as perceptual pointers in the figure. For example, the intersection query refers to blocks C and B, but the WMEs in that command structure aren't directly linked to underlying SVS state: if the conditions of the query were removed from WM, the items referred to would remain.

We can call these structures perceptual references, which are weaker than perceptual pointers[9]. Processing in Soar creates references simply by copying the string value of the item pointer. If the

[9] This distinction between perceptual pointers and references should not be directly mapped onto the difference between pointer and reference types in programming languages such as C++. In this case, I'm using the terms closer to their nontechnical definitions: physically *pointing* to an object is a very direct way of *referring* to that object.

original item pointer goes away, along with its underlying SVS structure, this reference might still exist in WM, and will cause an error if used.

The SVS processing that matches on working memory structures and creates new structures is considered to be similar to a set of elaboration rules. That means that structures placed in WM by SVS processing are supported by the structure that was matched on to initiate that processing. Once the WMEs describing the imagined block **`C`** in the figure go away, so does the imagined block itself. This scheme allows SVS structures to piggyback on the truth maintenance system in Soar, SVS structures inherit the support and subgoal properties of the WME structures that describe them. For example, if the specification for an image is o-supported at the top state, the corresponding state SVS will effectively also be o-supported and part of the top-state.

This scheme for integrating SVS and Soar has not been directly implemented: Soar actually connects to SVS through the **`io-link`** at the **`top-state`**, like an external environment. However, most aspects of the conceptual interface have been approximated with a set of library production rules that can be included in any system using SVS. The largest unimplemented aspect of the interface is properly filtering what perceptual items are available to processing in each subgoal. As implemented, perceptual pointers resulting from imagery are mirrored at the top-state: in the example in Figure 10, as implemented, a duplicate **`top-state.svs.object-id C`** structure will appear while the image command in the substate is present. This means that processing in Soar's top-state has access to an image created locally by one of its substates, which would not be possible for a typical locally-created working memory structure. Fixing this sort of conflict, which can be detrimental when building larger agents, is an area for future work.

For most types of interactions between Soar and SVS, the change to the memories of SVS is monotonic: new structures might be created, but other structures aren't removed or otherwise changed. This is not always true, though. For a few types of interactions, non-monotonic changes are required. For example, in the current system, the spatial scene is related to the visual buffer through a certain viewpoint (or camera position), which affects the visual generation process. There is only one such viewpoint, so changing it necessarily invalidates the old viewpoint. For non-monotonic processing like this, then, the command structure must always be at the top-state, since it is impossible for subgoals to use their own local, possibly conflicting, viewpoints.

The way that SVS processing is conceptually tightly tied to the details of Soar working memory is one difference between SVS and its predecessor system, SVI (Lathrop, 2008), which views perceptual processing as a specialized set of operators in Soar, rather than an extension of symbolic working memory.

3 Imagery Agents

We have now defined a basic system for visual and spatial processing within a symbolic cognitive architecture. In this section, the discussion will focus on how agents can be instantiated within this architecture to effectively solve problems.

Several agents have been instantiated in Soar/SVS and its predecessor systems. Here, we will focus on two agents that show the functionality of the spatial system as a whole. Previous agents have explored particular processes present in SVS in detail. Agents focusing on the predicate projection component can be found in (Wintermute & Laird, 2007), agents focusing on motion simulation can be found in (Wintermute & Laird, 2008), and an agent using visual imagery can be found in (Lathrop & Laird, 2009).

3.1 Blocks World Agent

Consider a slightly-modified version of a classic blocks world problem, the pegged blocks world. The goal in this problem is to stack four blocks in a particular configuration, **`A`** on top of **`B`** on top of **`C`** on top of **`D`**. Unlike the standard blocks world, the blocks cannot be placed freely on a table, rather there are two fixed pegs, and each block has a groove down its back that must be aligned to one of the pegs—essentially, there can only be two towers in the world, and their positions are fixed. Blocks can be moved from the top of one tower to the other, however, the blocks vary in size, and the pegs are close enough together that they may collide, depending on the exact sizes of the other blocks in the towers. Blocks can also be moved out of the way to a storage bin. Assume that moves between the towers are cheap (cost 1) and moves to and from the bin are expensive (cost 20). In addition, collisions are very expensive (cost 1000). So it is in the agent's best interest to solve a problem by moving blocks between the towers, using the bin only if absolutely necessary, and never causing collisions by attempting to move a block where it cannot fit (the same domain was used in Wintermute and Laird, 2009).

This problem can be represented in terms of both an abstract and concrete state. Assume that the agent uses a similar abstract state to what is normally used in the blocks world. The state includes symbols for the important objects in the world (the blocks, bin and pegs). Predicates about these objects are also encoded: **`on(X,Y)`**, indicating that block **`X`** is on object **`Y`** (which could be the base of a peg, the bin, or another block), **`clear(X)`**, indicating that block **`X`** is clear, and **`collided(X,Y)`**, for when blocks **`X`** and **`Y`** have collided. The initial abstract state of the instances we will consider is **`[on(A,peg1) on(B,peg2) on(C,bin) on(D,B)]`**, and the goal state is **`[on(A,B) on(B,C) on(C,D) on(D,peg2)]`**. In addition to the abstract state, a concrete perceptual state is also present—the exact sizes, shapes, and positions of the blocks are encoded in terms of continuous numbers. An example of the initial state is shown in Figure 11.

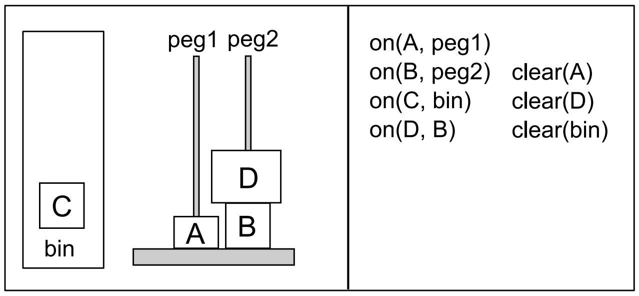


Figure 11: A pegged blocks world problem, represented concretely (left), and abstractly (right).

In the agent, the move actions can be simulated—the agent can use imagery to predict what would happen in the concrete state if a given block were to be moved, and based on that, can extract a description of the (hypothetical) next abstract state. Assume that imagery operations have no cost, as they are internal. With this capability, a plan for this problem can be expressed (Figure 12). The agent makes its move choices based on the abstract state, however, instead of acting solely in the world, it can also perform imagery actions. Then, based on the results of imagery, an external action can be chosen. Imagery will sometimes provide differing predictions for states that are identical at the abstract level: for example, all problem instances have the same initial abstract state, but in some instances moving block **D** to the top of block **B** will cause a collision, and in some instances it will not. This is represented in the plan as a branch: the next abstract state reached, and the associated action chosen differs based on the results of imagery. This happens at several points in the plan, but it is not necessary to use imagery before every move (for instance, moving a block to the bin is always successful, so there is no reason to simulate it).

The plan in the figure shows how the agent acts in four canonical cases where potential collisions can arise at different points.[10] In case 1, there is no way to productively move blocks between pegs; instead, they must all go into the bin before building the goal stack. In case 2, this is also true, but an agent using this plan cannot determine this until it has already made one movement (of **D** to **A**). In case 3, block **D** never has to be moved to the bin, and in case 4, neither **A** nor **D** ever need to be moved to the bin.

[10] These cases may or may not be representative of classes which exhaustively cover all possible instances with the presented initial abstract state. They seem to, but more work is needed to prove this or find a counterexample.

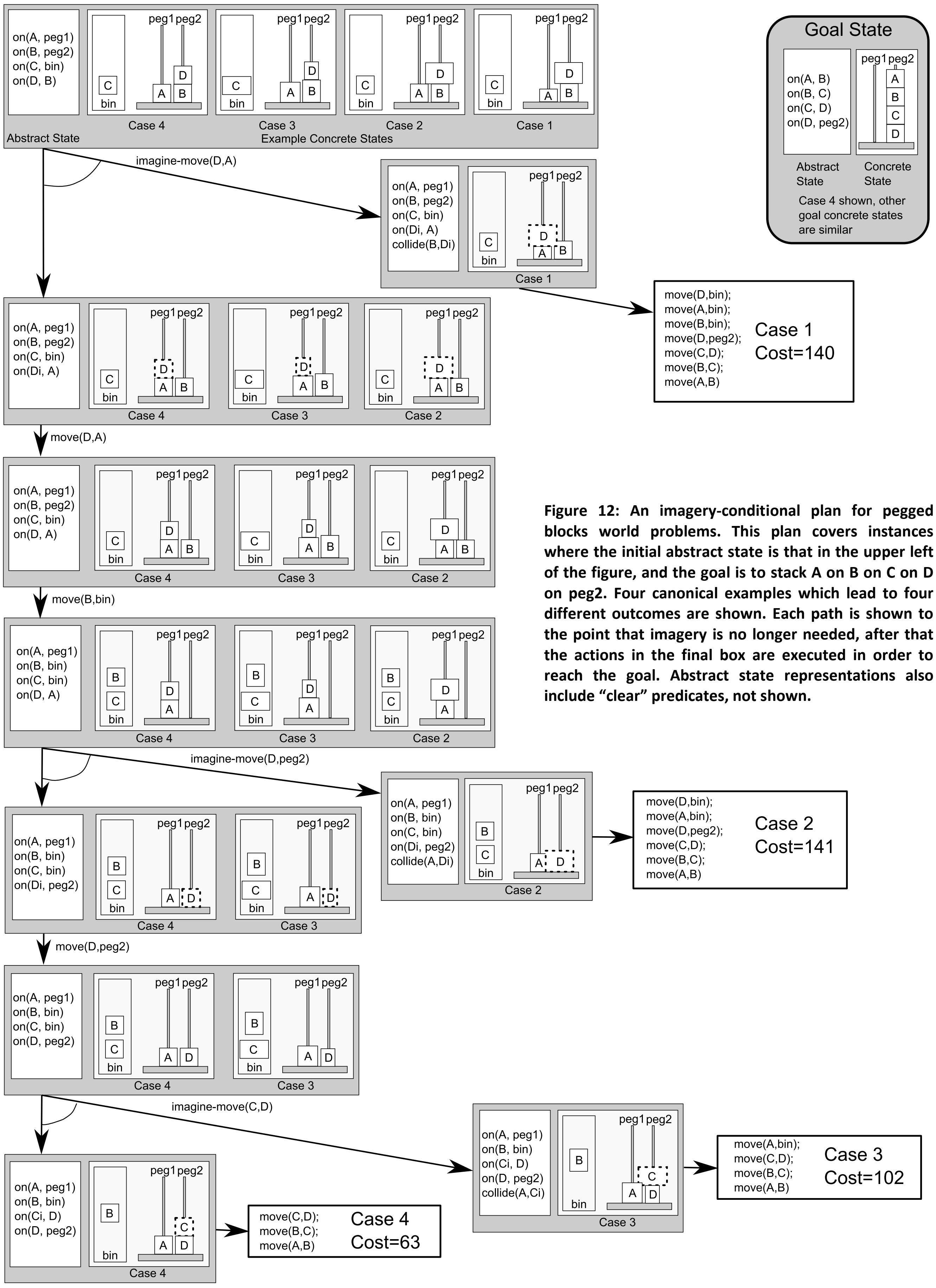


Figure 12: An imagery-conditional plan for pegged blocks world problems. This plan covers instances where the initial abstract state is that in the upper left of the figure, and the goal is to stack A on B on C on D on peg2. Four canonical examples which lead to four different outcomes are shown. Each path is shown to the point that imagery is no longer needed, after that the actions in the final box are executed in order to reach the goal. Abstract state representations also include "clear" predicates, not shown.

An agent following this plan has been implemented in SVS, and will be described here.

A trace of the agent running in an instance covered by Case 1 of the plan is shown in Figure 13. There are three basic types of operations the agent performs: initialization, plan execution, and imagery, which are color coded in the figure.

Upon starting, the first operation necessary is to construct the problem inside SVS's spatial scene. SVS supports connecting agents to external (simulated) environments, but for practical reasons it is sometimes desirable to use the system's imagery capability to build the initial scene contents. SVS library productions encode somewhat generic means to do this, the details of which will not be covered here. The first 7 decisions are occupied by this (**svs-build-secondary-structures** requires three application waves to complete, causing impasses at decisions 6 and 7). After building the world, the agent sets the camera position, making the blocks visible in the visual buffer. This does not affect functionality, but is useful for debugging, as the visual buffer is visible to the user. Then, the **create-start-state** operator initializes some bookkeeping state, allowing plan execution to begin.

As plan execution begins, the agent first needs to determine the initial symbolic state. This symbolic state is extracted from the scene; it is not knowledge that has been given directly to the agent. The state needed is a set of **on** and **clear**, and **collision** predicates. Recall that the basic properties that can be extracted from the scene in SVS are distance, direction, and topology. With the exception of **collision** (which is a basic topological property), these predicates cannot be extracted directly from the scene, but must be built out of more primitive information.

The relationship **on** can be reduced to more primitive terms by saying object **A** is **on** object **B** if **A** is above **B** (a direction property), and no other object that **A** is also above is closer to **A** than **B** is (a distance property). This definition is sufficient for blocks world, but a more

```
 1: O: O19 (svs-build-world)
 2: ==>S: S7 (operator no-change)
 3:    O: O83 (svs-build-temporary-structures)
 4:    O: O117 (svs-build-base-structures)
 5:    O: O142 (svs-build-secondary-structures)
 6:    ==>S: S9 (operator no-change)
 7:       ==>S: S11 (state no-change)
 8: O: O167 (set-camera)
 9: O: O169 (create-start-state)
10: O: O179 (wait-for-svs)
    blockC is on bin.
    blockD is on blockB.
    blockB is on point2.
    blockA is on point1.
11: O: O180 (imagine-action)
12: ==>S: S18 (operator no-change)
       blockC is on bin.
       blockD is on blockB.
       blockA is on point1.
       blockB is on point2.
13:    O: O191 (move)
14:    O: O195 (wait-for-svs)
       Moving block blockD has
         collided with blockB!
       blockD is on blockA.
15: O: O183 (move)
16: O: O208 (wait-for-svs)
    blockD is on bin.
17: O: O215 (process-cost)
    Cost: 20
18: O: O218 (move)
19: O: O235 (wait-for-svs)
    blockB is on bin.
20: O: O242 (process-cost)
    Cost: 20
21: O: O244 (move)
22: O: O262 (wait-for-svs)
    blockA is on bin.
23: O: O269 (process-cost)
    Cost: 20
24: O: O271 (move)
25: O: O283 (wait-for-svs)
    blockD is on point2.
26: O: O284 (process-cost)
    Cost: 20
27: O: O290 (move)
28: O: O298 (wait-for-svs)
    blockC is on blockD.
29: O: O299 (process-cost)
    Cost: 20
30: O: O304 (move)
31: O: O311 (wait-for-svs)
    blockB is on blockC.
32: O: O312 (process-cost)
    Cost: 20
33: O: O316 (move)
34: O: O322 (wait-for-svs)
    blockA is on blockB.
35: O: O323 (process-cost)
    Cost: 20
Goal achieved, total cost 140
```

Figure 13: Trace of a Soar/SVS agent executing case 1 of the pegged blocks world plan in Figure 12.

Color-coded:
initialization
imagery
plan execution

general-purpose definition might replace the distance term with a topological relationship: the surface of **A** is tangential to that of **B** (that is, **A** is above **B** and touches **B**). However, the current implementation cannot extract tangential topological relationships, so this definition is not used.

The agent computes on as described as it relates blocks to each other, and to the bases of the pegs (objects called **point1** and **point2** in Figure 13). The bin is handled separately, an object is considered **on** the bin if it intersects the **bin** object. An agent using a richer representation of a bin could likely use the same definition of **on** for it as is used for the other objects, however.

So to compute the **on** predicate, the agent builds queries to extract the following information from the scene:

property 1a: For each pair (X,Y) of blocks, whether X is below Y.

property 1b: For each block X and peg-base Y, whether X is below Y.

property 2a: For each pair (X,Y) of blocks, the distance from X to Y.

property 2b: For each block X and peg-base Y, the distance from X to Y.

property 3: For each block X, whether X is discrete from the bin object.

The predicate is computed as follows:

For all blocks X where property 3 is false, it **on(**X**,bin)** holds; otherwise, for all objects Y where property 1 holds, find the minimum value of property 2 (using the same X,Y)—**on(**X**,**Y**)** holds for that assignment.

The **clear** predicate is derived from the **on** predicate: if no block is **on** block or peg-base X, **clear(**X**)** holds. In addition, **clear(bin)** always holds.

To determine **collision**, the following property is extracted:

property 4: For each unique pair of blocks (X,Y), where neither **on(**X**,bin)** nor **on(**Y**,bin)** holds, whether X and Y are discrete.

This property can be directly translated to **collide(**X**,**Y**)** if it holds. To simplify the current implementation, blocks in the bin are placed all at the same location, so it is necessary to exclude those blocks from the collision query.

To determine the state predicates as described, many atomic predicates (the properties above) need to be extracted from the scene—roughly 50 in a world with 4 blocks. These are cheap monotonic operations; they do not change the contents of SVS and hence do not interact with one another, so it makes sense to perform the operations in parallel, rather than serially over many decisions. Conceptually, these queries are similar to Soar elaborations, rules that derive monotonic implications of working memory state. Soar can process many elaborations in parallel, and does so in several phases within a decision. However, as SVS is implemented as an external environment to Soar, it can only interact with working memory *between* decisions. So in situations where elaboration-like processing happens in SVS, Soar takes an extra decision while SVS processes. This decision is filled by the **wait-for-svs** operator in the trace. The operator is part of the SVS library productions, and does not have to be deliberately proposed by the agent designer.

Once the agent establishes that an **`on`** or **`collide`** predicate holds, it prints out a message reporting it (**`clear`** predicates are not printed). One thing to note is that Soar's truth maintenance system allows the agent to process only changes: after a **`move`** operator is applied, only queries involving the moved block are calculated, other queries, along with their resulting working memory structures, remain. This is evident in the trace, as only one new **`on`** predicate is reported after every **`move`** (the rest still hold from the previous state).

Since this agent is not connected to any environment (its entire world is created through imagery), the **`move`** actions are implemented as imagery commands. A **`move`** action creates an image of the moving block centered on top of the target object (an indirectly specified predicate projection). As the new specification is added to working memory, the specification of the block at its previous location is removed, causing the corresponding object in SVS to disappear. Starting in the next decision, the newly added block image is considered 'real' by the agent.

While acting in the problem, the agent keeps track of the cost of each action, and accumulates it through the **`process-cost`** operator. Note that the agent has to re-perceive the scene to extract the new state in order to determine the cost (a collision may have occurred), so an extra **`wait-for-svs`** decision is needed after any action.

In this trace, there is one imagery subgoal (this is case 1 in the plan, so imagery is only checked before the first step). The processing in this subgoal is almost identical to processing in the superstate; in fact, the same productions that determine the state predicates and implement the **`move`** action match in both places. This is a capability SVS's tight integration with Soar's working memory structure enables: productions build local structures on the **`svs`** link of any state they happen to match on, and when that state is removed by Soar (e.g., if an impasse is resolved), the corresponding structures in SVS are automatically cleaned up. In the subgoal here, an image of the moved block is created, the next state is inferred, and the result (whether or not a collision occurred in the imagined next state) is returned to the superstate. The presence of this result causes the subgoal to retract, and with it the image of the imagined block and the queries to determine the imagined next state are all automatically removed from SVS.

This agent can handle (and has been tested with) all four cases covered by the plan.

3.2 RRT Motion Planning

In this section, we will examine a Soar/SVS agent constructed to perform motion planning for a car-like robot. Motion planning is the problem of determining a sequence of actions that will move an object (such as a robot) through space to a goal, while avoiding obstacles. The version of the problem considered here is very constrained: the locations of all obstacles and the goal are known a priori, mapping and localization are not a concern.

Motion planning research has usually been pursued outside of the context of creating generally intelligent agents. Earlier approaches focused on efforts to exactly compute optimal paths for particular classes of robots, such as polygon robots that can move in any direction. This involves computing exactly the configuration space of the robot, a space in which any path corresponds to a real path to robot is able to follow. As motion planning has progressed to address problems involving more and more realistic robots, however, this exact computation has become intractable (Lindemann & LaValle, 2003).

```
make tree rooted at initial state
while tree does not reach goal
        generate random state -> Xr
          or choose goal state -> Xr
         with some probability
        get closest existing state to Xr -> Xc
        extend Xc towards Xr -> Xn
        if no collision occurred
                add Xn to the tree, connected to Xc
```

Figure 14: RRT Algorithm

One reason for this difficulty is that certain kinds of constraints on motion are infeasible to capture in representations like configuration spaces. Nonholonomic constraints result from systems where the number of controllable dimensions is less than the total number of degrees of freedom. For instance, a car is nonholonomic, since its position can be described by three parameters (x, y, and an angle), but it is only controllable in two dimensions (driving forward and reverse, and steering left and right). Where it is relatively straightforward to calculate the configuration space of a robot that can turn in place, this is not as simple with a car-like robot. In addition to nonholonomic constraints, traditional geometric motion planning approaches also have trouble incorporating dynamic constraints, where the path of the robot is affected by dynamics, such as a car that can't stop without slowing.

Recently, sampling-based approaches have become popular for planning with dynamic and nonholonomic constraints (LaValle, 2006). In sampling-based motion planning, the goal is not to exactly compute the configuration space, but instead to sample it through simulation. While previous approaches required difficult-to-calculate specialized representations that were specific to the particular motion under consideration, motion planning through simulation requires only a basic spatial representation, as details particular to the motion are encapsulated in the controller. If the simulation is accurate, motion plans can be guaranteed to meet nonholonomic and dynamic constraints.

This development from computation of configuration space to sampling reflects only two of many prominent techniques in motion planning. It is also worth mentioning behavior-based approaches, where representations are eschewed, and motion planning emerges from relatively simple mappings from perceptions to actions (e.g., Brooks, 1991) While the approach in this section most certainly involves internal representations, it allows techniques developed in this tradition to be used as controllers within a broader symbolic AI system (in our case, a set of differential equations from Fajen & Warren, 2003, are used).

RRT (Rapidly-exploring Random Trees, LaValle, 2006) is a sampling-based motion planning algorithm that works by constructing a tree of states of the robot, rooted at the initial state, and adding nodes until that tree reaches the goal. Nodes are generated by extending the tree in random directions, in such a way that it will eventually reach the goal, given enough time. Each path from the root of the tree to a leaf represents a path that the robot could take, constantly obeying all constraints on its motion. The tree is constructed by the algorithm in Figure 14.

Two of the steps in this figure hide the true complexity of the algorithm. The steps to get the closest node to the random state, and to extend that node towards the random state, can both take substantial computation. This computation is also specific to the exact problem being solved, where the rest of the algorithm is general to all motion planning problems.

To determine the closest existing state to a random state, some metric must be determined that can measure the distance between states. In the case of car path planning, a simple metric is the Euclidian distance between the position of the car in the two states, with the condition that the distance is infinite if the target state is not in front of the source.

The other problem-specific step in the algorithm is extending the chosen node towards the new state, while detecting collisions along the path. A typical approach is to numerically integrate differential equations that describe the vehicle dynamics to simulate motion, resulting in a sequence of states parameterized by time. This simulation must occur within a system capable of detecting collisions.

The RRT algorithm has been instantiated in a Soar/SVS agent (Wintermute, 2009). The problem considered is that of planning to drive a car from an initial state to a goal region, while avoiding obstacles in a known environment (the agent only determines a plan, it is not connected to an actual robot). The car motion model takes as input the identity of a car in the scene, and the location of a goal. Inside this model, a system of differential equations that describe the configuration of a simple car-like vehicle as a function of the time and goal location is used. When integrated, these equations can yield a sequence of configurations parameterized by time, allowing for simulation. These equations were determined by combining a model of human movement and obstacle avoidance (Fajen and Warren, 2003) with a simple car model (LaValle, 2006). The human model controls the intended steering angle of the car, and this steering angle determines the next position of the car. A constant speed is assumed.

The controller locally avoids obstacles: each obstacle affects the steering of the car, with nearer obstacles located towards the front of the car having the most influence. This reactive obstacle avoidance alone can solve simple problems, but more complicated problems can’t be solved through reactive steering alone, as a solution needs to be composed out of several distinct movement subgoals.[11]

The controller simulates motion towards a goal, while maintaining the nonholonomic constraints of the vehicle. Along with geometric models of the car and world in the LTM of SVS, it is the low-level knowledge that was added to the existing SVS system to implement this planner.

Symbolic Soar rules were written to perform the algorithm in Figure 14. The algorithm relies on existing architectural functionality in the interface between Soar and SVS. As a metric for node distance, our implementation used the Euclidean distance, with the condition that the distance is infinite where the goal is not “in front” of the node. SVS predicate extraction mechanisms were used to extract distances, and to query for an in-front relationship. The motion model described above enables simulation, and SVS supports querying for intersections between objects in the scene, enabling collision detection. The only new mechanism needed in SVS to support this algorithm was a predicate projection method to generate random goal points in the scene, which was a simple addition.

[11] The local obstacle-avoiding controller here was directly compared (with favorable results) to a similar controller to a controller that simply steers towards the goal in (Wintermute, 2009).

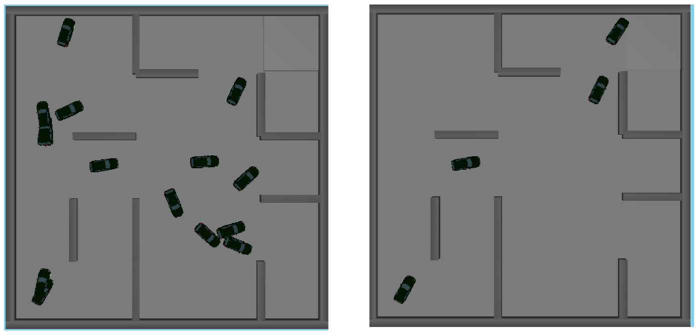


Figure 16: States of SVS Spatial Scene during RRT planning. The problem is to drive a car from lower-left to upper-right. Left: RRT tree, just before a solution is found. Right: Sequence of car positions that solve the problem.

Examples of the SVS scene during RRT planning are shown in Figure 16, and an example partial trace of Soar decisions is shown in Figure 15. Soar stores the RRT tree as a symbolic structure in working memory. The nodes in that tree are perceptual pointers into SVS—they point to specific objects in the scene, which can be seen in the figure. Soar proceeds by adding new random point objects to the scene, and querying for the distance from each node to that object. These distances are then compared to find the closest. A simulation is instantiated with that node as the initial condition (creating a new car object in the scene), and this simulation is stepped until a certain time is reached, the goal is reached, or Soar detects a collision with an obstacle. In all but the last case, the termination of the simulation results in a new tree node being added. In addition to moving towards random points, with a certain probability the agent instead tries to extend the tree directly towards the overall goal, biasing the tree in that direction.

The decision trace in Figure 15 shows six iterations of the algorithm. Initialization is similar to the previous example, and need not be covered again. Each iteration starts with a decision to either generate a random point to use as a goal (seen as a **`generate-projection`** decision), or a decision to try and extend the tree towards the overall goal (seen as a **`choose-goal`** decision in the last iteration). After deciding on the location to extend towards (a target), the agent must determine which existing state (which node in the tree)

```
  1: O: O22 (svs-build-world)
     ...
  9: O: O291 (rrt-plan)
 10: ==>S: S10 (operator no-change)
 11:    O: O293 (init-tss)
 12:    O: O294 (generate-projection)
 13:    O: O300 (wait-for-svs)
 14:    O: O301 (rrt-extend)
 15:    O: O303 (run-svs-simulation)
 16:    ==>S: S12 (operator no-change)
 17:       O: O343 (svs-step)
          ...
 37:       O: O363 (svs-step)
 38:    O: O365 (register-node)
 39:    O: O366 (generate-projection)
 40:    O: O371 (wait-for-svs)
 41:    O: O372 (rrt-extend)
 42:    O: O374 (run-svs-simulation)
 43:    ==>S: S14 (operator no-change)
 44:       O: O414 (svs-step)
          ...
 59:       O: O429 (svs-step)
 60:    O: O431 (remove-failed-sim)
 61:    O: O432 (generate-projection)
 62:    O: O437 (wait-for-svs)
 63:    O: O438 (behind-all)
 64:    O: O439 (generate-projection)
 65:    O: O444 (wait-for-svs)
 66:    O: O445 (rrt-extend)
 67:    O: O447 (run-svs-simulation)
 68:    ==>S: S16 (operator no-change)
 69:       O: O487 (svs-step)
          ...
 82:       O: O500 (svs-step)
 83:    O: O502 (register-node)
 84:    O: O503 (generate-projection)
 85:    O: O508 (wait-for-svs)
 86:    O: O509 (rrt-extend)
 87:    O: O511 (run-svs-simulation)
 88:    ==>S: S18 (operator no-change)
 89:       O: O551 (svs-step)
          ...
 99:       O: O561 (svs-step)
100:    O: O563 (remove-failed-sim)
101:    O: O567 (choose-goal)
102:    O: O569 (rrt-extend)
103:    O: O571 (run-svs-simulation)
104:    ==>S: S20 (operator no-change)
105:       O: O611 (svs-step)
106:    O: O613 (remove-failed-sim)
107:    ... etc.
```

Figure 15: Trace of a Soar/SVS agent executing RRT planning. Initialization shown in blue, iterations of the algorithm shown in alternating font styles.

should be extended toward that state. This involves predicate extraction queries: orientation queries are set up so that the in-front-of relationship is checked between each node in the tree (which is a pointer to an actual car object in the scene) and the target object, along with distance queries between those. Of all of the nodes for which the orientation query matches, that with the shortest distance is chosen. If a random point is generated that is not in front of any node, it is rejected, this is the **`behind-all`** operator in the third iteration. Note that queries are again sent to SVS in parallel, but an extra **`wait-for-svs`** decision is needed to evaluate them after generating a random point, since that new object won't be present until the end of the **`generate-projection`** decision.

After finding a node to extend, the agent uses a motion model to simulate the car driving towards the target. This is set up through the **`rrt-extend`** operator. SVS includes task-independent library productions to allow easy creation of motion simulations. Here, the agent specifies to simulate moving the car towards the target, until either a maximum time is reached (the common case for a successful extension), or a predicate extraction query matches, detecting that the car has collided with an obstacle or reached the goal. As the motion model used avoids obstacles, the perceptual pointers to the obstacles in the problem must also be provided to the SVS motion simulation process. After the relevant queries are set up, the operators **`run-svs-simulation`** and **`svs-step`** (part of the SVS library) take over. Based on the results of the simulation, the image of the car at its new location is either added as a new node (if the extension was successful), or removed from the scene (if a collision occurred). This continues until the overall goal is reached, at which point the agent removes all of the non-solution nodes in the tree (as seen on the right of Figure 16). The agent halts at this point. In a more complete agent, the plan would then be executed by moving the robot towards these locations.

4 Summary

The design of SVS as it relates to spatial processing has been presented, along with agents that use it.

SVS includes specialized short- and long-term memories for visual and spatial information. These memories are accessed by higher-level processing in Soar through a symbolic interface. Perceptual pointers and the predicate extraction process provide symbolic processing with information about the contents of the spatial memories of SVS. These memories can be modified by symbolic processing through imagery; images can be created through qualitative predicate projection, by simulating motion, or by retrieving memories. A library of Soar productions allow SVS structures to be tightly integrated with symbolic working memory.

A Soar/SVS agent to solve a pegged blocks world problem has been presented, along with an agent to solve motion planning problems. These agents perform very different tasks, but use the same basic pieces of architectural functionality to perform them; they differ only in the knowledge present in the systems, not in the architecture.

5 References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, *111*(4), 1036-1060.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, *47*, 139-159.

Chandrasekaran, B. (1997). Diagrammatic representation and reasoning: some distinctions. In *AAAI Fall Symposium on Diagrammatic Reasoning*. Boston, MA.

Chown, E., Kaplan, S., & Kortenkamp, D. (1995). Prototypes, location, and associative networks (PLAN): Towards a unified theory of cognitive mapping. *Cognitive Science*, *19*(1), 1-51.

Cohn, A. G., Bennett, B., Gooday, J., & Gotts, N. M. (1997). Qualitative Spatial Representation and Reasoning with the Region Connection Calculus. *GeoInformatica*, *1*(3), 275-316.

Eberly, D. H. (2004). *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

Fajen, B. R., & Warren, W. H. (2003). Behavioral dynamics of steering, obstacle avoidance, and route selection. *Journal of Experimental Psychology: Human Perception and Performance*, *29*(2), 343-362.

Funt, B. V. (1980). Problem-solving with diagrammatic representations. *Artificial Intelligence*, *13*, 201–230.

Grush, R. (2004). The emulation theory of representation: Motor control, imagery, and perception. *Behavioral and Brain Sciences*, *27*(03), 377-396.

Hernández, D. (1994). *Qualitative Representation of Spatial Knowledge* (p. 202). Springer.

Kosslyn, S., Thompson, W., & Ganis, G. (2006). *The Case for Mental Imagery*. New York: Oxford University Press.

Laird, J. E. (2008). Extending the Soar Cognitive Architecture. In *Proceedings of the First Conference on Artificial General Intelligence*.

Lathrop, S. D. (2006). *Incorporating Visual Imagery into a Cognitive Architecture: An Initial Theory, Design, and Implementation*. Technical Report, University of Michigan Center for Cognitive Architecture.

Lathrop, S. D. (2008). *Extending Cognitive Architectures with Spatial and Visual Imagery Mechanisms*. PhD Thesis, University of Michigan.

Lathrop, S. D., & Laird, J. E. (2007). Towards Incorporating Visual Imagery into a Cognitive Architecture. In *Proceedings of the Eighth International Conference on Cognitive Modeling*.

Lathrop, S. D., & Laird, J. E. (2009). Extending Cognitive Architectures with Mental Imagery. In *Proceedings of the Second Conference on Artificial General Intelligence*.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

Lindemann, S. R., & LaValle, S. M. (2003). Current issues in sampling-based motion planning. In *Proceedings of the International Symposium of Robotics Research*. Springer.

Pylyshyn, Z. W. (2001). Visual indexes, preconceptual objects, and situated vision. *Cognition*, *80*(1-2), 127-158.

Wang, Y., & Laird, J. E. (2006). *Integrating Semantic Memory into a Cognitive Architecture*. Technical Report CCA-TR-2006-02, University of Michigan.

Wintermute, S. (forthcoming). Representing Problems (and Plans) Using Imagery. *Submitted to the AAAI Fall Symposium on Multi-representational Architectures*.

Wintermute, S. (2009). Integrating Reasoning and Action through Simulation. In *Proceedings of the Second Conference on Artificial General Intelligence*.

Wintermute, S., & Laird, J. E. (2007). Predicate Projection in a Bimodal Spatial Reasoning System. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-07)*. Vancouver, BC.

Wintermute, S. & Laird, J.E., (2008). Bimodal Spatial Reasoning with Continuous Motion. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08)*. Chicago, IL.

Wintermute, S., & Laird, J. E. (2009). Imagery as Compensation for an Imperfect Abstract Problem Representation. In *Proceedings of the 31st Annual Conference of the Cognitive Science Society*.

Wintermute, S., & Lathrop, S. D. (2008). AI and Mental Imagery. In *AAAI Fall Symposium on Naturally Inspired AI*. Arlington, VA: AAAI Press.