SRI International

A KNOWLEDGE-BASED ARCHITECTURE FOR ORGANIZING SENSORY DATA

Technical Note No. 399

December 16, 1986

By: Grahame B. Smith and Thomas M. Strat

Artificial Intelligence Center
Computer Science and Technology Division

This paper also to appear in the Proceedings of the International Autonomous Systems Congress.

SRI International®

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

A Knowledge-Based Architecture for Organizing Sensory Data

Grahame B. Smith and Thomas M. Strat

Artificial Intelligence Center
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025

This paper describes an architecture for an information manager that is at the core of a sensor-based autonomous system. The architecture provides the means by which sensor-based data can be integrated with stored knowledge to provide the information needed for autonomous behavior. The overall architecture can be viewed as a community of independent processes, each of which interact with an active database whose structure mirrors that of the three-dimensional world.

1 Introduction

This paper describes the architecture of an information manager that is at the core of a sensor-based autonomous system. The architecture accomodates data from a wide variety of sensors as well as from other sources of stored knowledge. Each source of information imposes its own constraints upon the system design; however, our design decisions were motivated by the requirements for organizing visual data. More general sensory data includes visual data, but the information that all sensors capture is information about the visual world — information that is comprehendable only in terms of the visual world. The task of organizing this information is the task of providing a framework for data about the world, whatever the data source (sensor type). The visual world and the means for structuring data sensed from that world are thus the subjects of this paper.

Current models of machine vision describe the vision process as a series of tasks that convert the visual signal into a set of symbols that characterize the entities in the scene. The series of tasks from signal to symbol is commonly described by vision researchers as moving from low-level to high-level vision. Low-level vision processes the input signal to find features of the signal such as image structure, e.g. intensity edges, or features of the three-dimensional scene, e.g. surface shape. High-level vision is more cognitive in nature: it usually assumes that objects in the scene have been delineated, and that the task at hand is to use world knowledge and reasoning techniques to recognize the objects and to determine the relationships among them. Intermediate-level vision has the role of converting low-level features into objects suitable for higher-level processing. Intermediate vision converts image features into scene entities, signals into symbols.

In the past, machine vision research has concentrated on both low-level and high-level vision. Intermediate-level vision has received less attention, not because it is unimportant but because we needed to understand first what could be extracted from images and what we could do with scene objects if we were able to find them.

At the heart of the intermediate-level vision task is the temporal difference between low and high level vision. The visual signal we receive and its features are transitory in nature. They exist for a short time before they change; new features appear and old disappear. However, the objects of high-level vision have a continuity of existence. They exist when they are not viewed. They exist when there are no features in the current signal to expose them. Because intermediate-level vision must map transitory image features into scene entities that demonstrate this continuity of existence, it is the output of intermediate-level vision, rather than its input, that has temporal consistency. As a consequence, intermediate-level vision because of its very nature must use previous output as input, as well as the features of the signal, if it is to maintain temporal consisteney. It must supplement the results of signal processing with stored knowledge of the world — knowledge of the nature of objects in the world, knowledge of the constraints imposed on objects by physical reality — knowledge that learning may provide in biological systems, but that must be otherwise supplied in less accomplished systems.

An autonomous vehicle explores a world that has persistence. Objects encountered exist when they are no longer in view. Moving about in the world requires knowledge of the environmental continuity. To permit autonomous movement, a vision system must be able to map transitory image features into persistent scene entities, the task that is assigned to intermediate-level vision processes. Because an autonomous vehicle must deal with continuity of existence, we must address the problems of intermediate-level vision in a way that has not been attempted in most previous vision tasks.

The work reported here was supported by the Defense Advanced Research Projects Agency under Contract DACA76-85-C-0004.

In addition to their temporal differences, the various levels of vision processing can be characterized in other ways. Low-level vision processing usually requires a fixed set of input on which to carry out its operations, and makes little use of other information that is available. Because a few image measurements do not characterize scene entities, intermediate-level vision must have available many inputs to mix and match as is necessary. A flexible architecture is required so that the results of other processing are available for input to the intermediate-level vision process. Previous results, such as confirmation of previous hypotheses, are the inputs necessary for mapping signal to symbol.

Low-level vision processing is usually independent of the task at hand. The zero-crossing operator is the same whether we are looking for roads or houses. High-level vision and intermediate-level vision are task dependent. To classify the land cover of the terrain, we must know the task at hand before we can determine the appropriate classification. If we wish to determine whether the ground can support an autonomous vehicle, we do not need to know the grass type. However, if our task is to estimate wheel slippage, then details like grass type are important. Task dependence has always been a feature of high-level vision, but there we have symbols to work with. The utility of intermediate-level vision can be substantially increased if task dependence can be moved below the symbolic level.

Intermediate-level vision must have flexible access to many sources of information if it is to produce results that are reliable and temporally consistent. The careful design of an architecture to supply the various data is a prerequisite to building an autonomous vision-based system. The objects that intermediate-level vision deals with and the results it produces are not the quantitative objects of low-level vision nor the symbolic objects of high-level vision, but rather the qualitative descriptors that interpolate from quantitative signal to symbolic objects. Intermediate-level vision must integrate the top-down approach of high-level vision with its bottom-up, low-level counterpart. High-level models must be rendered and matched against image data as symbolic information is converted to iconic, while attributes of image data must be identified and classified when iconic data are mapped into symbolic. The knowledge system architecture described here seeks to provide a base on which various and varied approaches to machine vision may be explored.

2 Vision System

It is impracticable, with today's technology, to implement the activities of a vision system in a sufficiently complex monolithic algorithm that can cope with the irregularities and imperfections of the outdoor world. For this reason, the overall architecture of our vision system can be viewed as a *community* of interacting *processes*, each of which has its own limited goals and expertise, but all of which cooperate to achieve the higher goals of the system. The various processes may represent sensors, interpreters, controllers, user-interface drivers, or any other information processor that can be imagined. Each process can be both a producer of information and a consumer. Information is shared among processes by allowing them to read data stored by other processes and to update that information. Each process continually and asynchronously updates information based on sensor readings, deductions, renderings, or other interpretations that it makes.

Each process is a *knowledge source* that brings its expertise to the processing of the data that represent the known state of the world. These processes span the range from low-level image processing to symbolic manipulation, and their output will be available for use by all other knowledge sources. Symbolic information may be used to set the parameters in an image-processing procedure, while image properties like texture may be used to confirm a deduced orientation of a supporting surface. The type of information that needs to be shared is enormously varied. The database that stores this information must be able to accept this vast assortment of data types and make it available to requesting processes.

Our system includes a global database through which information is shared. Because all processes share information, the communication bandwidth between this database and the various processes is of concern. If the granularity of the information to be shared is too fine, then the communication channels will be overloaded with an enormous number of transactions, each of which involves small amounts of data, while a granularity that is too coarse requires complex knowledge sources that are beyond our ability to construct. We view the knowledge sources as substantial entities that attempt to share data objects that are composite in nature. For example, we do not expect that an image-processing routine would write intensity-edge information into the database, but rather that it would share conclusions about the three-dimensional objects that are in the world. Of course, these three-dimensional objects will not be identified, nor will they be the final partitioning of the scene into world objects, but they will be entities with which other processes can associate parameters and semantics. This does not mean that the database contains only symbolic objects, but rather that it contains objects that have some semantic character, such as a horizontal planar surface with approximately constant albedo. There are fewer transactions within the system, but each is associated with a significant amount of data.

Some knowledge sources may need to communicate with others at a level that is not provided by the global database. Such communication is private to those sources, and implementation is the responsibilty of the designers of those processes. This level of information sharing often entails a certain computational speed requirement and usually a processing sequence that can be prespecified. Although any system that interacts with a complex world may use this form of close coupling between certain processes, we have tried to focus on the problems of sharing information that is of a higher level and is substantially unstructured.

If processes are to communicate through the database, the language of communication must be rich enough to allow items to be shared. Relevant information extracted from the database is of little use if the receiving process cannot understand it or make use of it. With the diversity of information that is available, we choose to share that informa-

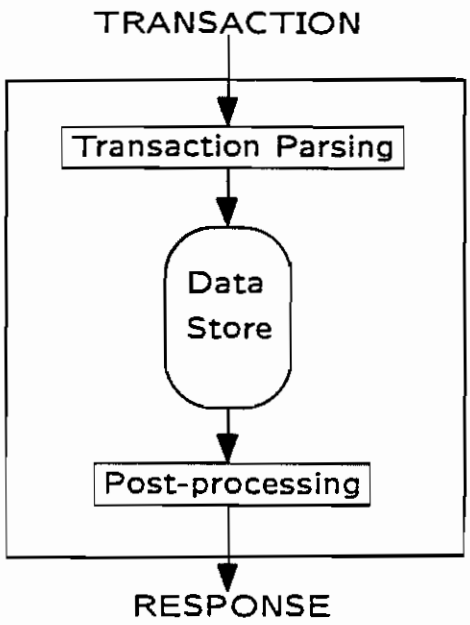


Figure 1: Transaction Processing. The request parser determines what must be retrieved from store, while post-processing of the retrieved items occurs before information is returned.

tion through *semantic labels* that classify the information in the database. These labels must reflect the multiple levels of specificity inherent in the information itself. The labels form a *vocabulary* describing the information that is stored in the database. Accessing information by means of the semantic label allows processes to be independent of the particular data syntax used to store the information. We allow database access through logical combinations of the semantic labels, as well as procedural definitions to be passed to the database so that a user may supplement the vocabulary with additional terms. Passing procedural definitions to the database also reduces the communication bandwidth otherwise needed to return the results. Figure 1 shows this view of transaction processing, in which the database has been passed a request that necessitates the request parser to determine what must be retrieved from store, while post-processing of the retrieved items occurs before information is returned to the requesting process.

A system that views processes as individual experts that may make conflicting interpretations of the data must have a policy to determine what is stored in the database. For example, if two processes determine the height of a particular tree to be substantially different, whose opinion should be stored: the last one given, that of the process with more expertise, or the average of the two? There is no "correct" way to determine a single value. Traditionally, information integration has been accomplished when the data are inserted into the database, and the data that are then stored are expected to be conflict-free. In our system, all processes are considered equal, and only their *opinions* are stored. This approach reflects the view that conclusions are a function of the data used, the knowledge sources that provide that data, and the anticipated use of the conclusion. The user of information should have the opportunity to filter that information with knowledge of both its content and its source. Information in our data store can be modified only by the process that created it, although other processes can cast their opinions. To emphasize the contrast with conventional databases we therefore view our data store not as a database , but as an *opinion base*.

The opinion base stores information in the form of opinions from the system processes about the domain of interest. Another form of information that is critical to the performance of the overall system is the knowledge used by the various knowledge sources. Should that knowledge be stored in a database (possibly the one used to store domain knowledge) or should it be encoded within the knowledge sources? In some processes, particulary low-level image processing, the flow of control is well known, and efficiency issues require that the knowledge be embedded in the procedures; however, other processes, particularly goal-driven ones, gain flexibility of control if the knowledge is separate from the engine that applies that knowledge. Because we expect a variety of processes to be used in intermediate-level vision, we have selected a strategy in which there is no global repository for the knowledge used by the various processes. Each process is free to determine its own knowledge representation. We have a future interest in having processes that modify the ways in which other processes operate, perhaps by generalizing the rules they use. We thus see a distinct advantage in building processes in which the knowledge used by each process is encoded in a form suitable for modification by external processes.

Any system that consists of a collection of independent, asynchronous processes must have a control mechanism that coordinates these processes to achieve the system's goals. In our system, each process is continually active, going about its task of processing the data that define the current state of the world and placing its opinions in the database. When certain combinations of data occur, we must be able to interrupt particular processes and have them deal with this new information. We use a daemon approach to implement this strategy. Daemons are placed in the database by the processes that should be informed when particular events occur, and the processes are responsible for determining how to proceed when they are interrupted by these daemons. Control by means of the database is therefore data driven. Alternatively, any process is free to call procedures that are imbedded within another process, thus allowing control to be passed by procedure call.

Control that is data driven is unlikely to be coordinated to achieve the goals of the system if those goals are not available to the various processes that are performing the data transformations. An important part of sensory integration is planning which activities will contribute to the more general goals of the larger system in which the sensory system is embedded. In our case, we interface with the goals of a *planning system* that controls the activities of an autonomous vehicle. A planning system is viewed simply as another process or set of processes that may access the

database. The list of tasks that the vision system is attemping to achieve serves as data that individual processes must use to prioritize their own activities. Conclusions and data transformations, no matter how correct or clever, are irrelevant if they are unrelated to fulfilling the mission of the highest-level system.

3 Database

The database that we have designed to store the domain data has many of the usual database features. It stores a collection of *data tokens* that contain the domain knowledge and has a set of indexing structures overlaid on these tokens so that data manipulations based on the domain requirements, such as data retrieval, may be implemented efficiently. Unlike many vision-system databases, the database has a continuity of life that exceeds a single execution of the system. In this respect it is much more like a conventional database, whose integrity and usefulness must persist over an extended period. Data acquired during execution of the system becomes knowledge stored in the database for future use. To ensure that the internal integrity of the database is maintained, processes do not have direct access to the data tokens; instead copies of the data are transferred between the database and the process. Clearly, data copying is computationally expensive, which is incompatible with real-time performance. We therefore provide a mechanism in the data access language that allows a process to pass a procedure to the database so that internal processing can be used to minimize the data transferred and the amount of copying that is necessary.

The approach we adopt for controlling integrity is dictated by a development environment in which the system is not built by a single person or group but rather is a set of processes provided by disparate implementers. Protecting the data from being corrupted by an errant process is critical if we want to avoid rolling back the database to a previous version or editing it between actual uses. However the mechanism used to reduce data copying, sometimes at the expense of integrity, is desirable for certain time-critical processes if real-time performance is to be achieved.

Because all processes are considered equal and their opinions are stored, the database will contain conflicting and incompatible views of the state of the world. Some processes may exist solely for the purpose of resolving such data inconsistencies. Of course, even these processes will only be allowed to cast an opinion. User processes may choose to take more notice of the opinions of these conflict resolution processes than of the opinions of processes whose conclusions are drawn from less data. The conflict resolution processes will continually process data in the database (as spare computational resources allow), but they are conservative in nature, preferring not to cast an opinion unless they have overwhelming evidence to support their conclusion. However, a user process may call one of these conflict resolvers to cast an opinion even if it would not have otherwise intervened. Our approach then is to allow inconsistencies to be resolved whenever the data is sufficient to support the resolution, or whenever a user process requires that resolution, i.e. at access time. This approach differs from other approaches that attempt to maintain a consistent data set: in these approaches resolution must occur at insertion time. The approach we adopt is to resolve if necessary, rather than to resolve always. Often a decision-making process can take action without the need to expend resources in resolving data discrepancies. For example, the navigation module of an autonomous land vehicle may be faced with the conflicting data that the object ahead is either a tree or a telephone pole. If the task is to move forward avoiding obstacles, the vision system does not need to resolve whether the object ahead is a tree or telephone pole. The resolution requirement is a function of the task, not simply the data.

A database that stores opinion will rapidly consume storage resources unless a mechanism is provided that will allow data to be deleted or at least archived. A process that is the supplier of data may have little ability to evaluate the usefulness of that data, yet it is the useful data that we would want available in the database. The approach we have adopted is to have processes sponsor data; that is, a process (probably a process that uses a particular data token) will allow that data token to be "charged" against its resource allocation. Many processes can sponsor a single data token, and they are charged proportionately. When a process nears its resource limit (or at any time) it can withdraw its sponsorship of any data that it has sponsored. Data that are unsponsored are available for garbage collection (these data may be archived or deleted). In this manner each process is responsible for deciding what data it finds useful, and this collection of data forms the base of current available information. Clearly, this procedure is not fail safe. Critical data may be removed before their criticality is realized. However, the criticality of data is measured in terms of a process's willingness to pay for it and presumably in terms of the current usefulness of that data.

Although a data token is unsponsored, it will not necessarily be removed immediately. An information producer may not wish to sponsor data for which it has little use, so it may be some time before a sponsor for this information is found. To avoid deleting useful data, the process whose job is to remove data tokens evaluates additional information, such as length of time the token has been in the database, as well as sponsorship information, before it is removed. Data removal is a continuous process, so that the database can be assured of having adequate storage when time-critical tasks demand that computational resources for garbage collection be suspended.

Each process in the system does not have the same resource allocation. At particular times some processes may be more valuable than others. One process has the task of allocating database resources to the other tasks. The allocation is based on the frequency with which data tokens produced by a process are consumed by another process. Such a frequency measure is a moving statistic that allows the allocation to adapt to the current situation. As is usual, data tokens are time stamped to indicate the last time they were modified — that is, the last time a new opinion was added to one of the data slots — and they are time stamped for last use. The time stamps provide data for the resource allocator and the garbage collector.

Data tokens are produced by individual processes and are passed to the database for storage and subsequent retrieval. For the database to access information from within the token, or for a requesting process to be able to extract information from a token, each must either know the form of that information or have some procedure for recovering it. In the design of a system we can choose to use a standard structure for a data token, such as a record structure in which the position of parameter slots are known, or we can use a standard syntax for the token, such as a list of attribute-value pairs, or we can by procedural attachment add functions that retrieve values from the internal data structure of the token.

With standard structures, position, rather than name, gives us access to the data but we require all processes to use some predetermined set of structures. In a system in which different processes do entirely different tasks, it is unlikely that one could find, no matter how clever, a single, or small, set of representations that would be natural representations of the data for all the processes that must have access to that data.

With both fixed syntax and procedural attachment to a data structure, a vocabulary of terms is needed to access the data slots. This is the approach we take. We use a vocabulary of terms that spans the entities and relationships of interest in the application domain. For an autonomous land vehicle, the vocabulary consists of words or labels that describe the outdoor environment, e.g. tree and height, so a process could ask a data token that represented a tree for that tree's height. The actual structure used to hold the data can be invisible to the user who gains access to the information through the labels. The labels must to be known by all processes that wish to access this information in the database. This semantic level does seem to be the appropriate level on which to share information.

Should we use a fixed syntax like attribute-value pairs to hold the information in the database and provide a simple routine to retrieve the value given the attribute, or should we use the more complex approach of attaching to a data structure a set of functions that can retrieve the value of a data slot given the slot name? We take the latter approach to increase the functionality that is available when we retrieve a value based on slot name. From the point of view of systems building, in which parts of the system are built by independent groups, this approach places the decisions for the form of the data structure and the accessing functionality within one group and provides a clean interface with the database. Each process can now select its own internal representations for the data it produces, and that data can be shared through access functions that are based on terms or labels in the vocabulary which describes the underlying domain. A common vocabulary requires that each process know how to translate from its internal representation to information in vocabulary form. This avoids the need for each process to know how to translate into the individual representations used by other processes. Additionally, new processes can be added to the system without retrofitting the new representations to the older processes.

A collection of data tokens is not a database unless there is a means of accessing the information in the collection in a manner that does not require a search through the entire set. A set of indexing structures that allows access in a more direct manner must be based on the subsets of the data that need to be retrieved. These structures are therefore based on the domain requirements and relate to the semantics of the actual data stored. Our architecture for sensory integration is implemented in the task domain of an autonomous land vehicle navigation. The indexing structures that we use are associated with the need to retrieve information that is appropriately grouped for the task of navigation in the three-dimensional world. A *spatial directory* that forms subsets of the data based on spatial location, and a *semantic directory* that forms subsets of the data based on object class are the principle indexing schemes that we use to organize storage and retrieval of data tokens. Figure 2 gives an overview of transaction processing by means of directories in the database designed to support autonomous vehicle navigation.

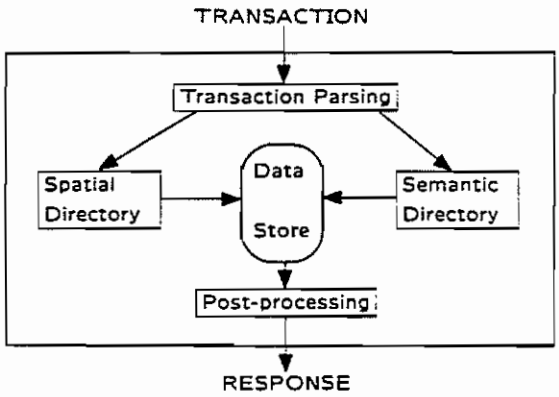


Figure 2: Transaction Processing — Autonomous Land Vehicle Database. Access to the data store is by means of a spatial directory, a semantic directory, or both.

4 Spatial Directory

The spatial directory organizes the data tokens into groups determined by spatial location. Because an autonomous vehicle may roam about in an extensive environment, we need a representation of that environment that can deal with its spatial extent. In addition, the representation must be efficient in indexing data when the data are distributed nonuniformly over the environment. Data will need to be accessed at various levels of resolution depending on the task that is being addressed. Route planning needs lower-resolution data than does, for example, landmark identification or obstacle avoidance. Particular data may need to be stored at multiple levels of resolution to match the requirements of different tasks. The world is three-dimensional but the vehicle is restricted to a two-dimensional surface embedded in this world. Although there are many reasons for choosing a two-dimensional index, such as latitude and longitude, and then representing the third dimension as a data value, we

chose to use a three-dimensional index. Our selection was motivated by the advantage such an index gives in encoding spatial relations within the directory, in generating visibility information, and in using this architecture in other spatial domains in which movement is not restricted to a two-dimensional surface.

The three-dimensional index selects a volume in space that we represent as *voxels* [4]. The largest voxel is the world, which is subdivided into smaller volumes as we need to represent spatial position with higher precision. The index granularity is fine enough to be able to position an object in a volume that is precise enough for the application. Recall that this index is an index into a directory; in the directory cell are pointers to the data tokens associated with the volume of space represented by this index. Data tokens need not be placed in the directory at the finest index available but only at the precision with which their spatial location is known. A tree whose position is unknown would be placed in the largest voxel; this voxel represents the entire world.

The voxel-based directory not only gives a range of position resolutions, it also allows different parts of the world to use different resolutions for storing data. Parts of the world that have little data associated with them may choose to place all the pointers to data tokens representing objects in this area in coarse-grained volumes, while the part of the world in which the vehicle is active can be subdivided into finely partitioned volumes. We not only have multiple resolution, but we also can select resolution relevant to the area concerned.

In selecting a voxel-based representation of space, we have the option of dividing that space into regular voxels in which all voxels, at a given level of subdivision of the space, are of equal size, or we can choose to divide the space into irregularly sized chunks. Irregularly sized voxels have some attractions, as they allow irregularly shaped objects to be confined, and hence indexed, within a volume that matches them. Regularly sized voxels often are unnecessarily large when they are large enough to contain a irregularly shaped object. However, if we use irregularly sized voxels we may need multiple indices to allow for overlapping voxels that are indexing different irregularly sized objects in the same volume of space. Multiple indices increase the computational load, and we are, after all, trying to index data tokens in an efficient manner. We therefore use a regular subdivision of space in which each voxel is subdivided into eight equally sized and shaped smaller voxels.

In making this choice we must address the problem of indexing objects whose shape does not match this partitioning of space. Generally, it is easy to place stationary compact objects within a voxel that can completely contain them, but objects like linear structures, surfaces, and moving objects require alternative approaches. Linear structures like roads, rivers, telephone wires, and fences are stored as one data token, but pointers are placed in all the voxels through which the structure passes. We use the smallest-sized voxels that are appropriate; for example, the voxel size for a road will be determined by the road width so that we can be assured that the road "fits" within the voxel.

The same approach is taken with other extended objects, such as surfaces: a single data token has pointers to it from the set of voxels through which the surface passes. The size of the voxel is selected by the process inserting the surface into the database, based on such factors as accuracy of the surface shape, and extent. Recall that this placement in space is to aid retrieval, not to specify exactly where things are. Detailed location information in available from within the data token. There is no need to place objects in the spatial directory in the smallest voxel that might be possible.

Moving objects are usually compact objects so they present little problem in placement at their current position, but there may be times when we want their track represented in the directory. We use the same approach we used for linear structures and extended objects: we represent the moving objects with a single data token and point to the token from voxels associated with its track.

An advantage of a multiresolution spatial directory is the ease with which we can represent approximate location. We place an object in a voxel that is large enough to contain the limits of its possible locations. Object location may be approximate because of image processing errors when detecting objects in imagery, or because we do not know our exact position when we make an observation. The latter is particularly relevant in the case of an autonomous vehicle. Data can be added to the database before its position is known, and then, when better location information is known, the directory can be updated by moving the data to a smaller volume. If this is not done the data will be retrieved and examined when requests are processed for data from the original larger voxel. A background process whose task is to move objects to their most precise location within the directory (when processing resources are available) accomplishes the directory update and thereby achieves retrieval efficiency. Hence all data can be directly inserted into one directory whether their location is known accurately or only approximately.

Having all data, whether its position is known or uncertain, within one directory structure allows us to respond easily to data retrieval requests that want "all objects that are within a certain volume in space" as well as "all objects that are possibly within that particular volume of space." Clearly, in the task domain of an autonomous land vehicle, knowing what *might be* ahead and what *is* ahead is necessary for competent navigation and obstacle avoidance. Within the voxel structure, "within a volume" maps to the tree of voxels below (finer than) the voxel containing the volume while "possibly within a volume" maps to the tree above (coarser than) the voxel containing the volume. When data can be retrieved on the basis of their location, then retrievals on the basis of spatial relations are also possible.

The spatial directory encodes the spatial relationships between items stored in the database. As objects are moved or their spatial positions refined, these spatial relations are maintained without additional processing resources. New objects entered into the database encode their spatial relationship with previously entered data. In our task domain, we expect to retrieve items based on relative position — objects to the right of the road, trees casting shadows on the road, and so on. Having an indexing structure that matches the world structure allows this without the overhead that would be presented by alternative schemes, such as a rela-

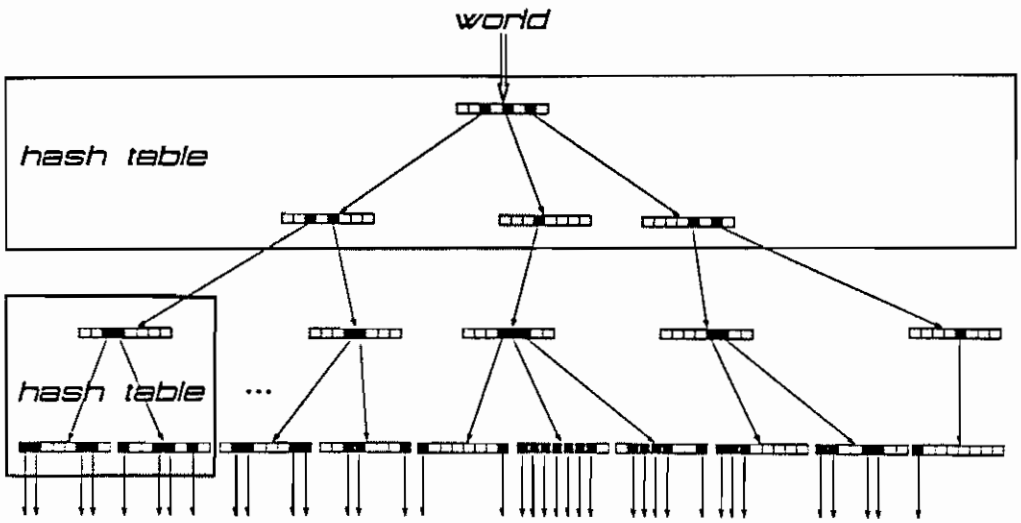


Figure 3: Octree Representation of the Voxel Description of Space. Hash tables are used to implement the octree; more than one level of the octree is stored in a single hash table.

tional database.

The reduction of computational resources used to maintain the database was also instrumental in our treatment of time. The database is always assumed to represent the world at current time. If historical information is to be stored, then it must be time-stamped, otherwise it is implied that the data reflect the state of the world as it currently is. We adopted this approach so that we could avoid elements of the traditional frame problem [1]: if time is a parameter of the data token, then this token has to be updated even when the real data has not changed but time has passed. We take the usual approach adopted in conventional databases, in that information is assumed to be still true if it has not been altered or specifically marked as applying only to some particular interval of time.

Voxels are the representation of the world used in the spatial directory, but there is the independent issue of how we represent voxels in our implementation of the spatial directory. We use a "pointerless" *octree* [3] that itself is implemented by multiple hash tables. The use of an octree to implement a voxel representation is natural; our selection of the pointerless approach was based on the expectation that many voxels will contain no data, and many voxels will not be subdivided into smaller units. Hence the more usual approach of using cells with explicit pointers to the finer cells will produce many cells containing mainly null pointers. With the pointerless approach, only voxels that contain data tokens are allocated any storage, and null pointers are not used. Figure 3 shows an abstract view (using null pointers) of the way we use an octree to represent the voxel description of the world. The actual implementation uses hash tables to store the links between voxels. The number of levels of hash tables is in fact somewhat less than the number of octree levels, because several octree levels are stored in a single hash table, as shown in Figure 3.

5 Semantic Directory

The spatial directory provides an indexing scheme that matches the spatial nature of the data in the task domain; the semantic directory provides an indexing scheme that matches the semantic nature of the data in that domain. As previously mentioned, we use a vocabulary of terms to facilitate communication between processes. The semantic directory specifies these terms and defines the set of connections between them. The vocabulary provides a set of labels that is used to describe the data tokens in the database. Such a set is dependent on the task domain, and for autonomous land vehicles we use terms that label objects in the outdoor environment, such as tree, road, rock, meadow, or ditch, as well as terms with less specificity, such as immovable_object, obstacle, or object.

The need for terms that define the semantics of things in the world at various levels of abstraction or multiple levels of resolution is apparent if we wish to interpret imagery as seen from a moving vehicle: objects usually appear first at a distance, at poor resolution, and gradually change form as we approach them. The levels of abstraction that we need are a function of the processes we have and their ability to instantiate the terms. There is no point in being able to describe leaves on a tree if the sensors are incapable of resolving objects that small. Equally there is no point in describing trees as belonging to the superset "wooden objects" if no process makes use of that set. The vocabulary choice that we have made is based on our assessment of the competence of low-level image processing routines and the

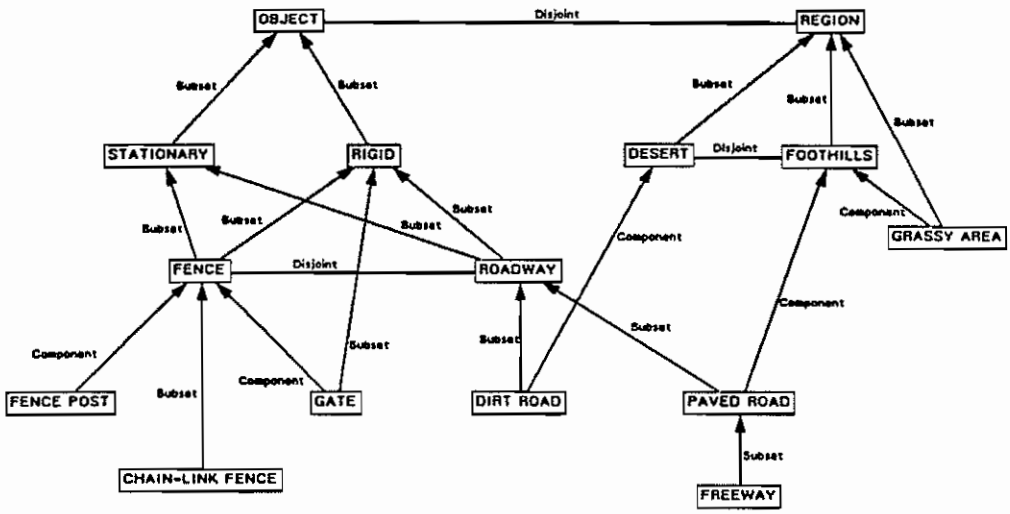
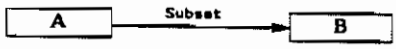


Figure 4: Semantic Directory. Implemented as a semantic network, it gives access to the database data tokens by means of their semantic type.

requirements of higher-level processes. The choice is critical to sensory integration, for within the vocabulary we are restricting the means of integration, the information that higher-level processes can transfer to the low-level routines (and vice versa), and the functionality requirements of both higher and lower-level processes. In absolute terms, successful sensor integration demands selection of an appropriate vocabulary.

Any vocabulary whose constituent terms span a wide range of specificity in a domain must include terms that are related to one another. The second component of the semantic directory, a semantic network [1], defines these connections. The network itself has two parts: one which defines the specialization of terms by a graph, that is, a lattice that specifies *subset/superset* relations and that is augmented by the inclusion of the *disjoint set* relation, and a second part that describes the decomposition of composite objects into parts. While the first part indicates relationships that must hold, such as "a pine tree is a tree," the second decomposes composite objects into parts that are usually present, such as "fire engines usually have ladders." The first part of the network is used for inference; for example, in inferring that a pine_tree is a tree, which is an immovable object, which is an object, and so on. The second part gives default values that may be used to trigger some process to find them, or may be used by an evidential reasoning process that is attempting, say, to classify an object based on what has been detected and what one might expect to see when viewing that particular object. For example, when a process is attempting to decide whether an object that is composed of several vertical rectangular objects and some horizontal lines could be a portion of a fence, knowledge of the expected parts of a fence is crucial to that determination. Additionally, the network provides a means for inheriting properties from a more general class; e.g. if a tree is usually composed of branches, leaves, and a trunk, then a subclass, like pine_trees, will inherit this parts decomposition as its default description. The approach taken reflects the need, on one hand, for the system to reason about objects, while, on the other hand, the system must be able to recognize composite objects on the basis of their likely parts. A mechanism for logical inference and a mechanism for object decomposition that is usual but not unequivocal must therefore be provided.

The semantic network we use is implemented as a graph in which the nodes represent the vocabulary items and the labeled arcs represent the relationships among terms. Both the subset/superset and disjoint set relations, together with the composite object decomposition, are combined on the one graph using various labels on the arcs to distinguish between them. For example, the lattice fragment



encodes the sentence $\forall x : (A(x) \Rightarrow B(x))$, while

encodes $\neg\exists x : (C(x) \wedge D(x))$. This network representation allows selected inferences to be made rapidly through graph operations. The particular implementation allows display of the semantic network in its entirety or of selected clusters of related information. Figure 4 shows a small part of our semantic network. The graphical display of the network is the interface we use to build the semantic directory and to add new words and relations to our vocabulary.

Each node of the semantic network is associated with a

vocahulary term, and to it we attach pointers to all the data tokens in the database that have been labeled with this term. The nodes of the semantic network can be accessed by the vocabulary label, and thus provide a directory to data tokens on the basis of the semantic label.

Although we view the semantic directory as a graph structure and display the semantic network as a graph, the implementation uses hash tables for speed of access. When data tokens are added to the database or when additional labels are added to a token's description, the semantic directory is updated appropriately.

Data tokens are attached to the most specific network nodes possible. If, for example, a data token had been labeled by a process as heing a paved_road, then it is attached only to the semantic network node for paved_road even though all paved_roads are known to be roadways. This approach was adopted to save storage as well as to provide a straight-forward implementation of the retrieval request to return all objects that are paved_roads as opposed to all objects that might be paved_roads. The second descriptor includes objects in the more general class "roadways" as well as those labeled "paved_roads." Paved_roads are found attached to the nodes of the lattice that form the tree rooted at the node labeled paved_road, whereas roadways that *might be* paved_roads are found attached to the nodes of the network tree *above* the node labeled paved_road. This arrangement parallels the mechanisms used in the spatial directory to find objects that are at a particular location, as opposed to those that might be at that location. It is the responsibility of the access routines to retrieve the appropriate items from the database by means of the semantic network.

The semantic network serves partly as a definition of the meaning of concepts. If a process designer wishes to know what questions he can ask of a data token that is, for example, a tree, the network specifies the relevant terms, such as height, or color. The semantic network defines more than just the communication language between processes; it defines something of the domain concepts that all processes must use. However, while the concept "tree," for example, may be seen in the semantic network to include pine_trees, and oak_trees, and so on, and while a tree is an immovable_object and an object, and while it has parts (and properties) of height, and color, it is not "defined" hy the network. The network does not define for a process the concept "tree;" it specifies only the concepts that processes can use to communicate about a tree. A particular process may determine that an object is a pine_tree on the basis of its temperature and the soil type around it, but it must share its information in terms of the concepts defined in the network. This approacb was adopted for important pragmatic reasons — it is impossible to "define" a concept like a tree; yet we need to communicate information about a tree in terms that other processes understand.

6 Other Directories

The system architecture we have described is independent of the indexing structures that are overlaid on the database; to change those structures requires only changes to the parser that processes database requests (as can he seen in Figure 2). The extensihility of the directory system allows future requirements to he accommodated without change to the overall system structure. The two directories we describe were devised to allow an autonomous land vehicle to navigate through a world in which most objects are static and motion comes primarily from the movement of the vehicle itself. In other scenerios, this will he inadequate. In environments in which there are many moving objects, and fast-moving objects that are likely to impact the mission results, other directories that index the database through additional parameters, such as those associated with movement, are vital. The architecture described has the flexibility to accommodate such extensions.

7 Process Control

We have described the various processes that form the system as independent, asynchronous processes that can be activated he means of daemons imbedded in the database or by more conventional procedure calls. Each method uses vocabulary terms to interact with the database. Each process is continuously executing, although a process may put itself to sleep only to be awakened when predetermined data conditions exist. Who determines these conditions? Should every process he permitted to determine the conditions needed to interrupt another process? Some processes may be time critical and prefer not to be interrupted. Our approach is to require that the process itself set these conditions within the database. Any process can attach one of its daemons to any data slot of any data token, so that the process will be interrupted whenever any new or changed opinion modifies that data slot. We selected data slots rather than data tokens as the items on which to attacb daemons because data tokens usually represent a complex item and any one process is probably interested in only some aspects of it: for example, the navigational module of an autonomous vehicle will want to be interrupted if a sensor process gives a new opinion on the position of an obstacle, but it is unlikely to need to be interrupted if the obstacle's color changes. It is, therefore, the responsibility of a process to determine when it is to be interrupted.

In a like manner, it is the process that determines what action to take when it is interrupted. The interrupt handler is part of the definition of each process. As processes are quite varied, there is no sense to the notion of a generic interrupt handler. Clearly, processes may choose to continue with what they are doing rather than to process the interrupt if they assess the current task to be more relevent to mission success than that associated witb the interrupt. Conversely, a process may instead suspend or abandon what it is doing in favor of the interrupt. The overall system concept is that of a loosely coupled system in which all processes work on their goals cognizant of the overall goals of the mission. Each process determines bow it can best support the mission goals and is responsible for the means to achieve this.

The process architecture we use parallels that of blackboard systems that were brought to prominence in the build-

ing of speech understanding systems [2]. In these systems data were placed on a blackboard; if the combination of data on the blackboard met the preconditions for a particular procedure to execute, then that procedure was triggered and put on the schedule for computing resources. In an important way, the approach of activating processes using daemons differs from the triggering mechanism used on blackboard systems. We do not have a pattern matcher whose job is to trigger processes when a particular pattern of data appears in the database (or on the blackboard). For efficiency reasons, the patterns that pattern matchers are to recognize must be predetermined and compiled in at system building time. In a system that is loosely coupled, in which different processes may be present during different executions of the system — in which the system must function even if some of the processes (or hardware) fail — an approach to pattern matching that decentralizes the responsibility for determining whether a process should be triggered seems more manageable. We have chosen to trigger on an opinion being changed rather than on a particular pattern in the data itself. In selecting this mechanism, we weighed the cost of the additional processing that is done by the interrupt handler in each process against the computational cost of running a generalized pattern matcher.

In any system that is a collection of processes, priority will sometimes need to be given to processes that perform time-critical tasks. At other times, the system could be underutilized. As a result some processes should be scheduled as foreground jobs, which compete for resources when they request them, while others should be background processes using only spare resources. We identified some of the background processes: the module that resolves data inconsistencies, the one that recovers storage space, and parts of the resource allocator itself. The system should never be idle. We allocate computational resources to modules via a separate process, a metalevel process, that changes the time slice allocated to various processes. A process that produces data, including opinions, that are used by other processes gets more resources than a producer of unused data. In addition, a process can request more resources if it determines such a need, so that critical processes can ask for priority.

Our current system implementation is one in which all the various processes execute on one computer system and all interact through a common virtual address space. This approach was adopted to eliminate the system building necessary to run experiments on multiple processors. However, the design of the system assumes a virtual environment in which there are many processors running in parallel, with a communications network between them. This accounts for the design decision of the rather loose coupling between processes. On a network of parallel processors, we would expect some processors to be dedicated to particular processes whose computational task is matched to the particular machine hardware. Other processes would be allocated among the available processors. Although we are aware of the bottleneck that might be caused by centralizing the database we envisage a system in which the process accepting requests for database transactions will be centralized but the database itself and the procedures that carry out the internal processing may be split across processors.

8 Tasks

The information system that we have described presupposes that the job of interpreting sensor data can be subdivided into pieces, and that it is the combination of the processing provided by these pieces (using stored knowledge) that achieves sensor interpretation. The goal of sensor interpretation is clear: we need to build a model of the world that is being sensed, but what the individual tasks are, and how stored knowledge is used is not obvious.

In the vision literature we see a wide range of experiments that have probed for an answer to the first of these, identifying the tasks. However, this question has usually been posed in a context in which little data were available, save the sensory signal. In a system like the one we describe, knowledge is vitally important; we might therefore expect the division to be into tasks that are somewhat different to those used when the signal is the only data. To be more specific let us look at an example of how the availability of knowledge may influence sensory processing on an autonomous vehicle.

Given that our database includes a general description of the terrain and the major objects in the area, we can use the generic iconic models of the objects to construct the image that the sensor expects to see. One task that must be performed is to confirm that the expected objects are present. This is model-based vision, but it differs from the conventional model-based approach in that the models we have are only generic.

The procedure for verifying models must be able to match properties and parameters of this model to the data, rather than use template-based matching of intensity images. Of course we will not be able to generate all the objects in the scene; some, such as unexpected objects, will not be in the initial database. Here a bottom-up approach is needed. However, as we verify the existence of some objects, tasks like image segmentation will be focussed on those areas that still must be explained.

Even the approach taken to a low-level procedure like segmentation is altered by the availability of terrain data. That data can be used to break the interdependence of surface slope and surface albedo in determining image intensity, and hence can allow segmentation of the intensity image to be replaced by segmentation of the albedo image. In addition, textures seen in previous images can directly influence the manner in which segmentation proceeds in the new image. Stored knowledge, in this case terrain data, can even be used to correct the textures for perspective distortion. The results obtained by algorithms for low-level processes, like segmentation, are now as much a function of the database knowledge as the signal.

The correct decomposition of the job of interpreting sensory data into tasks is an open research issue, but one that will be greatly influenced by the availability of stored knowledge. In a similar fashion, higher-level reasoning tasks are influenced by the increasing competence of low-level processes to provide information in summary form. The integration of higher-level reasoning and low-level signal processing should not be achieved by resorting to mechanisms in which higher-level processes reason about image features; high-level processes should use the qualitative descriptors

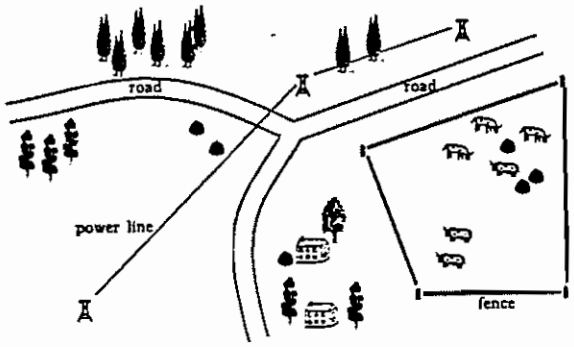


Figure 5: Database Display. Generic models are used to display the current contents of the database. Sketch maps designed with the same tool are used to put information into the database.

that intermediate-level vision can assemble from low-level signal processing and stored knowledge.

9 Experimental Environment

Any project that involves substantial software development makes use of available software tools and builds others where they do not already exist (or are unattainable). We have mentioned using a graph manipulation package to build the semantic network, and various processes have made use of existing image-processing, graphics, and three-dimensional modeling packages. Our system is built in Lisp and makes extensive use of the flexibility a Lisp environment can provide. The system described runs on the Symbolics 3600 family of Lisp machines. All system-building tools can be executed as independent processes that run simultaneously with the processes that manipulate data and carry out reasoning activities. They provide an interactive environment in which to experiment. An example of the flexibility such an environment can provide is shown in Figure 5, which displays a portion of the database. Because we must be able to determine the current state of the database as processing proceeds and because most of the data contained in it describe spatial information, we choose to display it pictorially. As some data tokens may only be labeled with general terms, such as immovable_object, we display generic iconic models of the data tokens. Entering spatial data into the database, particularly more qualitative data like a sketch map, is made easy with such a tool. We create a display using generic models and place the sketch-map data into the database.

Tools that lessen the effort needed to build systems make it possible to embark on the experiments that require a complete system to be in place before the simplest trial can commence. Only through experimentation with real data on a running system can competence be fairly evaluated.

10 Summary

The natural, outdoor environment in which an autonomous land vehicle operates imposes substantial obstacles to the integration of the vehicle's various sensory, planning, navigational, and control activities. The complexity of the domain and the requirement for high reliability rule out approaches that do not make substantial use of stored knowledge about the environment. An intelligent database that competently contributes to the processes that perform these various activities is central to the overall design of an autonomous system.

Today's technology is not capable of directly integrating sensory information with stored knowledge in one step. To cope with the irregularities and imperfections of the outdoor world, a series of interactions is needed to reach tentative conclusions that constrain the final outcome. For this reason, our software architecture for sensory integration is a community of interacting processes, each of which has its own limited goals and expertise, but all of which cooperate to achieve the higher goals of the system.

Processes must be able to take advantage of relevant knowledge that may be available. The design of a knowledge system must include a means for effectively communicating semantic information to the multiple and varied processes that wish to consider it. A vocabulary of terms and a set of connections among them serve this purpose in our system. The vocabulary consists of a domain-specific set of terms that have been identified as being both useful for, and instantiable by, the computational processes. A semantic network is used to encode the specialization lattice of the concepts and the physical decomposition of composite objects.

The database architecture for an autonomous system must allow multiple representations of world data. It must support quantitative and qualitative, inconsistent and approximate data at multiple levels of resolution. Our architecture is based on spatial and semantic directories that organizes the various representations of the knowledge to allow for focussed processing, for flexibility of access, for modularity of task processing, and for asynchronous process control. The directories both link information stored in the different representations and encode the relationships among objects. They permit the achievement of partial data consistency, as required for the task at hand, rather than complete consistency of all data, relevant or not. In a departure from the traditional strategy of resolving all conflicts at the time of insertion, our knowledge system defers this chore until the data are required — when the relation of the information to a task is known, and when more data are likely to be available.

The autonomous system that we describe consists of a community of interacting processes that attempt to cooperate in achieving the goals of the system. The database is an active participant in the system, not merely a data store, merging the functional aspects of process control and data organization.

References

[1] Barr, Avron and Edward A. Feigenbaum, *The Handbook of Artificial Intelligence*, William Kaufmann, Inc., Los Altos, California, 1981.

[2] Erman, Lee D., Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy, The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, *Computing Surveys*, Vol. 12, pp 213-253, June 1980.

[3] Samet, Hanan, The Quadtree and Related Hierarchical Data Structures, *Computing Surveys*, Vol. 16, pp 187-260, June 1984.

[4] Sribari,
Sargur N., Representation of Three-Dimensional Digital Images, *Computing Surveys*, Vol. 13, pp 399-424, Dec 1981.