Picture Interpretation:

A Symbolic Approach

Sandy Dance Terry Caelli Zhi-Qiang Liu

1995

Preface

Traditional methods for image scene interpretation and understanding are based mainly on such single-threaded procedural paradigms as hypothesize-and-test or syntactic parsing. As a result, these systems are unable to carry out tasks that require concurrent hypothesis testing.

In this book we explore a method for symbolically interpreting images based upon a parallel implementation of a network-of-frames suggested, for example, by Minsky (1975), to describe intelligent processing. The system has been implemented in an object-oriented environment in the logic programming language Parlog++ and includes the propagation of uncertainty through each frame (object) using Baldwin's (1986) formulation. The system is tested with several scenarios of increasing complexity, culminating with the legal interpretation of traffic intersection images.

The authors would like to thank Andrew Davison for much help with the implementation language of this project, and for providing exceptional advice on ideas, papers and this book. We must also acknowledge the Computer Vision and Machine Intelligence Laboratory at Melbourne University as a whole for providing a stimulating and lighthearted atmosphere for intellectual jousting. The Computer Science Department must be recognized for providing a supportive role in the work involved. We would like to acknowledge the Australian Computer Graphics Centre of the Royal Melbourne Institute of Technology, and in particular Gordon Lescinsky and Mike Gigante, for their help in obtaining traffic video and digitizing it. Finally, the first author would

like to thank his wife Jill and children Felix and Rupert for their tolerance of this folly.

This work has been supported by a number of funding sources and employers. The Defense Science and Technology Organization supported the early stages of this work. Funding has been received from the Centre for Intelligent Decision Systems, and from the Computer Science Departments at the University of Melbourne and Curtin University. Finally we would like to acknowledge the support of the James Herbert Miriam Bursary through the Faculty of Engineering at the University of Melbourne.

S.D.

T.M.C.

Z.Q.L.

Melbourne, April, 1995.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This book concerns high-level vision – the interpretation of images in the light of domain knowledge. In this realm, it is not sufficient to simply label regions or objects in the image. Rather, it is necessary to "tell the story" **behind** the image, ie, to interpret the "intentionality" of the image objects. This book explores a specific architecture for this problem: an interacting network of agents or frames, each asynchronously working on their own aspect of the interpretation. The architecture has been named SOO-PIN: symbolic object-oriented picture interpretation network. It is based upon recent developments in:

- Cognitive science,
- Object-oriented software engineering,
- Connectionism in Artificial Intelligence.

The approach taken in this study has been to investigate the architecture in a series of domains of increasing complexity. Initially a very simple domain was chosen in order to design the basic modules and communication protocols. Once that was successful, the system was adapted and expanded to determine scaling of the concepts involved. Success was measured in both correctness of interpretation for a given scene, and ease of adaptability of the system to

changing domain knowledge. These studies culminated in the analysis of a complex traffic scene analysis system.

In the course of the study a number of specific ideas were developed:

- The use of an object-oriented concurrent logic programming language for scene analysis [19].
- The use of the above architecture for traffic scene analysis [18].
- The modification of Baldwin's support logic programming for use in scene analysis [20].
- Distinguishing symbolic and spatial data, and recognizing the need to ground some symbolic predicates in the data. This reflects, in some ways, the distinction between internal and external representations as discussed by Slezak [63].
- For the interpretation of traffic, a velocity detection mechanism was developed which is founded on token matching but using a combination of symbolic and spatial predicate matching to derive the best trajectory for the vehicles [20].

Chapter 2 is a description of a number of earlier high-level vision systems which are essentially single-threaded approaches using either some form of syntactic parsing (ie, using transition networks) or hypothesize and test procedures. Multi-threaded systems are then discussed ending with some similar approaches to that described here, together with their drawbacks – which have motivated the approach adopted in this book.

Chapter 3 deals with the foundations of the approach taken here, drawing on the work of Herskovits, Lakoff, Minsky, Schank and Hewitt. The chapter ends with a review of logic programming and object-orientation, describing the work of Shapiro and Takeuchi which unites these two paradigms, and culminating in the choice of an implementation language, Parlog++.

Chapter 4 describes in detail the investigation and implementation of the system, its data structures, how the modules (here called "concept-frames"

to distinguish them from frames, classes and objects as used in the literature) were implemented in Parlog++, and how they were networked together into a message passing system.

Chapter 5 describes the first two simple domains used for testing the system, the "cutlery" and the "wheels" scenario. **Chapter 6** then goes onto describe the initial "traffic" scenario, which uses video images of intersections as input. The hierarchy of "concept-frames" used to deal with this domain is described, together with the necessary low-level processing.

Chapter 7 explores the incorporation of "belief" or "uncertainty" into the network – from a vision perspective. Specifically, Baldwin's approach has been adopted (with reservations and extensions). The chapter finishes with a few trial runs of the system utilizing the derived uncertainty measure, together with analysis and summary.

Chapter 8 describes how the traffic scenario was enhanced with the calculation of car velocities. It first reviews some of the literature on velocity determination and the correspondence problem. It goes on to describe the technique used in SOO-PIN using the "best match between 3 frames" technique, and its implementation.

Chapter 9 is the culmination of the study, containing a series of analyzed traffic scenes, each one discussed at length, highlighting the systems successes and failures.

Chapter 10, the conclusion, points out the generality of the system, and its limitations, together with desirable extensions – the directions research should continue in the future.

Chapter 2

A Recent History of Image Interpretation

Computational vision has traditionally focused on problems of low-level sensing, pattern and object recognition, and relatively little attention has been paid to making sense of an image once primitive structures have been identified. This entails fitting the image into a broader context, or equivalently, bringing to bear domain knowledge in order to produce an interpretation. Such domain knowledge is typically symbolic and so the problem involves interpreting numerical data with more logical or syntactic information. The problem, then, reduces to that of finding the best architecture for integrating this domain knowledge with the labeled image data. Previous approaches have been based on the single-threaded graph processing programs, using hypothesize-and-test or syntactic parsing. Some of these "picture language" systems reported in the literature are described below. More recently there has been a move to implement systems based on the object-oriented paradigm, and these are described later in this chapter.

2.1 Graph Based Systems

These systems use some form of graph parsing or traversing algorithm to process the image. This can take place in top-down or bottom-up directions.

They can also be characterized as single-threaded or multi-threaded. Single-threaded systems, as used here, are those in which at any one moment there is at most a single hypothesis under consideration, whereas multi-threaded systems entertain more than one hypothesis simultaneously. This single- vs. multi-threaded distinction is not the same as the sequential vs. concurrent architecture distinction introduced later.

2.1.1 Augmented Transition Networks

The first technique considered is the augmented transition network (ATN) [14, p199]. This technique was developed for parsing natural language, but has been adapted for vision. It is based on the notion of states (the graph nodes) and the paths for moving between states (the arcs between nodes). The traversal through the graph is determined by the sequence of tokens (language elements or image segments) on the input. It is a top-down technique because it starts with the graph describing a full sentence or pattern, and invokes subgraphs to parse nonterminal symbols as required. The algorithm is augmented with stored data from past states which is used to influence decisions.

Tropf and Walters [64] describe a vision system which is one of the earliest to employ an ATN. It is used to control an analysis-by-synthesis approach (hypothesis-and-test), with the augmentations storing the model-to-pattern associations, ie, the 3D space constraints upon the object. The ATN describes the structure of 3D objects, which the system locates in the input image.

Bajcsy et al[5] describe a system that takes 3D polyhedral shapes derived from aerial photographs of urban scenes and uses an ATN to perform high level scene analysis, guided by user queries. This system approaches the vision problem as analogous to language, and treats the world knowledge as a grammar, and the segmented faces and the relations between them as a dictionary.

2.1.2 Semantic Networks

The semantic network (or associative network) [14] is a way of representing a collection of concepts (nodes) and the relationships between them (the arcs). In fact, the predicate calculus is an equivalent representation, but procedurally, the semantic network differs from predicate calculus in that it indexes the relationships by their endnodes. This optimizes traversal of the network from concepts through relationships to other concepts.

Nieman et al[51] describe a system that uses a semantic net to model relations between concepts in image sequences of the human heart, and use production rules to draw inferences (diagnostic descriptions). The system creates a "search tree" of competing instances (multiple hypotheses) at each node of the network, and uses an A* search for overall control.

Another medical system is that of Dellepiane et al[23] who use a semantic network to model the relations between parts of 2D and 3D models of brain tomographic imagery. They use the hypothesize-and-test paradigm constrained by network relations, and common data is stored in a blackboard. The output is labeled regions.

Govindaraju et al[33] use a top down approach based on a semantic network to handle one aspect of their multi-modal system for understanding newspaper images. This aspect relates 2D newspaper photographs to their captions, and, for instance, finds named people in the image. They also use an augmented transition network (ATN) to parse caption text, generating part of the semantic network used in processing the image. The rest of the network derives from domain knowledge.

2.1.3 Constraint propagation networks

These are networks in which nodes represent propositions, and arcs represent dependencies, or justification links, between them. Constraint propagation networks are logically equivalent to the propositional calculus (as opposed to predicate calculus) as there is no way to represent quantified expressions [68].

Mulder et al[49] describe a series of refinements of the MAPSEE sys-

tem that, given a 2D geographic sketch map constrained by syntactic rules and represented by plotter commands, uses local constraint propagation over a graph to handle labeling hypotheses. Reiter and Mackworth[55] use the MAPSEE system as a basis for analyzing vision problems in terms of logical frameworks. From this perspective, vision problems reduce to theorem proving on a first-order logic database consisting of prior knowledge, image domain knowledge, constraints derived from contingent knowledge about the scene, and, finally, queries about the scene. One limitation of this perspective is that many spatial predicates can not be computed symbolically, and so the interpretation problem can not be exclusively cast as a logic program. This is further discussed in Section 4.2.

Boddington et al[10] also use constraint propagation, in this case an assumption-based truth maintenance system (ATMS), to maintain partial results of competing interpretations. They develop a system, CARRS, and show that the system is able to locate cars in some natural scenes. Provan, however, has shown that such ATMSs generally have exponential complexity [54]. Nevertheless the system is feasible in this case because of judicious early pruning of hypotheses.

2.1.4 Discussion

Representing problems in terms of graphs, while logically equivalent to predicate calculus, is a good way of representing problems for humans, in so far as the logical links between parts of the problem become clear. Moreover, the graphical representation suggests ways of solving problems which can, in some cases, be more efficient than logic-based approaches (these are discussed later in Section 3.3).

All these systems (with the exception of that of Mulder et al [49] and Nieman et al [51]) are however, single-threaded (here meaning that there is at most one hypothesis under consideration at any one time) which reduces the range of problems they can handle (or alternatively, increases the complexity). Moreover, the systems are complex, non-modular constructions which impede easy understanding and implementation.

2.2 A German School

In Germany a number of related projects have been exploring high-level vision based on sequences of images, and outputting German language interpretations.

One example is the traffic interpretation system of Schirra et al [61]. The scene is segmented by motion vectors resulting in identified objects in the sequence, which are described in terms of enclosing rectangles corners and displacement vectors. The output from the low level system is described in terms of static and dynamic objects (geometric scene description or GSD). This data is fed to the language system CITYTOUR ([56]), in which the user is assumed to be in the scene, and interacts with the language system which answers questions about the scene from the users viewpoint. This system handles static spatial relationships, for example, *in front of, behind, to the right* and the procedures then produce degrees of applicability (belief) for the relationships which are used for linguistic hedges (words like *maybe, possibly*). It should be noted that the static data is hand generated, and only the dynamic data is computer generated. Further, the link between the vision and language subsystems is one way (no feedback) and are, in fact, connected via TCP/IP between different cities !

Another approach similar to Schirra et al's is SOCCER (Andre et al [4]) which generates soccer descriptions of image sequences from soccer games. Static parts of the scenes are manually generated, dynamic parts are fed in from a hypothetical vision system, in the form of a geometric scene description (same terminology as Schirra et al's). They employ a data driven bottom-up strategy, using case models to recognize events using a transition net, which, in turn, triggers language output.

Neumann [50], takes 3D geometric scene description sequences, which contain data on the time, location and orientation of objects found in traffic scenes (but actually hand generated), and use a failure-driven / backtracking algorithm on a semantic network to generate case frames (or thematic roles, an objects thematic role specifies the objects relation to an act [68]). These frames are used as the interface to a more formal logic programming language

for the high level interpretations.

These German systems are grouped together because they use the same representation for the low-level data (GSD), and generate a similar level of output (German language). They all use a top-down strategy for generating the language output, in the first case through interacting with user queries, and in Neumann's case through his semantic net and case frame techniques.

However, like the graph-based systems discussed above, these systems also suffer from complexity, containing a number of sophisticated subsystems using a variety of different techniques working together via heuristic constraints. One way of overcoming such cases of complexity (and difficulties with software) is to develop systems with intrinsic concurrency. The object-oriented approaches sets up the framework for such systems.

2.3 Object Oriented Approaches

In the 1990s, a new generation of vision systems was developed, those based on the object-oriented paradigm. This approach overcomes some problems of the earlier systems in that the systems are organized in a modular fashion, with a coherent design philosophy which hides the individual subsystem data-structures within objects.

Feri et al [30] describe a blackboard system, based on a single-threaded geometric reasoner controlling a hierarchical system of knowledge sources (objects) in bottom-up fashion. It identifies 3D objects from data fusion of monochrome and infra-red images, based on prior knowledge of geometric constraints. Their system is able to generate low level object descriptions such as CAD models, but makes no higher level interpretations.

SIGMA developed by Matsuyama and Hwang [45] is a system that finds objects in aerial monochrome views of housing estates. It uses an object-oriented approach, where each object instance is used to establish a concept, and the control is by a sequential failure-driven reasoner working on the equivalents of Horn clauses (in fact the same control mechanism as that used in Prolog). The low-level processing is under the control of the higher-level rea-

soning. However, the system does not go beyond identifying objects in the scene to include, for instance, high-level concepts like "what kind of housing estate?", or "what are the best utility routings?".

Bell and Pau [7] developed an object-oriented logic programming system for picture interpretation. Their system is based on the Prolog failure-driven/backtracking hypothesize-and-test control mechanism. The object oriented component of the system is implemented with a preprocessor that translates the code into standard Prolog. This three level (feature, application dependent, and object identification) system is used to find objects (cars) in natural scenes.

Although object-oriented, these systems are still single-threaded, and, in fact, sequential (as opposed to concurrent). This imposes restrictions on the range of problems they can easily deal with. In Section 3.1 we explore an alternative to this approach.

2.4 Other Recent Systems

Two interesting recent systems that do not fall into the object oriented paradigm are the following:

Bobick and Bolles [9] introduced a bottom-up system based on a representation space which is a lattice of descriptions, from local image regions to more general descriptions. This includes multiple views as well as refinement and augmentation of the description. This representation scheme is used in a real time system for robot vehicle vision. Sequential images allow improving resolution as the robot moves through the landscape. They eliminate processing artifacts through detecting temporal stability of objects through the image sequence.

Huang et al[41] use a Bayesian belief network and inference engine (HUGIN [3]) in sequences of highway traffic scenes to produce high-level concepts like "car changing lane" and "car stalled". In general, belief networks propagate values around the network as vectors, with each link having associated matrices reflecting the conditional probabilities [52]. For this rea-

son, Huang et al's system is regarded as bottom-up, having a lot in common with the connectionist paradigm. One problem with Bayesian inference is that each node must have a set of exhaustive and mutually exclusive states, which is often difficult to obtain in vision. This problem is further explored in Section 7.4.1.

Tables 2.1 and 2.2 summarize the survey of vision systems given above. The first table is divided into sections (as in the text) corresponding to systems based on ATN, semantic net, constraint propagation and the German group of systems. The second table, corresponding to systems published in the 1990s, is divided into object-oriented systems and the rest.

Paper	Input	Method	Output	Remark
Tropf 1983 [64]	images of objects	ATN	objects	single thr. LL
Bajcsy 1985 [5]	aerial stereo pairs	ATN	scene model via user queries	single thr. LL
Nieman 1985 [51]	images of human heart sequence	production rules over semantic net controlled by A* search	diagnostic descriptions	multi thr. HL
Dellepiane 1987 [23]	sets of slices from brain tomography	prod. rules over semantic net with hyp. & test	brain components	single thr. LL
Govindaraju 1990 [33]	newspaper images	semantic net relates language to vision using hyp. & test.	relates faces to caption text	multi thr. HL
Mulder 1987 [49]	sketch map encoded by plotter cmds	local constraint propagation over graphs	labeled map	multi thr. LL
Boddington 1990 [10]	images of cars	constraint propagation and ATMS	found cars	multi thr. LL
Schirra 1987 [61]	video traffic sequences	user query driven geometric reasoner	traffic descriptions	single thr. HL
Andre 1988 [4]	video sequences of soccer	bottom-up event recognition with transition net	description of soccer match	multi thr. HL BU
Neumann 1989 [50]	video traffic sequences	hyp. & test over semantic net	descriptions of traffic events	single thr. HL.

Table 2.1: *Survey of status of vision systems, pre 1990.* Single thr. *means single-threaded, referring to systems that process no more than one hypothesis at any instant, as opposed to* multi thr.. LL *means low level output,* HL *means high level output. All systems are top-down except those labeled BU (bottom up). Note: for brevity, only the first author on each paper is named.*

Paper	Input	Method	Output	Remark
Draper 1989 [27]	natural images	concurrent processes and blackboard	region labels	multi thr. LL OO (Sec. 3.2)
Feri 1990 [30]	visual & IR images	bottom-up geometric reasoning	region labels	single thr. LL BU OO
Matsuyama 1990 [45]	aerial views of housing estates	hyp. & test over Horn clauses	objects	single thr. LL OO
Bell 1992 [7]	car images	OO Prolog extension	objects	single thr. LL OO
Bobick 1992 [9]	real-time robot image sequence	lattice of desc'n in finite state machine, bottom up processing	objects	single thr. LL BU
Huang 1994 [41]	highway traffic sequences	Bayesian belief inference engine	traffic situation desc'ns	multi thr. HL BU

Table 2.2: *Survey of status of vision systems, post 1990.* Single thr. *means single-threaded, referring to systems that process no more than one hypothesis at any instant, as opposed to* multi thr.. LL *means low level output,* HL *means high level output. All systems are top-down except those labeled BU (bottom up).* OO *means object-oriented architecture. Note: for brevity, only the first author on each paper is named.*

Chapter 3

Foundations

In this chapter, we consider the foundations of some of the ideas used in the design of our system SOO-PIN. We show how the three important strands of: frames, logic programming and object orientation come together in the language Parlog++, and how this is the optimal vehicle for our network-of-frames.

3.1 Human Conceptualization

The systems described in the previous chapter, including the object oriented ones, all have a rather monolithic design based on the sequential processing paradigm. While this is legitimate, it does not capture the ways in which humans describe images, therefore alternative design philosophies are required. From a psycholinguistics viewpoint, Lakoff[43] has pointed out that human categorization cannot be defined in terms of "necessary and sufficient conditions" (which we take to mean a conjunction of propositions), but rather is explained in terms of "idealized cognitive models" (ICM). ICMs are active structures (processes) that define categories in terms of prototype effects, ie, objects are members of categories to the extent that they relate to the prototype via the following:

- "frame-like" structures,

- "image-schematic" structures, which map between spatial concepts having similar schemas, (for example, the container schema refers not only to a cake in an oven and tea in a cup, but also to categories and members),
- metaphor, or structural similarity,
- metonymy, or similarity of salient features or parts.

Lakoff's ideas bear a striking resemblance to those of Herskovits [36], who investigated the spatial prepositions *at, in* and *on* in English, and how they were used. She went on to look at "projective prepositions" which relate one object to another in space. She claims that knowing the objects and their coordinates is not sufficient to select the appropriate preposition. Spatial prepositions have "ideal meanings" which are qualified in the real world by "sense shift" and "tolerance shift". Sense shifts are discontinuous shifts to another, conceptually close relation. For instance, the word *on*, whose ideal meaning is that of *support from below and contiguity*, when used in the expression "the apple on the branch", has been sense-shifted to mean *support from* **above** *and contiguity.* Tolerance shifts involve gradual measurable deviations. For instance in the expression "the book is on the table" when there is a table-cloth between the book and the table [36].

The ways in which objects relate to their ideal meaning, according to Herskovits, is via the following:

- **salience** - referring to the part or aspect of an object that is relevant in the context. For instance, "the queue at the counter" refers to the salient part of the queue, namely the head of the queue, being next to the counter rather than any other part of the queue.
- **relevance** - aspect of relation to be emphasized in the context. For instance, the expression "the light bulb is in the socket" uses the word "in" to emphasize the bulbs function, ie, whether it will

work, rather than just its position, where the word "under" would have been just as good.

- **tolerance** - how much deviation from the implied ideal varies with context.

- **typicality** - similar to default reasoning. For instance, the expression "the fountain is behind the town hall" implies that the fountain is near to or next to the town hall, but the ideal meaning of "behind" does not involve proximity, proximity is the default assumption.

Lakoff and Herskovits' ideas suggest the need for mediating active and intelligent agents operating between concepts in the system and the world. Below we explore one scheme, namely frames, mentioned by Lakoff, for implementing such agents.

3.2 Frames and Related Ideas

The notion of frame-like structures emerged in the mid 1970s and was crystallized by Minsky[47] in his 1975 essay "A Framework for Representing Knowledge". In this essay he outlined an approach to the fields of both vision and natural language processing based on the notion of a "frame", which is a data structure for representing a stereotyped situation. Attached to each frame are "terminals" (or slots) in which specific information about the frame is stored. Terminals can contain "demons" for calculating and validating information, or default values, or links to other frames. If sufficient terminals find mismatched data, the frame may invoke another with better matching terminal constraints. Thus, in a vision example, when entering an unfamiliar room, the room frame terminals contain default values like left wall, right wall, ceiling but with no details. The terminal demon for, say, the left wall actively seeks out information to instantiate its values and, if necessary, activates the wall subframe. If the wall demons cannot find good values, then the "room" frame may be replaced by, say, a "backyard" frame. This system has elements of

top-down and bottom-up control: bottom-up when a new frame is activated or from the tension of a mismatched frame, top-down when a frame activates its terminal demons.

Similar ideas put forward around the same time are Carl Hewitt's "actors" and Roger Schank's "scripts". An actor[37][38][70] is a potentially active piece of knowledge (procedure) which is activated when it is sent a message. Actors interact by sending messages to other actors, which can take place concurrently. An actor system is defined by the following:

- what actors (objects) exist,
- what messages they receive,
- what they do upon receiving a message,
- what acquaintances (other known actors) each actor has.

Scripts[57][60] are stereotyped sequences of events in a particular context. Scripts consist of a set of slots which have indications of the kind of data within them and possible default values. Scripts differ from frames insofar as they refer to specific events and the slots have specific information in them common to such events, for instance, "entry conditions", "results","props","roles" and "scenes". This formalism is based on Schank's earlier work on "conceptual dependency" – the idea that most events portrayed in natural language can be expressed as graphs made up of a small number of primitive acts with "dependencies" between them.

Frames, and the related ideas, were taken up by the natural language processing community, but, until recently, vision applications have been limited. However, a recent example in vision is SCHEMA, developed by Draper et al[27]. In this system there are concurrent processes communicating through a blackboard and the control is frame-based with structure matching and forward chaining. This system is used to segment natural 2D images, but is not concerned with high-level interpretation. It should be pointed out that in such systems one needs to keep the interconnectivity between frames reasonably low, or the time spent handling non-productive messages becomes untenable

as the cardinality of the system goes up. Blackboard systems are subject to this, unless one introduces some kind of partitioning of the blackboard. However, in this case, the blackboard system is then logically equivalent to a network-of-frames with appropriate connectivity.

The systems of Bell and Pau[7] and SIGMA[45], do not qualify as networks-of-frames because failure-driven backtracking is incompatible with such an approach. That is, frames, as we define them, are message passing processes that exist in real time and cannot directly retract (ie, backtrack) a message once sent. For instance, if two modules A and B are in communication, and module A experiences a failure and backtracks to some previous decision point, and if, in the meantime, it has sent a message to module B, then in general the system state cannot be reverted to the state which existed at the time of the decision point unless the effects of the message on B were also backtracked. However, if this mechanism existed, then the independence of the modules would be compromised.

In this thesis we develop an alternate implementation of Minsky's frames idea, involving a network of concurrent processes, each dealing with one concept or aspect of the interpretation. One novelty of this system is that the network is extended beyond the object recognition stage to higher-level concepts based on interactions between objects and rules (knowledge, road laws, etc) – for instance "car A gives way to car B due to *road rule X*". This is comparable to the level of output from Neumann's[50] system which generates concepts like "car A is overtaking truck B". However, in Neumann, reported interactions between cars is limited to their spatial relationships and do not refer to road rules or the drivers' intentions.

Another novelty in this system is the use of an object-oriented concurrent logic programming language for its implementation. This environment offers several advantages:

- the high-level declarative nature of logic programming allowed a fast prototyping of the system. It also provides symbolic processing, and pattern matching.

- object-orientation reflects, to some extent, the active and modular

nature of human concepts, as Lakoff[43] pointed out, and eases the construction and modification of the system.

- concurrency allows a full and natural implementation of the frames concept. It also positions this system to take advantage of parallel machine architectures.

3.3 Logic Programming

Logic programming (LP) effectively began with Robinson's [59] discovery of the resolution principle. This has since evolved through systems such as Planner[39] to its best known manifestation, Prolog. This is a language based on Horn clause logic (which is a subset of first order predicate calculus) and the resolution principle with a failure-driven backtracking control mechanism. However, Prolog does not exhaust the possible logic programming paradigms. LP languages have different interpretations. For instance, Prolog can be interpreted declaratively, in which the program is regarded as a logical statement, and its meaning is the set of ground formulae that its clauses imply; or procedurally, in which the program is regarded as a precise specification of how the inference will be controlled through time. The procedural interpretation of Prolog is single-threaded.

There is another class of LP, concurrent LP, where the language is a collection of Horn clauses together with the parallel evaluation of a goal with respect to these clauses. This class has a third interpretation based on the idea that goals can be regarded as a system of *concurrent processes*. Concurrent Prolog and Parlog[17] are such languages.

Parlog differs from (sequential) Prolog in how clauses are chosen for execution, Prolog executes each clause in a goal sequentially, backtracking and trying the next clause upon failure. This is the basis of its depth-first search. Parlog has a construct called the "commit" operator which delimits a "guard" condition in a clause – since all the clauses of a goal can be tried concurrently, the first clause for which the guard succeeds is committed to (ie, that clause is selected for evaluation, the rest are ignored), with no backtracking. This is called "don't care" or "committed-choice" non-determinism as opposed to Prolog's "don't know" non-determinism. This mechanism in which alternate clauses are tried concurrently is an example of "or-parallelism". Another concurrency mechanism available in Parlog is "and-parallelism", in which, when a goal is dependent upon the conjunction of several subgoals, the subgoals are evaluated concurrently.

To further elaborate the process interpretation of LP, below we relate

process concepts to the corresponding logic programming concepts:

- a long-lived process corresponds to a goal which calls itself recursively.
- A process state corresponds to an unshared variable that a recursive goal passes to itself as an argument upon the recursive call (and thus propagates values from call to call through time).
- Two processes in communication via a channel can be modeled by a conjunction of such recursive goals sharing a variable (which is itself a recursive data structure - ie, a list). For instance, one process, called the producer, instantiates the head of the list to a value, then calls itself recursively with the rest of the list (called the tail) as an argument. The other process, called the consumer, waits for the head of the list to become instantiated. When it is instantiated, the consumer processes the head and then, like the producer, calls itself recursively with the tail as an argument. See Figure 3.1.
- Reply to a query (back channel communication) corresponds to the producer sending an uninstantiated variable to the consumer through the channel. Upon receiving the variable, the consumer instantiates it to a value which is thus available to the producer.

In this way many of the properties of concurrent programming can be mapped into Parlog[58].

3.4 Object Orientation

The object-oriented programming (OOP) paradigm is a model for programming based on the notion of computational objects which communicate via message passing. This paradigm is similar to Minsky's[47] frames and Hewitt's[38] actors described above (in fact Hewitt was influenced by an early version of SMALLTALK). OOP is characterized by the following features:

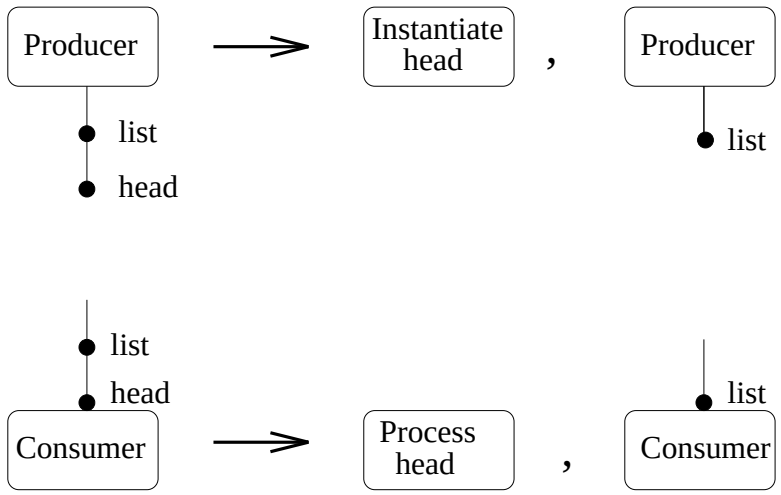

Figure 3.1: The communication channel in Parlog implemented using a shared list. On the left the producer and consumer goals are shown, with their shared variable – the list with its head – as the vertical line with the black disks. On the right is shown the result of the evaluation of the two goals, with the producer instantiating the head to a value to be communicated and then calling itself with the remainder of the list, and the consumer processing the instantiated head and also calling itself with the remainder of the list. This diagram has been adapted from [58].

- creating an object
- sending and receiving messages
- modifying an object's state
- forming class-superclass hierarchies

Shapiro and Takeuchi[62] have shown how these features can naturally be implemented in Concurrent Prolog through the following identifications:

- An object is a process that calls itself recursively and holds its internal state in unshared arguments.
- Objects communicate with each other by instantiating shared variables.

- An object becomes active when it receives a message, otherwise it is suspended.
- An object can respond to a message either by sending the answer back in another message, or instantiating an uninstantiated answer variable in the message.
- Class hierarchies and inheritance can be formed using filters (Shapiro and Takeuchi [62]), which actively pass messages from a class to the superclass when the method to handle a message is not available in the class.

Parlog++[21] is an object-oriented extension to Parlog. The OOP features (except for hierarchies and inheritance) described by Shapiro and Takeuchi are built into the language so that the programmer can write object-oriented code in a natural manner. As Shapiro and Takeuchi point out, their view of objects is based on Hewitt's actor model of computation, which in turn is influenced by and influences Minsky's frames model. Thus Parlog++ is the most suitable language in which to implement our network-of-frames system since it embodies the concurrent frame-based concepts required whilst being a high level logic programming language with the consequent symbolic processing and pattern matching capability.

Chapter 4

Architecture of SOO-PIN

In the last chapter, it was argued that the ideal image interpretation system is a concurrent frame based network. It was also argued that a suitable language in which to implement such a system is the Parlog++ language. What is now required is a system concept and architecture. Hewitt's [38] prescription for defining an actor system (pg. 17), provides a guide for describing SOO-PIN, as explained below. His first point, "what actors exist", is now dealt with.

What is envisaged is a network of independent frames (which are here called "concept-frames"), each dealing with an aspect of the overall interpretation. To this end they communicate with other concept-frames, and also access the (processed) image itself. Concept-frames store instances of the concept it is concerned with (which are called "concept-instances") as data (as opposed to some object-oriented systems where instances are themselves objects or classes). Each concept-frame carries its code (or *methods*, in OO language) which determines its behavior.

The next two decisions in Hewitt's list, "what messages the actors receive" and "what they do upon receiving a message", are now dealt with. The problem of what messages to convey around a network is similar to the situation dealt with by Green [34] in the distributed artificial intelligence (DAI) context. He deals with real-time systems in the area of robotics, and his system, like SOO-PIN, also partly derives from the work of Minsky and Hewitt. In his system, activation framework objects (AFO) form a community of experts

which communicate by means of message exchange. These messages come in 3 main types:

- automatic: if the AFO is given or deduces some new piece of information, it will inform those AFOs that it knows and can use this information.
- on-demand: one AFO can ask another about the current value of an hypothesis, or can ask for the AFO to evaluate some data it is sent, or data that it needs to acquire to respond to the request.
- suggestive: in this mode, one AFO sends a message to another that results in a change in the activation level of the evidence for or against some hypothesis that is in the domain of the second AFO.

He points out that automatic messages correspond to forward chaining (or bottom-up), and on-demand to backward chaining (or top-down).

In accordance with Green, SOO-PIN has two classes of message, query and informative, which correspond to Green's on-demand and automatic message types respectively. The query messages contain queries like "list all concept-instances of a certain concept-frame with a certain property" (called in SOO-PIN an `anyInst` message), or "does a certain concept-instance have a certain property" (the `getVal` message). The informative messages, in bottom-up mode, inform concept-frames about the existence of a concept-instance in another concept-frame, and suggests the recipient check specific relations regarding it (the `check` and `create` messages)[1]. The other informative message simply updates specific concept-instances with new properties, and do not initiate any other activity (the `updVal` message). SOO-PIN has no analogy to Green's "suggestive" class of messages, as this relates to the scheduling mechanism used there, although it does bear some resemblance to the `belUpd` message introduced later in Section 7.4.2.3.

[1]It is worth noting that the informative messages have the effect of a spreading activation through the network. This is analogous to the mechanism used in the system GRANT (Cohen et al [53]) in which funding agencies are matched with funding applicants.

The final prescription in Hewitt's list – "what acquaintances each actor has", is reflected in SOO-PIN by the list in each concept-frame of other concept-frames that are sent information in automatic (bottom-up) mode. However, SOO-PIN allows concept-frames to react to messages from anywhere.

4.1 Procedural Description of SOO-PIN

In accord with the prescription given above, we now describe how SOO-PIN behaves for a sample case in which the system recognizes a bicycle from low level data.

When a concept-instance of the `wheel` concept-frame (see Figure 4.1) is created –for whatever reason– it sends messages `checkA` to all associated concept-frames (according to how the system has been connected to reflect the domain knowledge, as per Hewitt's acquaintance list) informing them of its existence. These messages, in turn, prompt receiving concept-frames (ie, the `bicycle` concept-frame) to check their "existence criteria", possibly sending inquiry messages `inquiryB` to other concept-frames to establish the criteria, and, if successful, they create a concept-instance (ie, a bicycle instance). This new creation, again, results in `checkB` messages being sent, for instance, to the `street` concept-frame, and thus completing the loop. The existence criteria, and messages sent as a result of existence, for all the interconnected concept-frames in the system, constitute the "World Model" (as Bajcsy et al [5] call it) – meaning the prior knowledge about the situation that gives meaning to the low-level data in a given image. The specific state of the system as a result of the low level data – the "Scene Model" [5] – is embodied in the list of instances for each concept-frame.

As pointed out above, this architecture embodies both data-driven and knowledge-based control, where the automatic (**check** and **create**) messages represent the data-driven mode, and the on-demand (**getVal** and **anyInst**) messages represent the knowledge-based mode, in that these messages are prompted from the exploration of higher-level hypotheses. Using these two

modes together avoids the large search space that results from using either alone. For instance, knowledge-based control can entail exhaustive depth-first search as the system begins searching at the top nodes of the search tree, and must check all nodes down to the data level, usually with failure-driven backtracking. On the other hand, data-driven search modes entail combining all data in all possible patterns in order to explore higher nodes in the search tree, and as Tsotsos [65] has pointed out in the area of vision, this is NP-complete (meaning that it has exponential complexity). Our claim that mixing top-down and bottom-up modes is more efficient than either alone, is backed up in the area of linguistic parsing by Allen [2], who gives mixed-mode chart parsing as an example.

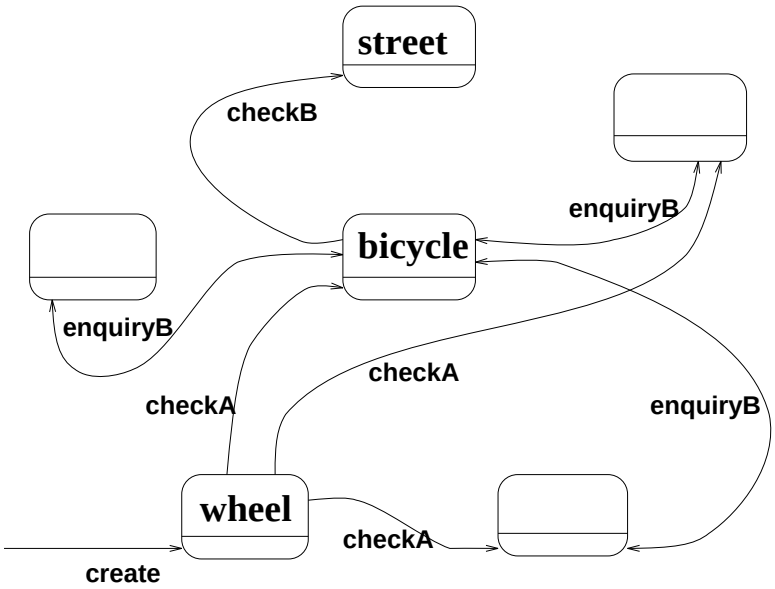

Figure 4.1: The SOO-PIN system concept: object concept-instance creations activate other associated objects through sending "check" messages. These activations in turn prompt further activations.

SOO-PIN requires a mechanism whereby a number of concept-frames can communicate with each other. One means by which this can be arranged, using a mechanism pointed out by Ringwood [58], is to use a central process with which all concept-frames are in a client-server relationship. The output

channels from all these clients (the concept-frames) can then be merged together and input into the server. In SOO-PIN, this server process is called the "switchboard". Thus the function of the switchboard is to keep track of the concept-frames' input channels and redirect messages from one concept-frames to another (see Figure 4.2). It is also the switchboard's job to spawn off new concept-frames as required. In Parlog++, spawning takes place through invoking goals using and-parallelism, as mentioned in Section 3.3. The concept-frames and switchboard also write output messages directly to windows on the user console.

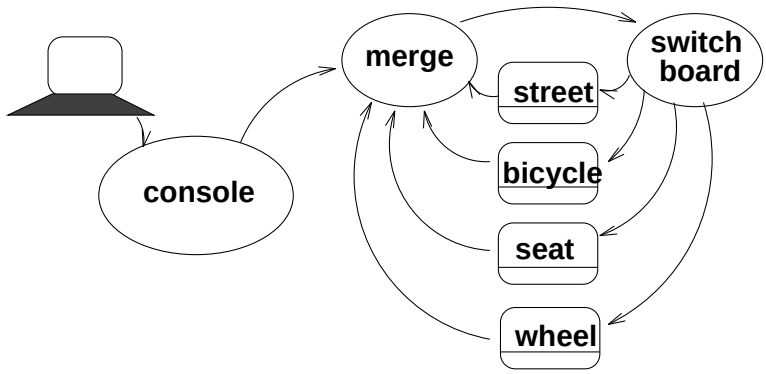

Figure 4.2: The implemented system architecture. Messages are relayed around the system by being merged into the switchboard's input channel, and then distributed to various concept-frames (shown as rectangles). Ellipses represent other concurrent processes, and arrows are message channels.

It should be pointed out that because SOO-PIN uses a network-of-frames for its control-mechanism, and is implemented in a logic programming language, it falls into Bunke's [12] hybrid category of PC-SN (Predicate Calculus – Semantic Net) in his overview of pattern recognition architectures.

Having sketched the structure of SOO-PIN, it is now necessary to fill it out by considering some of the other constraints and requirements of the system. One of the prime considerations with SOO-PIN is that it is not simply a logic programming symbolic inference engine, but deals specifically with **image interpretation**. Hence the following section is concerned with

how spatial data is integrated into SOO-PIN. This theme is taken further in the following section (Section 4.3) when we consider the interplay between the spatial data of SOO-PIN and high-level concept-frames, especially those of compound objects.

4.2 Spatial Data

If a compound object C is composed of objects A and B, then an object D may be near C but not near A or B. That is, "nearness" depends on the *scale* of the objects involved. This is not something that can be handled purely symbolically, but must refer to the spatial data underlying the objects A, B and D. Thus, when the system needs to instantiate the nearness predicate for D and C, it first needs to determine the components of C, and then the components' spatial coordinates. If there is a deeply nested set of components the system would need to search down the component tree for the components with spatial data. This could be a large search and adds to the complexity of the system. For this reason, in SOO-PIN, when a compound object is created, an entry is inserted in the spatial database with the spatial coordinates of the new object. These entries are calculated from the coordinates of its components, and are used to calculate spatial predicates for compound objects in the same way as for primitive objects.

Of course, the nearness predicate is but one of many potential spatial predicates and, in general, the system could not store a large range of spatial relationships as symbolic predicates. For instance, the system may store the fact that objects A and B are near, but also requires to compute their collinearity. In fact, as Herskovits [36] has pointed out, even the same predicate can vary in meaning depending on the context – illustrating the need to refer back to the spatial data. This is an important principle in the area of computer vision (and other areas of AI as well), and relates to the necessity of embedding cognitive systems in their environment in order to give meaning to their symbols – situatedness. Situationists have made radical claims that cognitive systems *only* use context for their representations, and that there

are no internal representations [16]. However, Slezak [63] has pointed out the need to distinguish the representation used internally to implement cognitive systems from that used for external communication, and how this clarifies some of the situationists' claims. However, in the context of SOO-PIN, this reduces to noting the distinction between the representation of the spatial data and the symbolic language of Parlog++.

4.3 Compound Objects

High level concept-frames (for instance, compound objects) introduce an "identity" problem. When a new concept-instance is to be created, the system must check through existing concept-instances to see if the instance already exists. The question of what constitutes a "unique key" to the concept-instance can be subtle, and depends upon the domain (prior) knowledge of the world. To further elaborate, if a system determines that a car exists at some point in space, but there is already another car there, the knowledge that only one car can exist at any given point would lead the system to identify the two. However, other objects can coexist in space, for instance, cakes and ovens. This question of how ordinary objects work and interact has been discussed by Hayes [35]. In SOO-PIN, each concept-frame has its domain knowledge built in. Thus the "existence criteria" of concept-frames embody the question of the identity of concept-instances.

Domain knowledge also comes into play in the case where "negative information" is generated, for instance, from the deletion of a concept-instance. Here the receiving concept-frame applies its domain knowledge to decide whether any concept-instance critically depends upon the sender. If so, it too is deleted. However, the situation becomes complex when evidence from several sources justify the existence of the concept-instance. Deletion happens when, for instance, a bicycle is found to be joined to a third wheel (in the case of a tricycle), or an existing component wheel is deleted (making it a unicycle). This is an example of "non-monotonic reasoning" [46]. It is not compatible with traditional predicate calculus, but in logic program-

ming systems can be implemented procedurally. Humans use non-monotonic reasoning often, for instance, when a default assumption is over-ridden by subsequent information. Thus, in general, frame-based systems also deal in non-monotonic reasoning because they too use slots with default values.

4.4 Concept-Frame Structure

This section defines the syntax of the data structures used in SOO-PIN concept-frames and instances, and in the messages.

The concept-frame needs to store the concept-instances together with their properties. The mechanism chosen to do this is the list, a data structure used extensively in Parlog++, for instance, in implementing the data channel (see Figure 3.1). Thus the concept-frame maintains a list of concept-instances, which have the following structure:

```
inst(Id,PropList)
```

Here, each concept-instance has its identity string, `Id` above, and maintains a list of its properties, be they unary or binary, as the `PropList` list. These structures are further broken down in the following table:

```
Id          = id(ObjectType,InstNumber)
ObjectType  is the same as the concept-frame name
InstNumber  is a unique identifier for the concept-instance
PropList    is a list of Prop
Prop        = reln(RelnType,OtherId) or desc(Desc)
RelnType    is any relationship, ie ``near'', ``joined'', ``above''
OtherId     is the Id of the other party to the relationship
Desc        is a structure giving a description, ie ``colour(green)''
```

Below is a schematic example of a typical concept-frame, showing its reaction to received messages. The code can be read, between the "clauses" and "end" statements, as a simple case statement (as in the language C), where the system, upon receiving a message, searches down the list of clause heads for one that matches the message. The clause body is then evaluated (not shown here).

```
bicycle.
        Out o-channel      %output channel name
invisible InstList state, OutFile state
                %state variables

initial open(bicycle.log,write,OutFile)
                %activities when starting up concept-frame
clauses
        create(Id,PropList)
                %Concept-instance Id is added to InstList
                %with its Props

        check(composedOf)
                %checks if a concept-instance should be created

        check(Prop)
                %checks if a concept-instance has relationship Prop
                %with sender.

        negCheck
                %sender has been deleted,
                %checks if a concept-instance should be deleted
                %in turn.

        inquiry(Id,Prop,FoundProps)
                %Prop is searched for in concept-instance
                %Id, and the answer bound to FoundProps

        update(Id,Props)
                %Props are added to property list of
                %concept-instance Id

        WrongMsg writeMy(OutFile,[WrongMsg]).
                %default clause for erroneous messages

        last
                %clause to execute upon input channel close

end.    %end of concept-frame
```

The first message received by the concept-frame activates the "initial" clause, the persistent data of the concept-frame is stored in the "state" variables defined in the "invisible" section, output messages to other concept-frames are placed in the "o-channel" declared after the concept-frame name, user messages are written to the text file defined in the initial section. Erroneous mes-

sages fall through to "WrongMsg" where they generate error text messages, and finally, when the input channel closes, the "last" clause is executed.

In accordance with the parallelism of Parlog++, all the clauses in the above code can execute in parallel, spawning a process for each message on the input channel. They can also run sequentially, finishing one message before starting the next. In SOO-PIN, the concept-frames run in parallel with each other, while within a concept-frame, the code is implemented in parallel for simple situations like sorting a list. Everything else is executed sequentially.

Table 4.1 shows the action initiated by each message. This varies with the logic built into each concept-frame, but these are the overall consistencies.

```
MESSAGE
        ACTION
                RESULTING MESSAGES
create(Id)
        creates a concept-frame if it does not already exist
                check message sent to associated concept-frames

check(composedOf)
        checks if concept-instance already exists in relation
        composedOf with sender.
                If so, update sent to sender with this fact.
                If not, then sends inquiry to associated
                concept-frames to satisfy existence criteria.
        checks if sender is a new component of existing instance
                update sent to sender with this fact.
        checks if this is a new concept-instance,
                check sent to associates of new instance,
                update sent to sender with "composedOf" Prop.
        checks if any concept-instance should be deleted as a
        result of this information,
                negCheck sent to associates of deleted instance.

check(Prop)
        checks if any concept-instance exists in rel'n Prop with sender
                sends inquiry messages to determine this.

negCheck(Id)
        informs receiver that Id has been deleted, may result
        in deletion of this concept-frame.
                If so, negCheck sent to associates.

inquiry(FoundList,Prop)
        returns in FoundList a list of instances of this
        concept-frame satisfying Prop, or list of
        properties of specific instance satisfying Prop.

updval(Id,Prop)
        updates Id with Prop properties.
```

Table 4.1: *Table showing the actions performed, and the messages sent, by a concept-frame as a result of receiving various messages. The received messages are shown in the left column, the actions are given in the middle column, and the resulting messages in the right column.*

Chapter 5

Simple Scenarios

Having described the motivation and structure for SOO-PIN, we now consider two simple implementations which demonstrate its power. First we deal with the "cutlery" scenario, which uses no spatial database but calls upon predefined relational predicates to drive the network. Second, we describe the "wheels" scenario, which extends the implementation to include "negative" information (see Section 4.3) in a more complex network.

5.1 Cutlery Scenario

Consider a camera observing a tabletop with various objects on it. We also assume that the low-level vision problem of identifying and locating the various objects has been solved, and the job of SOO-PIN is to interpret the image. The scenario (see Figure 5.1) consists of the primitive concept-frames `knife`, `fork`, `spoon` and `chopstick` which are created initially from the low level vision system. The higher-level `stick_set` embodies the concept of a pair of near and parallel chopsticks, `setting` which deals with various combinations of knife, fork and spoon, `chinese_setting` which similarly deals with stick_sets and spoons, and at the top level is `dinner` and `yumcha`, which are interpretations of the scene as a whole. Note that the primitive concept-frames, ie, knife and fork, have no existence criteria as they are created directly from the low-level data.

The spatial database in this scenario consists of various assertions about which objects are near and parallel to each other, which is sufficient to group the various cutlery pieces into "table settings".

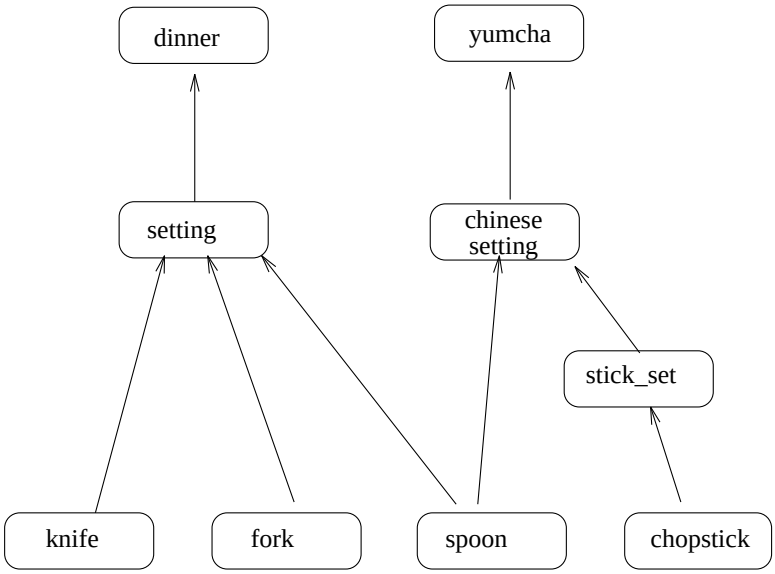

Figure 5.1: *The cutlery scenario network-of-frames. The arrows refer to* `check(composed-Of)` *and* `create` *messages, inquiry and update messages are not shown.* `stick_set` *refers to a pair of chopsticks.*

Creation of the primitive concept-frames (`knife`, `fork`, `spoon` and `chop-stick`) results in the following messages:

```
knife:
      check near to fork
      check near to spoon
      check composedOf to setting

fork:
      check near to knife
      check near to spoon
      check composedOf to setting

spoon:
      check near to knife
      check near to fork
      check near to stick-set
      check composedOf to setting
      check composedOf to chinese-setting

chopstick:
      check near to spoon
      check composedOf to stick-set
```

The next level in complexity is that of compound objects. Defining these concept-frames requires more information, namely, the "existence criteria", the "uniqueness criteria" (ie, does the system add the sender to an existing concept-instance, or create a new one), and the messages sent upon new concept-instance creation.

Note that `chopstick` checks component-hood only with `stick-set`. As a single chopstick is not an acceptable implement in the `chinese-setting`, it is the `stick_set` that behaves like a cutlery implement.

```
Existence criteria:
        chopstick sending message is joined and
        parallel to another chopstick
Messages sent upon creation:
        check near to spoon
        check composedOf to chinese-setting
```

`Setting` results from a knife, fork or spoon in certain spatial relationships:

```
Existence criteria:
        sender is near and parallel to other knife, fork
        or spoon.
Uniqueness criteria:
        sender is not near and parallel to any cutlery
        already in a setting.
Messages sent upon creation:
        check composedOf to dinner.
```

`Chinese-setting` results from a `stick-set` and a `spoon` being in a certain spatial relationship:

```
Existence criteria:
        stick-set sending check message is near and
        parallel to spoon.
        spoon sending check message is near and
        parallel to stick_set.
Uniqueness criteria:
        sender is not near and parallel to any cutlery
        already in a setting.
Messages sent upon creation:
        check composedOf to yumcha.
```

`Dinner` is deduced from the activation of one or more `settings`:

```
Existence criteria:
        Any setting passed.
Uniqueness criteria:
        Any subsequent settings are added to the first dinner created.
No messages are sent upon creation as dinner is a top level production.
```

`Yumcha` is deduced from the activation of one or more `chinese-settings`:

```
Existence criteria:
        Any chinese-setting passed.
Uniqueness criteria:
        Any subsequent chinese-settings are added to the first yumcha
        created.
No messages are sent upon creation as yumcha is a top level production.
```

5.2 Trial Runs of Cutlery Scenario

This first simple system was initialized with:

```
create(id(knife,1)
create(id(fork,3)
```

and the "spatial database" was loaded with the following predicates:

```
near knife 1 fork 3
near fork 3 knife 1
parallel knife 1 fork 3
parallel fork 3 knife 1
```

When the system was run with this input, the following messages were passed through the `switch` concept-frame. Note that messages are structured `msg(target,message_body)`:

```
1   msg(knife,create(id(knife,1),[]))
2   msg(fork,create(id(fork,3),[]))
3   msg(knife,check(reln(near,id(fork,3)),_))
4   msg(spoon,check(reln(near,id(fork,3)),_))
5   msg(setting,check(reln(composedOf,id(fork,3)),_))
6   msg(knife,anyInst(_2362,reln(near,id(fork,3))))
7   msg(fork,check(reln(near,id(knife,1)),yes))
8   msg(spoon,check(reln(near,id(knife,1)),_))
9   msg(setting,check(reln(composedOf,id(knife,1)),_))
10  msg(fork,updVal(id(fork,3),[reln(near,id(knife,1))]))
11  msg(knife,updVal(id(knife,1),[reln(near,id(fork,3))]))
12  msg(knife,getVal(id(knife,1),reln(parallel,id(fork,3)),_))
13  msg(spoon,anyInst(_5847,reln(near,id(fork,3))))
14  msg(fork,updVal(id(fork,3),[reln(partOf,id(setting,4439))]))
15  msg(knife,updVal(id(knife,1),[reln(partOf,id(setting,4439))]))
16  msg(dinner,check(reln(composedOf,id(setting,4439)),_))
17  msg(fork,updVal(id(fork,3),[reln(parallel,id(knife,1))]))
```

As can be seen, the `knife` and `fork` sent `check` messages to `spoon` and `setting` (messages 4,5), `spoon` had no reaction as there were none, but `setting` reacted by sending inquiry messages (`anyInst` and `getVal`) to `knife` to determine if any concept-instances were near and parallel to each other (6,12). When `setting` was successful at creating an instance, it then sent update (`updVal`) messages informing its components of their new status (14,15,17). It then sent a `check` message on to `dinner` (16). Since `dinner` is created from any `setting`, it, in turn, generated the following output message:

```
*dinner 1 consists of the following 1 settings
    setting 4439 consists of the following pieces
        fork 3
        knife 1

no spoons, perhaps there is no desert!
```

It can be seen that the system generated a high-level interpretation on the minimal input data by a process of each concept-frame dialoguing with its associates. With this architecture it is quite easy to vary the response of the system, and the high-level concept-frames are particularly relevant for this purpose.

The next trial of the cutlery scenario involved an oriental meal. Here the `stick_set` was introduced to demonstrate that the system coped with a compound object that required updating the "spatial database", allowing other concept-frames to interact with it in the same way as for a simple object.

The system was initialized with:

```
create(id(spoon,1)
create(id(chopstick,1)
create(id(chopstick,2)
```

and the "spatial database" was loaded with the following predicates:

```
joined chopstick 1 chopstick 2
joined chopstick 2 chopstick 1
parallel chopstick 1 chopstick 2
parallel chopstick 2 chopstick 1
near spoon 1 chopstick 1
near chopstick 1 spoon 1
parallel spoon 1 chopstick 1
parallel chopstick 1 spoon 1
```

When the system was run with this input, the following messages passed through the `switch`:

```
1   msg(chopstick,create(id(chopstick,1),[]))
2   msg(chopstick,create(id(chopstick,2),[]))
3   msg(spoon,create(id(spoon,1),[]))
4   msg(stick_set,check(reln(near,id(spoon,1)),yes))
5   msg(knife,check(reln(near,id(spoon,1)),yes))
6   msg(fork,check(reln(near,id(spoon,1)),yes))
7   msg(chinese_setting,check(reln(composedOf,id(spoon,1)),_))
8   msg(setting,check(reln(composedOf,id(spoon,1)),yes))
9   msg(fork,anyInst([],reln(near,id(spoon,1))))
10  msg(knife,anyInst([],reln(near,id(spoon,1))))
11  msg(stick_set,anyInst(_10462,reln(near,id(spoon,1))))
12  msg(spoon,check(reln(near,id(chopstick,1)),yes))
13  msg(chopstick,updVal(id(chopstick,1),[reln(near,id(spoon,1))]))
14  msg(stick_set,check(reln(composedOf,id(chopstick,1)),_))
15  msg(spoon,check(reln(near,id(chopstick,2)),_))
16  msg(stick_set,check(reln(composedOf,id(chopstick,2)),_))
17  msg(chopstick,anyInst(_598,reln(joined,id(chopstick,1))))
18  msg(chopstick,getVal(id(chopstick,2),reln(parallel,
    id(chopstick,1)),_))
19  msg(chopstick,updVal(id(chopstick,1),[reln(partOf,
    id(stick_set,4440))]))
20  msg(chopstick,updVal(id(chopstick,2),[reln(partOf,
    id(stick_set,4440))]))
21  msg(spoon,check(reln(near,id(stick_set,4440)),_)),
    id(stick_set,4440)),_))
22  msg(chopstick,updVal(id(chopstick,2),[reln(partOf,
    id(stick_set,4440))]))
23  msg(chopstick,updVal(id(chopstick,1),[reln(parallel,
    id(chopstick,2))]))
24  msg(spoon,anyInst(_3803,reln(near,id(stick_set,4440))))
25  msg(stick_set,updVal(id(stick_set,4440),[reln(near,id(spoon,1))]))
26  msg(spoon,getVal(id(spoon,1),reln(parallel,id(stick_set,4440)),_))
27  msg(stick_set,updVal(id(stick_set,4440),[reln(partOf,
        id(chinese_setting,4441))]))
28  msg(spoon,updVal(id(spoon,1),reln(partOf,id(chinese_setting,4441))))
29  msg(yumcha,check(reln(composedOf,id(chinese_setting,4441)),_))
30  msg(stick_set,updVal(id(stick_set,4440),reln(parallel,id(spoon,1))))
```

It can be seen that amongst a number of messages from the spoon exploring the dinner option (messages 8,9,10), the `stick_set` instantiated itself from the chopsticks (14,16,17,18,19,20,22,23) and determined its proximity and parallelness to the spoon (21,24,25,26). This involved updating the `stick_set` entry in the spatial database with the relationships of its components, the chopsticks. The creation of the stick_set initiated the chinese_setting (27,28)

and thus the yumcha (29), which, at the end of processing, generated the following story:

```
*yumcha 1 consists of the following 1 settings
    chinese_setting 4441 consists of the following pieces
        stick_set 4440
        spoon 1

a lonely yumcha
```

Of course, with a real spatial database, the entry for stick_set need not simply have inherited the relationships of the chopsticks, but could have generated any appropriate entry. This initial prototype uses only symbolic predicates.

In summary, the cutlery scenario demonstrates how the network-of-frames functions with a simple domain, for instance, the concept of mixed top-down and bottom-up control, and concurrent execution of concept-frames, as is evident from the messages passing through the switchboard. Although the spatial data is here emulated with a symbolic list, this implementation also demonstrates a compound object (the stick-set) deriving its spatial properties from its components. In the next section the use of negative information (see Section 4.3) is explored.

5.3 Wheels Scenario

In this scenario, it is assumed that the scene contains wheeled vehicles, the job of the interpretation system is to deduce what the image is about on the basis of what vehicles it finds. Once again, this is a symbolic system only, in that the input consists of declarations of the primitive objects, and, again, the spatial data consists of symbolic forms of their spatial relationships. The vehicles under consideration are the unicycle, bicycle and tricycle. The primitive concept-frames are the wheel and the seat. The interconnections corresponding to the world model are shown in Figure 5.2. One reason for introducing this scenario is to explore the operation of the `negCheck` message. This arises from the `unicycle` concept-frame when it finds a second wheel

attached to a unicycle instance. Another reason is to explore the scaling of the system.

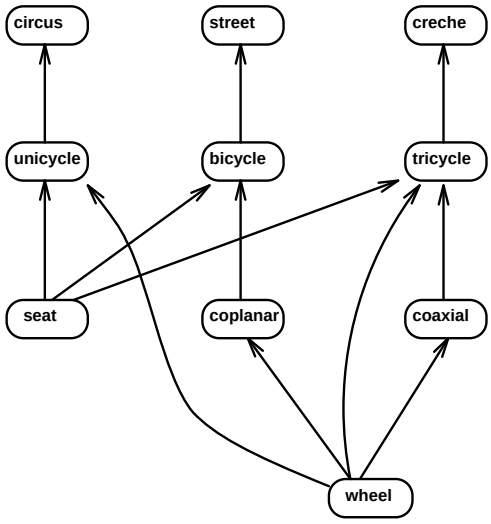

Figure 5.2: *The Wheels Scenario network. Boxes represent concept-frames and arrows show where CHECK(composedOf) messages are sent.*

The starting point is the `wheel` concept-frame. This is a primitive concept-frame and is created directly. Upon creation, "check joined" messages are sent to `seat` and `wheel`, and "check composedOf" messages are sent to `unicycle`, `coplanar`, `coaxial` and `tricycle`.

The `seat` concept-frame is similar, it too is a primitive concept-frame, and when created sends "check joined" to `wheel`, and "check composedOf" to `unicycle`, `bicycle` and `tricycle`.

`Coplanar` is a compound concept-frame and represents two or more wheels that are joined and coplanar. Its existence criteria are that the wheel passed to it is joined and coplanar with any other wheel. Upon creation it sends "check joined" to `seat` and "check composedOf" to `bicycle`, provided there are precisely two wheels.

`Coaxial` is similar to `coplanar`, as it represents two wheels that are joined and whose axes are collinear. Upon creation, however, this concept-frame sends "check composedOf" to `tricycle`.

`Unicycle` is created if a wheel is joined to, and under, a seat. It is deleted if another wheel is added to the system. Upon creation (deletion) this sends "check composedOf" ("negCheck") to `circus`.

`Bicycle` is created if a coplanar has two wheels and is joined to, and under, a seat. It is deleted if the number of wheels exceeds two. Upon creation (deletion) it sends "check composedOf" ("negCheck") to `street`.

`Tricycle` is created if a coaxial is joined and under a seat, and is joined to another wheel, and is deleted if the number of wheels exceeds three. Upon creation (deletion) it sends "check composedOf" ("negCheck") to `creche`.

The remaining concept-frames are the high level interpretations of the system, and correspond to "what the picture is about", ie, if a unicycle is found then a concept-instance of `circus` is created, similarly a tricycle results in a `creche` and a bicycle a `street`. The higher-level concept-frames send inquiries through the network and generate high level descriptions of the scene.

5.4 Trial Runs of Wheels Scenario

In the trial run described below, a unicycle and circus was generated initially. Then a second wheel was added. It can be seen how the system dismantled the circus hypothesis and went on to generate the bicycle/street-scene hypothesis.

Firstly, the system was initialized with:

```
msg(seat,create(id(seat,1),[])),
msg(wheel,create(id(wheel,1),[]))
```

and the database with:

```
below wheel 1 seat 1
above seat 1 wheel 1
joined wheel 1 seat 1
joined seat 1 wheel 1
```

Since many messages passed through the switchboard only a selected output of the concept-frames are shown below:

```
wheel:
        *created id(wheel,1)
        *check relation joined from seat 1
        *update id(wheel,1) with [reln(joined,id(seat,1))]
        *enquiry from id(wheel,1) re joined
        *enquiry from id(seat,1) re joined
        *update id(wheel,1) with [reln(partOf,id(unicycle,4448))]
        *update id(wheel,1) with [reln(below,id(seat,1))]

seat:
        *created id(seat,1)
        *check relation joined from wheel 1
        *update id(seat,1) with [reln(joined,id(wheel,1))]
        *enquiry from id(wheel,1) re joined
        *enquiry to id(seat,1) re reln(above,id(wheel,1))
        *update id(seat,1) with [reln(partOf,id(unicycle,4448))]

unicycle:
        *id(unicycle,4448) created from wheel 1
        *check existing [id(unicycle,4448)] from seat 1

circus:
        *circus 1 created from unicycle 4448
```

At this stage the system had deduced a unicycle from the single seat and wheel, and from the unicycle deduced a circus scene. However, when another wheel joined to the first was introduced, things became more complicated, as shown below:

```
message to wheel:
        msg(wheel,create(id(wheel,2),[]))
```

together with the spatial relations:

```
joined wheel 1 wheel 2
coplanar wheel 1 wheel 2
joined wheel 2 wheel 1
coplanar wheel 2 wheel 1
below wheel 2 seat 1
above seat 1 wheel 2
joined wheel 2 seat 1
joined seat 1 wheel 2
```

the system went on to produce changes as reflected by the output of the following concept-frames:

```
wheel:
        *created id(wheel,2)
        *update id(wheel,2) with [reln(joined,id(seat,1))]
        *enquiry from id(wheel,2) re joined
        *enquiry to id(wheel,1) re reln(coaxial,id(wheel,2))
        *enquiry to id(wheel,1) re reln(coplanar,id(wheel,2))
        *enquiry from id(wheel,2) re joined
        *enquiry from id(seat,1) re joined
        *update id(wheel,2) with [reln(below,id(seat,1))]
        *update id(wheel,2) with [reln(partOf,id(coplanar,4449))]
        *update id(wheel,1) with [reln(partOf,id(coplanar,4449))]
        *update id(wheel,2) with [reln(coplanar,id(wheel,1))]
        *id(unicycle,4448) removed from all relations

coplanar:
        *check relation joined from wheel 1 not found
        *check relation joined from seat 1 not found
        *enquiry from id(seat,1) re joined
        *check relation joined from wheel 2 not found
        *id(coplanar,4449) created from wheel 2
        *update id(coplanar,4449) with [reln(joined,id(seat,1))]
        *update id(coplanar,4449) with [reln(partOf,id(bicycle,4450))]
        *update id(coplanar,4449) with [reln(below,id(seat,1)),
                reln(below,id(seat,1))]

unicycle:
        *deleting instances id(unicycle,4448) composed of id(wheel,1)

circus:
        *deleting instances composed of id(unicycle,4448)

bicycle:
        *id(bicycle,4450) created from coplanar 4449

street:
        *street 1 created from bicycle 4450
```

It can be seen how the unicycle found that the new wheel was joined to its component wheel, which satisfied its delete criterion, thus it sent off `negCheck` messages to its associates. Meanwhile, the new wheel formed the `coplanar` compound with the original wheel, consequently the bicycle concept-frame created an instance based on the coplanar being joined, and under, the seat. Note how, like the `stick_set` in the previous scenario, this is an instance of a compound inheriting spatial relations from its components, in this case the

coplanar inherited the wheels' relations with the seat.

5.5 Summary

This chapter described two experimental implementations of the SOO-PIN concept, the cutlery scenario and the wheels scenario. The first experiment demonstrated, as expected, that the network-of-frames was able to come to the correct conclusions for a simple example. The concept of mixed top-down and bottom-up control, and concurrent execution of concept-frames was shown, and it also demonstrated the "spatial database" [1] being updated with new relationships derived for deduced compound objects.

The second experiment, the wheels scenario, showed a more complex network-of-frames which embodied an extra message, `negCheck`, to deal with deleted objects.

Based on the same architecture for high-level interpretation, in the following chapter we will discuss a more complex network for real scene interpretation.

[1] in this early implementation the spatial database was actually a list of spatial predicates.

Chapter 6

Interpretation of Traffic Scenes

In order to test the capabilities of the SOO-PIN concept, the system is used to interpret real images of outdoor scenes, in a context which is rich enough to require an interesting network-of-frames, and yet constrained enough to be tractable. The context chosen is that of traffic scenes. Here the images are aerial views of city intersections (see Figure 6.1), taken by video. The low-level processing is carried out to extract relevant objects (ie, cars) (see Section 6.5). These objects are made available to the network in the form of a spatial database and initiating messages passed to the low-level (primitive) concept-frames. The traffic scenario was implemented in stages. Initially it was implemented to run on single images, the low level data consisting of labeled cars together with their centroid and major axis coordinates. Later, the traffic scenario was extended to find and utilize the velocities of cars, and the SOO-PIN system was extended to handle uncertainty. This chapter deals with the initial traffic interpretation system.

The task of the network is to interpret vehicle activities (ie, vehicle A is turning right from the west) and to produce analyses of the scene of interest to, for instance, highway engineers and traffic light controllers. For example, the ability to analyze image data for the following queries:

- whether the car is on the wrong side of road or intersection,
- when a car should give way to another. For instance:

Figure 6.1: *Typical traffic scene processed by the SOO-PIN system*

- ◇ give way to right at intersection,
- ◇ give way to oncoming when turning right,
- ◇ at T-intersections cars in ending road give way to those on through road,
- ◇ at traffic lights or give way signs,

- traffic jams (ie, give-way deadlocks).

These concepts are incorporated in the network-of-frames shown in Figure 6.2, which is discussed in more detail below.

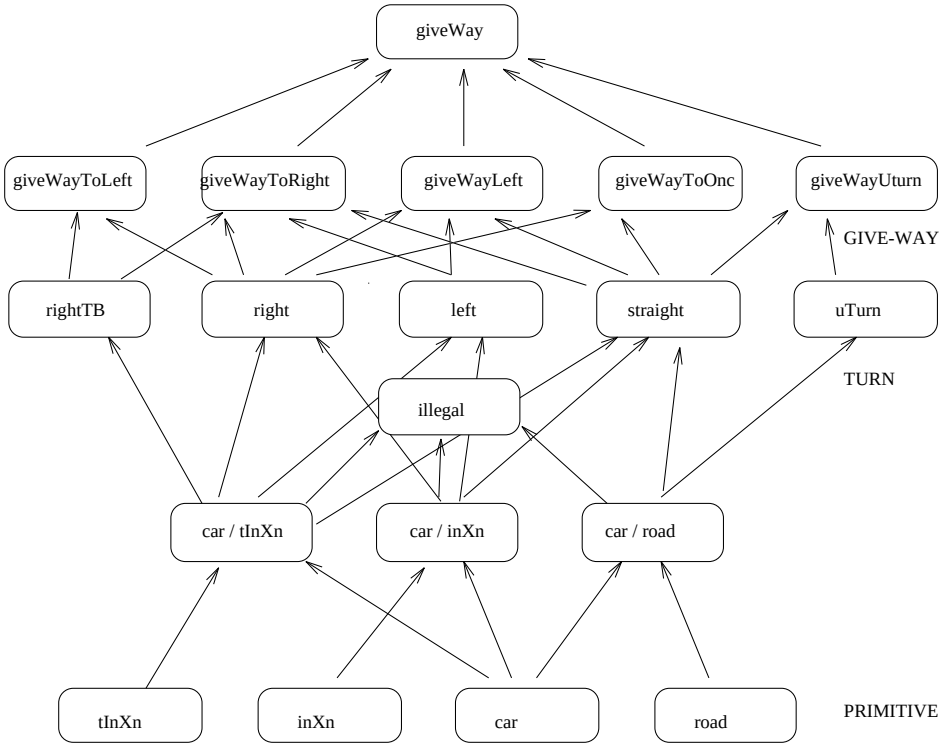

Figure 6.2: *The traffic scenario network-of-frames. The arrows refer to check or create messages, inquiry and update messages are not shown.* `inXn` *refers to "intersection",* `tInXn` *to "T-intersection",* `rightTB` *to a right turn into the through-road of a T-intersection,* `carInXn` *to the concept of a car in an intersection,* `carTInXn` *refers to a car in a T-intersection, and* `carRoad` *to a car in a road.*

6.1 Primitive Concept-Frames

The concept-frames `car`, `inXn`, `tInXn` and `road` correspond to the objects either found by low level processing (see Section 6.5), or given (ie, intersection, T-intersection and road are constant and input manually). These concept-frames simply send `check` messages higher in the network, store information about the concept-instances, and respond to queries about them (in much the same way as the low-level concept-frames described earlier).

6.2 Turn Concept-Frames

On this level, `carInXn`, `carTInXn` and `carRoad` determine the containment of cars in road structures (roads and intersections), ie, "car A is in intersection B". Upon receiving a message from a car or road structure concept-frame, these concept-frames activate a procedural routine written in the C language that read the spatial database and determine whether any of their instances contained the car. This is done because the high-level interactions of a car are dependent on the road structure.

When concept-instances of these concept-frames are created, a call is made to a C routine, `turn.c`, to determine what the car is doing in the road structure. Because images of intersections are, in general, not taken from directly above, this routine first uses an affine transformation that maps the intersection coordinates onto a square to normalize the car positions and heading angles (using transformations derived from [69]). This transformation is also useful because not all intersections are square, and so require to be normalized. The various car activities, ie, right turn, left turn and straight from the north, south, east and west, are unambiguously defined by regions within a 3D product space of the normalized car positions and heading angles (see Figure 6.3). This is analogous to the "spatio-temporal" buffer of Mohnhaupt and Neumann [48] in which traffic events such as turning or overtaking are represented as subsets of the "4D phase-space" of position, velocity direction and speed. The "phase-space" used in SOO-PIN is of lower dimensionality, being simply position and heading. The turn activity found by `turn.c` is sent

in a `create` message to the appropriate concept-frame, ie, `right`, `rightTB` (turning right into the through road of a T-intersection), `left`, `straight`, `uTurn` and `illegal`.

Note that heading angle, determined from the major axis of the car, contains no information about which end of the car is the front. This is deduced from within `turn.c` by assuming that cars are not heading in an illegal direction on the road. This problem of determining heading angle is dealt with directly by finding car velocities (see Section 8.1).

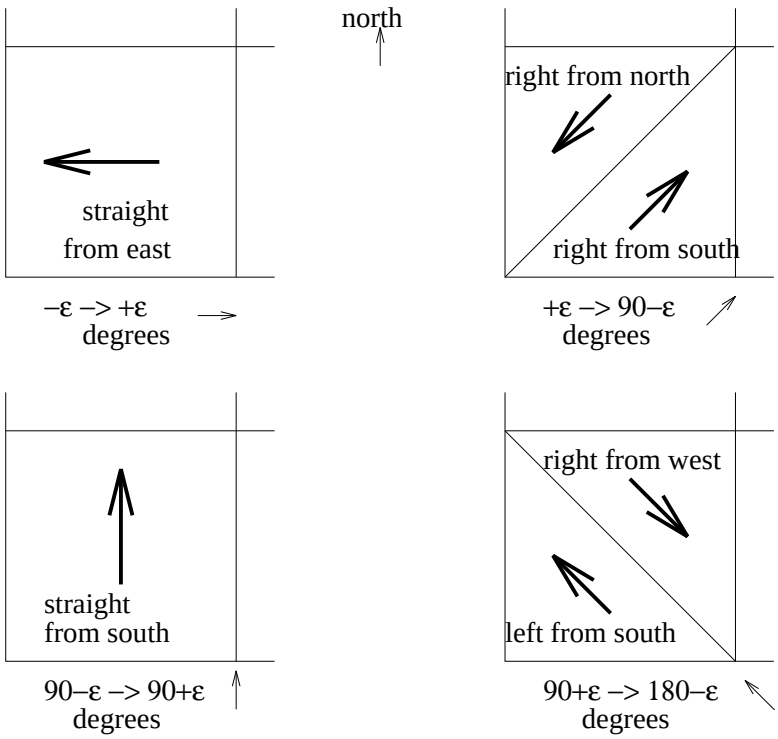

Figure 6.3: *Diagram showing how* `turn.c` *determines car activity from car position and angle. Given that the car is in the south-west corner of the intersection, for the angle range below each square, the system returns the activity as shown within the regions. The other 3 corners of the intersection work similarly.* ε *is the maximum deviation from straight-ahead that is accepted as straight, 6 degrees is used in the system.*

Note that `turn` concept-frames have no existence criteria and so instances are created by fiat. However, these are still useful concept-frames as they are

a repository of information about what the various cars are doing, and can be accessed by the usual inquiry messages from other concept-frames.

6.3 Give-Way Concept-Frames

When turn concept-instances are created, they send `check` messages to the appropriate give-way concept-frames. These concept-frames check the context of the car given in the message to see if any other vehicle is in a give-way relationship with it. For instance, if the `right` concept-frame sends a check message to the `giveWayToOnc` (give way to oncoming concept-frame, this first determines the car's heading, then checks the `straight` concept-frame for any cars coming from the opposite direction – both in the intersection and the adjoining road. If such a car exists, and the cars are close enough, and the traffic lights do not override the give-way relationship, then a `giveWayToOnc` concept-instance is created.

Upon a give-way concept-instance creation, a `create` message is sent to the `deadlock` concept-frame, together with the identities of the two cars involved. This concept-frame checks for cycles in give-way chains (for instance, if car A gives way to car B, and car B gives way to car C, and car C gives way to car A, then nobody can move), and if such a cycle is found reports a legal deadlock or traffic jam.

The high-level interpretation of the network as a whole is generated by `deadlock` and the `give-way` concept-frames and sent to a special output file.

6.4 Trial Runs using XFIG

Initially, the system was run on diagrams of intersections generated by XFIG, a Unix drawing tool. This involved no low-level vision, just reading the XFIG data file to find the road, intersection and car token coordinates. After this point, the system ran in the same way as the full image system described below. Figure 6.4 shows a simple situation in an intersection.

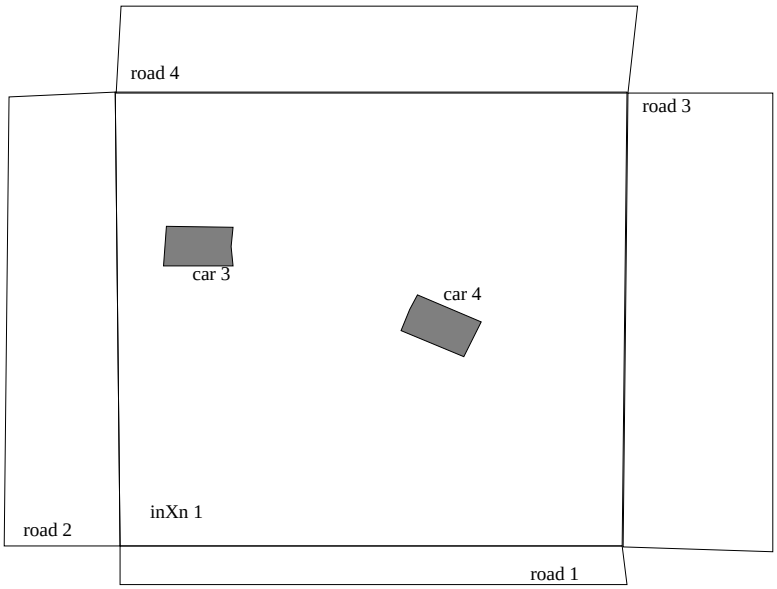

Figure 6.4: *Diagram showing a schematic intersection used for testing SOO-PIN. Roads and intersections are shown as line drawn quadrilaterals, and cars as grey filled rectangles. Each object is labeled with an identifier used in the interpretation system. This scene shows a simple give-way situation.*

Shown below is output generated by selected concept-frames from the run based on Figure 6.4.

```
output from car:
        *created id(car,3)
        *created id(car,4)
        *check relation in from inXn 1
        *check relation in from road 3 not found
        *inquiry from id(road,3) re relation in
        *check relation in from road 1 not found
        *check relation in from road 2 not found
        *check relation in from road 4 not found
        *update id(car,3) with [reln(in,id(inXn,1))]
        *update id(car,4) with [reln(in,id(inXn,1))]
        *inquiry from id(road,1) re relation in
        *inquiry from id(road,2) re relation in
        *inquiry from id(road,4) re relation in
```

After the two cars were created, together with the roads and intersection,

they began by sending check messages exploring spatial and compositional relations to the next level concept-frames, which can be seen in the check messages above. Success is shown in the update messages where the two cars were found to be in the intersection. The action then moved on to higher concept-frames, of which the output from `right` is shown below.

```
output from right:
        *inquiry regarding reln(composedOf,id(inXn,1))
        *inquiry regarding desc(from(south))
        *created id(right,4472)
        *inquiry to id(right,4472) re desc(from(any))
        *inquiry regarding reln(composedOf,id(inXn,1))
        *inquiry regarding desc(from(east))
        *inquiry to id(right,4472) re reln(composedOf,id(car,any))
        *update id(right,4472) with [reln(partOf,id(giveWayToOnc,4473))]
        *inquiry to id(right,4472) re reln(composedOf,id(inXn,any))
        *inquiry regarding desc(from(north))
        *inquiry to id(right,4472) re desc(from(any))
        *inquiry to id(right,4472) re reln(composedOf,id(inXn,any))
        *inquiry to id(right,4472) re desc(from(any))
        *inquiry to id(right,4472) re reln(composedOf,id(car,any))
        *inquiry regarding reln(composedOf,id(inXn,1))
```

After a couple of exploratory inquiries, one concept-instance of a right-turner was created, as was a `straight`. These two objects caused the various give-way concept-frames to begin generating inquiry messages, of which some can be seen above. `giveWayToOnc` was successful, as can be seen from the update message above.

```
output from giveWayToOnc:
        *check relation composedOf from id(straight,4470)
        *adding inst 4473 composed of id(straight,4470)
                id(right,4472) id(car,3) id(car,4)
        *check relation composedOf from id(right,4472)
```

`giveWayToOnc` was able to establish from the existence of `id(straight,4470)` that there was a right-turner coming from the opposite direction, and which was in range of it. Thus it created a new concept-instance, and generated the following interpretation:

```
*Give Way to oncoming:  id(car,4) turning right from east
        gives way to id(car,3) from west
```

Thus, briefly, it can be seen how the system was able to deduce the story behind the picture by a process of independent agents each working on their own concepts.

Below some more complex examples of SOO-PIN operating on XFIG diagrams of intersections are given, but because of the higher cardinality, it is too space-consuming to show all the messages that move around the system, so just the final interpretations are shown.

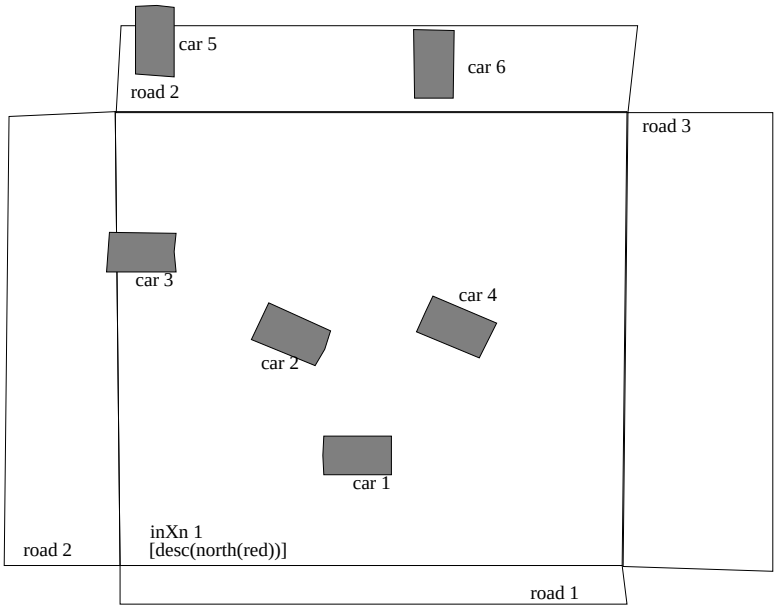

Figure 6.5: *Diagram showing a schematic intersection used for testing SOO-PIN. Roads and intersections are shown as line drawn quadrilaterals, and cars as grey filled rectangles. Each object is labeled with an identifier used in the interpretation system, the intersection is also labeled with the traffic light status. This scene shows a give-way situation modified by traffic lights.*

The output from Figure 6.5 is shown below:

```
*Give Way to oncoming:  id(car,2) turning right from west
        gives way to id(car,1) from east
*Give Way to oncoming:  id(car,4) turning right from east
        gives way to id(car,3) from west
*Give Way to Right overridden by Traffic Sign red :  id(car,6)
        from north gives way to id(car,4) from east
```

It can be seen how the two right-turners have given way to the two straight-throughers, and how the `giveWayToRt` concept-frame checked with the traffic light status of the intersection to modify its report on the give-way situation between `id(car,6)` and `id(car,4)`.

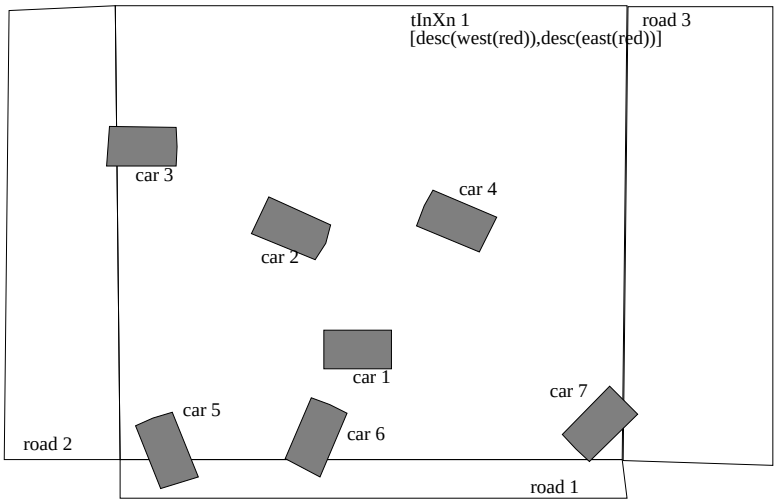

Figure 6.6: *Diagram showing a schematic T-intersection used for testing SOO-PIN. Roads and intersections are shown as line drawn quadrilaterals, and cars as grey filled rectangles. The through road is east-west. Each object is labeled with an identifier used in the interpretation system, the intersection is also labeled with the traffic light status. This scene shows a rather complex and unlikely configuration intended to demonstrate the system.*

Figure 6.6 shows a rather complex T-intersection in order to test the network with many objects and interactions, and also shows the slightly modified road rules that pertain in a T-intersection. The output is shown below:

```
*Illegal:  car id(car,4) from east is on the wrong side
        of T-intersection id(tInXn,1)
*Give Way to Left (T-inXn) overridden by Traffic Sign red :
        id(car,3) from west gives way to id(car,6) from south
*Give Way to oncoming:  id(car,2) turning right from west
        gives way to id(car,1) from east
*Give Way overridden by Traffic Sign red :  id(car,1) from
        east gives way to id(car,5) from south
*Give Way to Right overridden by Traffic Sign red :
        id(car,1) from east gives way to id(car,6) from south
*Give Way to left-turner:  id(car,2) turning right
        from west gives way to id(car,7) from east
```

The first novelty is that T-intersections generate an "illegal" report, as cars should not appear as if they were doing a right or left turn into the blank side of such an intersection. The other novelty (at least on Australian roads) is that cars on the through road have priority over cars on the ending road, and thus `id(car,6)` would normally have given way to `id(car,3)`, except for the case where traffic lights are present.

Figure 6.7 shows a section of road, with two cars `id(car,7)` and `id(car,8)` traveling straight in opposite directions, and the rest of the cars doing what the system interpreted as U-turns, ie, the cars were not parallel to the road. The system interpretation follows:

```
*Give Way to Oncoming:  id(car,6) U-turning gives way
        to id(car,8) going straight
*Give Way to Oncoming:  id(car,1) U-turning gives way
        to id(car,8) going straight
*Give Way to Oncoming:  id(car,1) U-turning gives way
        to id(car,7) going straight
*Give Way to Oncoming:  id(car,14) U-turning gives way
        to id(car,7) going straight
*Give Way to Oncoming:  id(car,14) U-turning gives way
        to id(car,8) going straight
*Give Way to Oncoming:  id(car,2) U-turning gives way
        to id(car,8) going straight
*Give Way to Oncoming:  id(car,2) U-turning gives way
        to id(car,7) going straight
*Give Way to Oncoming:  id(car,3) U-turning gives way
        to id(car,7) going straight
```

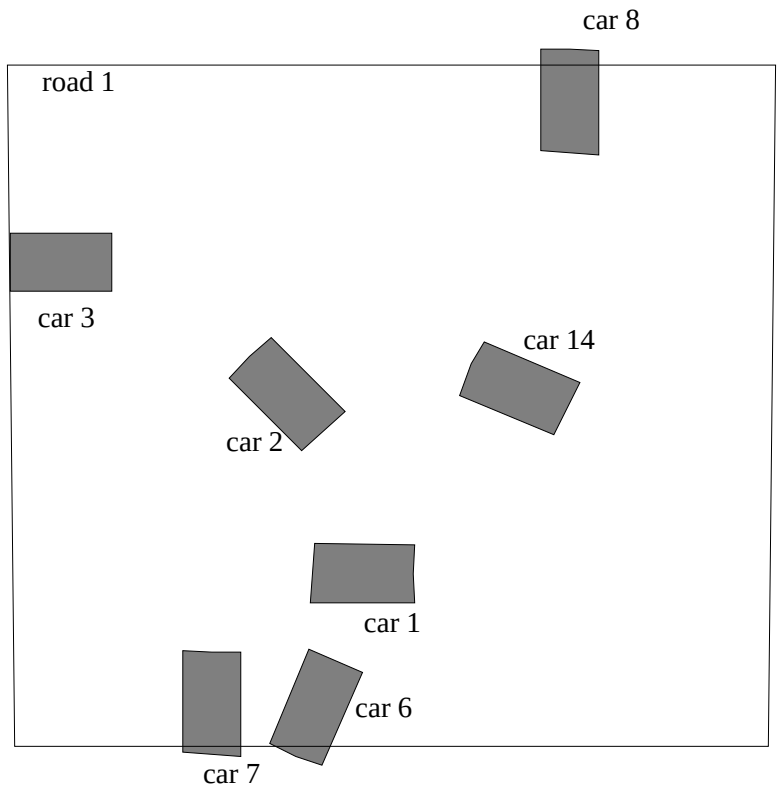

Figure 6.7: *Diagram showing a section of road used for testing SOO-PIN. The road is shown as a line drawn quadrilateral oriented north-south, and cars as grey filled rectangles. Each object is labeled with an identifier used in the interpretation system. This scene shows a rather complex configuration of cars in various orientations over a road.*

6.5 Low-Level Processing

For the actual intersection data, the system ran on images recorded on videotape from a camera mounted above city intersections and the images were digitized using an Abekas Digital Video system (in 720x576, 24 bit RGB format), see Figure 6.8(a). The subsequent low-level processing was performed using the IPRS library of images processing routines [26] used in the Computer Vision laboratory at Melbourne University. Background subtraction was used for the detection of vehicles by subtracting an image of the corresponding empty intersection (Figure 6.8(b)) from target views, that is, pixels with less than a threshold difference in RGB (red-green-blue) space between the empty and target images were set to zero, the rest to one, thus creating binary difference images (Figure 6.8(c)). These were then median filtered (using a 3x3 window) [32] to remove noise and smooth edges. The resulting regions were then separated into connected labeled regions using an "equivalence tables" technique, and 4-connectedness (ie, the label of each pixel in turn was compared with the labeling of it's neighbours to the left and below to determine whether to merge the labels or not), and then thresholded to remove regions too small to be vehicles. Assuming that each residual region corresponded to candidate vehicles [1], region (vehicle) attributes were computed corresponding to centroids and best-fitting major axes (orientations) (see Figure 6.8(d)). These values, together with the car identities and frame numbers, for each image, were stored in a "spatial database" made available to SOO-PIN. The (constant) intersection and road coordinates were also stored in this database.

Note how in the example given in Figure 6.8, the dark car toward the top right is lost before final recognition. This is inherent in the technique used, a more sophisticated low level processing technique could have been used to pick up these cases, but for the purpose of this study the current technique was sufficient for "proof-of-concept" of the system.

[1] As will be seen, one benefit of the symbolic approach is that inappropriate segments can be checked for their consistency, updated or deleted by the symbolic analyses

Another inadequacy of the technique used is that it required manual selection of the empty intersection, which, in a real world system, would have to be done every few minutes to cope with changing lighting conditions. A technique that could have been used to generate empty intersection images automatically is to use a form of median filtering over time, ie, from continuous video of the scene, save 5 images from the last, say, 10 minutes (long enough for the traffic lights to cycle twice). Then for each output image pixel choose the median of the corresponding pixels in the 5 saved images. Since most parts of the scene are free of cars most of the time, this yields an image of an empty intersection.

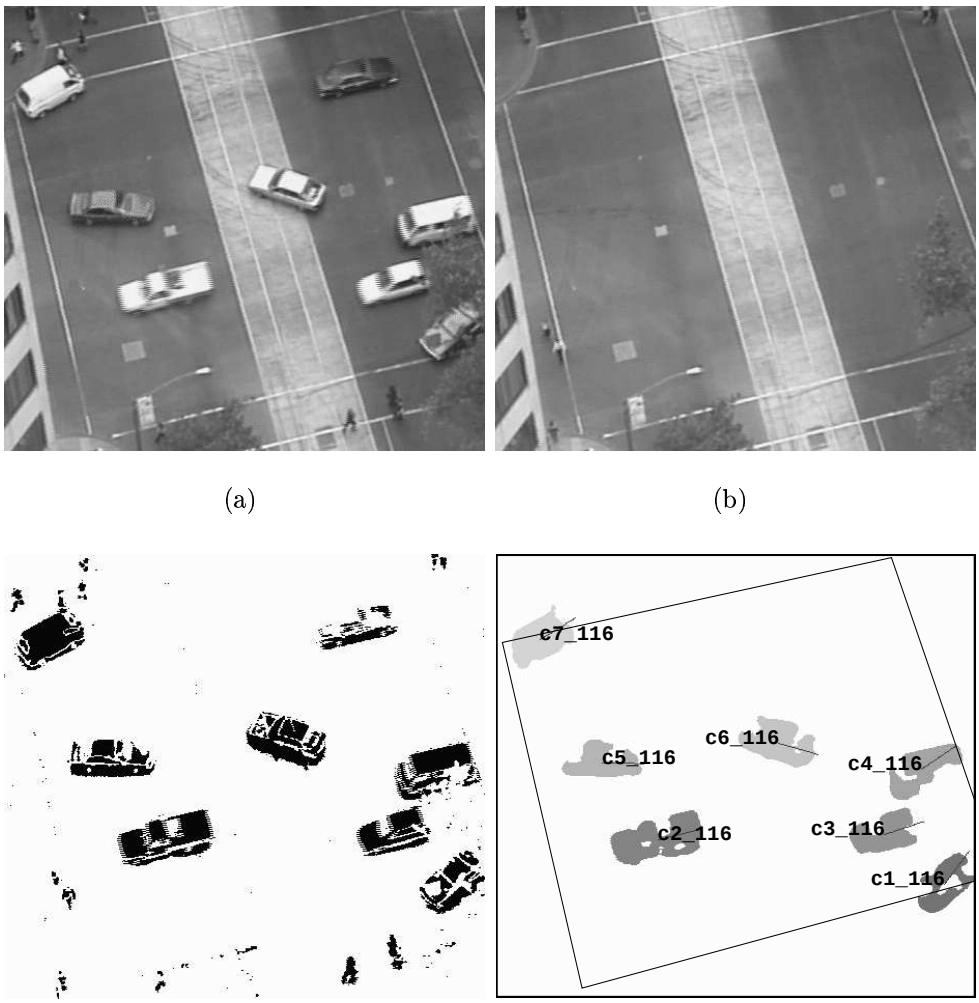

Figure 6.8: *Processing steps of real image: (a) Original image of intersection (b) Empty intersection (c) Raw differenced binary image (d) Median filtered, size filtered and labeled image, the lines on the cars are the major axes, the skew rectangle is the intersection boundary (determined manually).*

6.6 Trial Runs on Real Images

Using the image shown in Figure 6.8 as input into the traffic network described above yields the following interpretation:

```
*Give Way to oncoming:  id(car,c5_116) turning right from west gives
        way to id(car,c3_116) from east.

*Give Way to left-turner:  id(car,c5_116) turning right from west
        gives way to id(car,c4_116) from east.

*Give Way to oncoming:  id(car,c5_116) turning right from west gives
        way to id(car,c2_116) from east.

*Give Way to left-turner:  id(car,c5_116) turning right from west
        gives way to id(car,c1_116) from east.

*Give Way to left-turner:  id(car,c6_116) turning right from east
        gives way to id(car,c7_116) from west.
```

It can be seen how the system has interpreted `id(car,c4_116)` as a left turner, when clearly it was going straight ahead. The trouble was that the car was partly obscured by a tree, and thus the major axis of the car was skewed. This resulted in the left turn interpretation. Later in this study, this problem is rectified by the detection of car velocity.

The next example used an image from another intersection (see Figure 6.9), taken from a much lower camera resulting in an oblique view. This image demonstrated the usefulness of normalizing the image with an affine transformation onto a square. Potential problems with such normalization are that, firstly, it will not work if the images of the cars overlap due to perspective. Secondly, it has the effect of moving the centroid of the cars, when projected onto the plane of the intersection, further away from the camera. Both of these effects limit the amount of obliqueness, and thus the minimum height of the camera, that is acceptable.

The interpretation generated from Figure 6.9 is as follows:

```
*Give Way to left-turner:  id(car,c5_150) turning right from west
        gives way to id(car,c1_150) from east.
```

(a) (b)

Figure 6.9: *(a) Original image of intersection (b) Median filtered, size filtered and labeled image, the lines on the cars are the major axes, the skew rectangle is the intersection boundary (input manually).*

This is correct, as judged by eye, but it is not the full story. It can be seen that car `id(car,c4_150)` was also turning right, but because its major axis was roughly aligned with the road, it was labeled as a car traveling straight ahead.

6.7 Traffic Trial Summary

In this chapter we have demonstrated the SOO-PIN concept implemented in a traffic scenario. This has been a useful exercise as this involves quite a complex network of frames to implement. Initially, it was implemented to run from XFIG diagrams, which demonstrated the capability of the system over a broad range of input situations. It was then shown how the system runs on real images, which, while more complex to process initially, results in simpler, more restricted traffic situations (in general, when videotaping traffic, one very seldom, if ever, sees an interesting traffic law violation). It was found that the system performed quite satisfactorily on these images,

producing interpretations that the human observer would make.

Some limitations of the system resulted from using the orientation (major axis) of the cars, with their position in the intersection, to judge their heading and activity. This resulted in some inaccuracy in judging the direction the car was heading with consequent misinterpretation of the car's activity. This could be partly rectified if the velocity of the vehicles could be determined, which will be dealt with in a later chapter.

Another potential problem is the number of context dependent numerical thresholds involved in the implementation, for instance, the allowable angle that a car can be offset from straight ahead before it is judged to be doing another activity. This calls for some kind of measure of confidence in the judgments made within the various concept-frames to be passed around the network in the messages. This is dealt with in the next chapter.

Chapter 7

Uncertainty

7.1 Introduction

In vision systems, uncertainty comes from a number of sources. For instance, low-level object detection can return belief values deriving from identifying a given segment as a certain object, spatial predicate instantiation can return uncertainty from mapping continuous variables onto symbols, and high-level systems can involve uncertainty in deriving one proposition from others through such processes as induction. In SOO-PIN, there is a need to cope with uncertainty, at least to the extent of tracking it from predicate to predicate so that the final interpretations reflect the degree of uncertainty involved with their derivations.

There are two broad approaches to using uncertainty calculi in networks, the Bayesian approach and the Dempster-Shafer (DS) approach. The first is described by Pearl [52]. He shows how a network of nodes (in this case Markov trees) propagates belief. In this approach, each node represents a variable whose possible values are assigned probabilities (a frame of discernment), connected by links with the conditional probabilities expressed as a matrix. He distinguishes evidential support from causal support, which, in general, convey belief in opposite directions. In acyclic networks it is important to make that distinction as it prevents a change in probability at one node feeding back on itself and giving itself credence. One application of this approach

is that of Huang et al[41] who use a Bayesian belief network and inference engine (HUGIN[3]) in sequences of highway traffic scenes to produce high-level concepts like "lane_change" and "stalled". A tool using Bayesian networks based on Pearl's ideas is BaRT [40] which is used to classify ship images.

Bogler [11] has pointed out that in the case of data fusion and object recognition, the Bayesian approach has limitations, namely:

- sensors provide information at varying levels of abstraction, and, for instance, if a sensor says "the target is of type A, B or C" (where A, B and C are possible targets), with a certain probability, the Bayesian approach forces the system to divide the probability equally between the three types.

- the sensors are required to have a complete and accurate knowledge of both the prior probability distribution and the conditional probability matrices. If they do not have this data, the Bayesian approach forces them to guess.

- if sensors give contradictory readings, the Bayesian approach has no formal means of dealing with this. The usual solution is to distribute various likelihoods where they are unknown.

Bogler shows how the Dempster-Shafer approach solves these difficulties, and how this approach is used for target recognition.

Lowrance et al [44] in a report on evidential reasoning systems for the US navy, like Pearl, deals with Markov trees that convey belief bidirectionally, in this case from a Dempster-Shafer perspective (DS is explained further in Section 7.2 below). In Lowrance's system, each node is associated with a "frame of discernment" where belief is expressed in the form of a mass distribution over the set of subsets of the frame. Links are in the form of "compatibility mappings" between the sets of subsets of the respective frames of discernment, and thus play the role of Pearl's conditional probability matrices. A pair of nodes will have a compatibility mapping in each direction, and they are, like Pearl, careful to avoid positive feedback (which would result in propositions supporting themselves). A number of systems have been developed using the

Lowrance approach. For instance, Garvey [31] describes a helicopter route planning system in which each pixel of a topographic map is attached to a network of nodes representing such things as vegetation cover, terrain type, visibility and overall danger.

Wesley [67] presents a system for labeling regions of 2D monochromatic scenes using Dempster-Shafer evidential reasoning – as described by Lowrance. He limits the frames of discernment to the interpretation of given regions, and discusses independence of evidence, noting the differences between the DS and Bayesian interpretation. Wesley argues that it is necessary to develop a formal model to account for dependencies between knowledge sources, – a poignant issue which he does not pursue.

Baldwin [6] describes a system (FRIL) which uses the basic Dempster-Shafer formalism in a logic-programming inference system. This system is to be distinguished from the approaches of Pearl and Lowrance in that it does not deal with combining evidence within a frame of discernment, but rather with combining belief between independent propositions. Later (Section 7.4.1) we will show how this approach is particularly useful for vision.

The Dempster-Shafer uncertainty calculus is discussed below in more detail, along with some verifications of Baldwin's results which were not shown in his original paper [6]. Following this, some criticisms and caveats regarding Baldwin are dealt with, and the chapter finishes by showing how uncertainty is implemented into the SOO-PIN system.

7.2 Dempster-Shafer Theory

Like Bayesian uncertainty calculus, Dempster-Shafer uncertainty calculus starts from a set of exhaustive and mutually exclusive propositions in a "frame of discernment". Again, as in Bayes, there is an assignment of weights, but in this case the weight is assigned to *sets* of these propositions, including the set of all the propositions in the frame of discernment. This formulation allows for an expression of degrees of ignorance. For instance, if the frame of discernment is a set of possible burglars, then if the burglar is found to be

male, then a weight can be associated with the set of all male burglars. In Bayes, without prior knowledge, one would have to give an equal increment of weight to each *individual* male burglar, thus implicitly providing more information than we actually have. This weight assignment is called a "mass distribution", and obeys the following:

$$\begin{aligned} m : 2^{\Theta} &\mapsto [0,1] \\ \sum_{A_i \subseteq \Theta} m(A_i) &= 1 \\ m(\emptyset) &= 0 \end{aligned} \tag{7.1}$$

where Θ is the frame of discernment, and 2^{Θ} is the set of all subsets of Θ.

Uncertainty of a proposition P is interpreted via its "support" and "plausibility". The support is given by the sum of all the weights attached to subsets of the proposition, ie:

$$\mathcal{S}(P) = \sum_{A \subseteq P} m(A) \tag{7.2}$$

The plausibility of a proposition P is the difference between certainty and the support of $\neg P$, ie:

$$\mathcal{P}(P) = 1 - \mathcal{S}(\neg P) \tag{7.3}$$

The support and plausibility are referred to below as the "belief pair" or "belief interval" of a proposition.

Since the sum of all weights is unity, then $\mathcal{S}(P) + \mathcal{P}(P) \leq 1$ (Lowrance et al [44, p7]). If in a degenerate case each and only singleton propositions are given weight by the mass assignment, then the belief pair $[\mathcal{S}, \mathcal{P}]$ has the one value $\mathcal{S} = \mathcal{P}$ and is equal to the Bayes probability. In this way Dempster-Shafer uncertainty can be viewed as a superset of Bayesian probability. Pearl, however, views DS as philosophically quite distinct to Bayes, namely, that the DS belief values deal with the probability of the *provability* of propositions, whereas Bayes deals with the probability of the *truth* of them. In Pearl's view, a model for DS is provided by thinking of each frame of discernment as having a timer that assigns "truth" to each proposition for a fraction of

the time corresponding to the value of the mass of that proposition. One then finds the logical outcome of the assignments for each time instance, and ascribes mass to the outcomes by the proportion of the time they are true. This model produces the required DS calculus as described below.

7.2.1 Combining Evidence within a Frame of Discernment

If we have evidence for a proposition P in the form of a mass distribution m_1 and other independent evidence in the form m_2, then Dempster's rule of combination [24] allows us to calculate the combined mass distribution m_3 reflecting the combined evidence for P:

$$\begin{aligned} m_3(P) &= m_1 \oplus m_2(P) \\ &= \frac{1}{1-\kappa} \sum_{A_i \cap A_j = P} m_1(A_i) m_2(A_j) \\ \kappa &= \sum_{A_i \cap A_j = 0} m_1(A_i) m_2(A_j) \end{aligned} \tag{7.4}$$

The factor κ is referred to as the *conflict* between the evidence represented by m_1 and m_2 (Lowrance et al [44, p8]), ie, if there is conflict, there are pairs of sets with mass assigned by m_1 and m_2 respectively with no intersection.

This rule allows us to find the combined weight of evidence for an hypothesis from various sources. For instance, if one source of evidence suggests that a burglar is male with a mass distribution giving weight to the subset of males (of, say, 0.8), and another source gives weight to the subset of redheads (of, say, 0.7), then we can use Dempster's rule to calculate the combined mass and possibly suggest the culprit. Assuming the males are Alex, Bill and Chris, and the redheads are Chris and Diane, then using Equation 7.4, we need to calculate m_3 given the two distributions from the evidence, namely m_1 giving weight 0.8 to males and 0.2 to Φ (all the suspects), and m_2 giving weight 0.7 to redheads and 0.3 to Φ. From Table 7.1 it can be seen that the only non-empty intersecting subsets of the two distributions are *Chris*, *male*, *redhead*

$\cap$	*male*	Φ
redhead	*Chris*	*redhead*
Φ	*male*	Φ

Table 7.1: *Evidence combinations for the burglar example. This table shows the non-empty intersections of subsets assigned non-zero weight by each evidence source. Φ represents all the suspects, male is the set of males, and redhead is the set of redheads.*

and Φ. There are no empty intersecting sets, therefore $\kappa = 0$ and $\frac{1}{1-\kappa} = 1$. The calculation of m_3 is as follows:

$$\begin{aligned}
m_3(Chris) &= \frac{1}{1-\kappa} * m_1(male) * m_2(redhead) = 0.8 * 0.7 = 0.56 \\
m_3(redhead) &= \frac{1}{1-\kappa} * m_1(\Phi) * m_2(redhead) = 0.2 * 0.7 = 0.14 \\
m_3(male) &= \frac{1}{1-\kappa} * m_1(male) * m_2(\Phi) = 0.8 * 0.3 = 0.24 \\
m_3(\Phi) &= \frac{1}{1-\kappa} * m_1(\Phi) * m_2(\Phi) = 0.2 * 0.3 = 0.06
\end{aligned} \tag{7.5}$$

It can be seen that the highest belief in the combined evidence is for Chris as the culprit, with a weight of 0.56.

7.2.2 Combining Independent Propositions

Dempster-Shafer works well in situations where the frame of discernment is clear, with evidence bearing upon this frame of discernment from independent sources. However, as explained in Section 7.4.1, in Computer Vision, frames of discernment are quite simple, but evidence bears upon independent propositions. Therefore it is necessary to consider Baldwin's [6] work, where he deals with belief assigned to independent propositions. In this section a

result shown by Baldwin in which he combines belief intervals of two propositions in independent frames of discernment is justified. Another reason for exploring this is that not only is his approach compatible with vision, but such combinations fit nicely within logic programming. For instance, a typical Horn clause might be $P \Leftarrow A \wedge B$ (P is proven if both A and B are). To include uncertainty one can use Baldwin's combination of independent propositions rules to compute the uncertainty of P given the uncertainty A and B.

Given two propositions A and B with belief intervals $[\mathcal{S}(A), \mathcal{P}(A)]$ and $[\mathcal{S}(B), \mathcal{P}(B)]$, then Baldwin's combination rules are, firstly, conjunction:

$$\mathcal{S}(A \cap B) = \mathcal{S}(A).\mathcal{S}(B) \tag{7.6}$$

$$\mathcal{P}(A \cap B) = \mathcal{P}(A).\mathcal{P}(B) \tag{7.7}$$

secondly, disjunction:

$$\mathcal{S}(A \cup B) = \mathcal{S}(A) + \mathcal{S}(B) - \mathcal{S}(A).\mathcal{S}(B) \tag{7.8}$$

$$\mathcal{P}(A \cup B) = \mathcal{P}(A) + \mathcal{P}(B) - \mathcal{P}(A).\mathcal{P}(B) \tag{7.9}$$

As can be seen, these rules are very similar to the equivalent Bayesian rules. These rules allow propagation of uncertainty values through a network, each node passing a belief pair in a message to other nodes which can calculate the belief pairs for various logical statements. Of course, as explained in Section 7.3, it is important to ensure that the network contains no dependency loops, since this breaks the precondition for the combination rules.

7.2.2.1 Conjunction Rule

In his paper [6], Baldwin did not derive in detail the conjunction in the combination rule Equation 7.6. This rule is now justified.

If proposition B is in Frame of Discernment Φ and proposition A is in Frame of Discernment Θ, and we have $\mathcal{S}(B)$ and $\mathcal{S}(A)$, then we want to know $\mathcal{S}(A \cap B)$. In order to combine propositions in independent, "orthogonal"

Frames of Discernment we need to work in the cross product frame $\Theta \times \Phi$.

From the definition of the support of a proposition given in Equation 7.2,

$$\mathcal{S}(B) = \sum_{X \subseteq B} m_2(X) \tag{7.10}$$

then

$$\mathcal{S}(B) = m_2(X_1) + m_2(X_2) \cdots m_2(X_n)$$

where $m_2(X_i)$ are non-zero, and m_2 is the mass assignment for the frame Φ. Similarly for A:

$$\mathcal{S}(A) = m_1(Y_1) + m_1(Y_2) \cdots m_1(Y_m)$$

From the same definition,

$$\mathcal{S}(A \cap B) = \sum_{X \subseteq A \cap B} m_3(X)$$

$$= \sum_{X \cap Y \subseteq A \cap B} m_3(X \cap Y)$$

Since the only mass carrying members of $\Theta \times \Phi$ are sets of the form $X \times \Phi \cap Y \times \Theta$ where $X \subseteq \Theta$ and $Y \subseteq \Phi$,

$$\mathcal{S}(A \cap B) = \sum_{i,j} m_3(X_i \times \Phi \cap Y_j \times \Theta)$$

from the enumerated non-zero mass subsets of Θ and Φ given above.

Now from the definition of combined mass assignments given in Equation 7.4,

$$m_3(X_i \times \Phi \cap Y_j \times \Theta) = \frac{1}{1-\kappa} \sum_{V \times \Phi \cap W \times \Theta = X_i \times \Phi \cap Y_j \times \Theta} m_1(V).m_2(W)$$

$$= m_1(X_i).m_2(Y_j)$$

as there is only one pair $V \times \Phi \cap W \times \Theta$ equal to $X_i \times \Phi \cap Y_j \times \Theta$, namely

themselves. The κ is 0 as

$$\kappa = \sum_{V \times \Phi \cap W \times \Theta = \phi} m_1(V).m_2(W) \tag{7.11}$$

and V and W are orthogonal and always have non-zero intersection.

Thus

$$\begin{aligned} \mathcal{S}(A \cap B) &= \sum_{i,j} m_1(X_i).m_2(Y_j) \\ &= (\sum_i m_1(X_i)).(\sum_j m_2(Y_j)) \\ &= \mathcal{S}(A).\mathcal{S}(B) \end{aligned} \tag{7.12}$$

again from the definition of the Belief function.

7.2.2.2 Disjunction Rule

For disjunction (Equation 7.8) we can assume without loss of generality that the mass distribution in Φ is limited to $m_2(B), m_2(\neg B)$ and $m_2(\Phi)$ as the mass of all other subsets can be subsumed under the total frame Φ, similarly for Θ. This means $\mathcal{S}(B) = m_2(B)$, $\mathcal{S}(\neg B) = m_2(\neg B)$ and $\mathcal{S}(\Phi) = m_2(\Phi) = 1 - m_2(B) - m_2(\neg B)$ (this latter because mass distributions must sum to 1). Thus support for B is $m_2(B)$, plausibility of B is $1 - m_2(\neg B)$, and doubt about B is $1 - m_2(B) - m_2(\neg B)$.

To deal with the disjunction $A \cup B$ it will help to use the following Table 7.2. The masses assigned to the sets in the table are calculated from Equation 7.12. These values follow clearly from the conjunction rule above, except for the cases involving Θ or Φ. An example is shown below:

$$\mathcal{S}(A \cap \Phi) = \sum_{X \subseteq A \cap \Phi} m_3(X)$$

which since the only non-zero mass subset is itself

$$= m_3(A \cap \Phi)$$