An architecture for self-adaptation and its application to aerial image understanding.

Paul Robertson (pr@robots.ox.ac.uk)

University of Oxford, Dept. of Engineering Science,
19 Parks Road, Oxford, OX1 3PJ, England, UK

Abstract. Certain problems in which the environment is not well constrained do not lend themselves to a conventional open loop solution. Image understanding is such a problem domain. Image analysis programs are notoriously brittle. By using a self-adaptive approach for these problem domains we may be able to produce more robust and useful behavior. We describe an architecture that uses reflection to provide a mechanism for self-monitoring and self-modification and uses code synthesis as a means of modifying code dynamically in the face of changing assumptions about the environment of the computation.
Keywords: Aerial Image Analysis, Reflection, Code Synthesis, Agent Architecture.

1 Introduction

This paper provides an overview of the GRAVA [10] architecture and the rationale underlying its design.

This project began with the observation that a significant source of problems with interpreting visual scenes comes from our inability to precisely predict the nature of the environment in which the image interpretation programs are expected to operate. Unlike problems like adding up numbers which is unaffected by an outside environment, for visual interpretation problems we cannot be certain what the environment looks like. We believed that instead of having image analysis algorithms being applied "blind" that they should evaluate their performance and where necessary change the running system to accommodate the actual environment that they "observe". The need to reason about the computational nature of the system of which they are a part and to be able to make changes to that system suggested a reflective [3, 8, 11] architecture.

In the spring of 1998 DARPA introduced the term "Self adaptive software" to describe exactly the kind of problem described above. This project was later funded under the DARPA "Automatic Software Composition" (ASC) program.

A reflective architecture provides the mechanisms necessary to support two of the essential problems of self adaptive software – the mechanism for reasoning about the state of the computational system and the mechanism for making changes to it. Over the years there has been much work on reflective systems and the methods for their successful implementation are now well known [2, 4].

Most work on reflection concentrates on how to provide these two mechanisms in the domain of interest so that a programmer may make use of them. The problem of self adaptive software goes a step further. Not only must we have mechanisms that support introspection and modification of the systems semantics, we must also have an implementation of the "programmer" of these mechanisms.

The components of self adaptive software include:

1. The ability to "monitor" the state of the computation. This includes some kind of model against which to compare the computation.
2. The ability to "diagnose" problems.
3. The ability to "make changes" to correct deviations from the desired behavior.

The reason that adaptation is sought is to achieve robustness. The ability to adapt to an environment that is not quite what was expected should enable a program to continue to function where it would otherwise break. Self-adaptation appeals to our intuitive notion that programs are often brittle. The underlying reason why self-adaptation is useful in building robust programs however is that the environment is often – or usually – impossible to model accurately.

When the environment can be modeled exactly it is possible to build programs that are rubust without the need for self-adaptation. It is the lack of absolutes in the world that makes self-adaptation attractive.

1.1 A New Kind of Computation

There are many useful ways of partitioning computation but the dimension that interests us in this paper is the dimension of predictability of the environment. A computation that depends upon a completely determined environment will be referred to as a type-1 computation and a computation that does not depend upon the environment to be completely determined we will refer to as a type-2 computation. Almost all of computer science has focussed on type-1 computations. When we try to accommodate environments that are less well determined the complexity rapidly increases to the point where the programs are difficult to maintain. Programs that perform input/output are examples of where this problem is frequently faced. Because the environment is not well determined a program that has to (say) read in a command string must face the problem of dealing with the full range of possibilities and respond to them appropriately. If we knew that a program would read a well formed command each time it would be easy. Here we restrict the problem by defining any input that does not fit the pattern of a correct input as being an error. Even with this simplification I/O programs are often huge and buggy.

Hypothesis: Most interesting computational problems are type-2.

In the previous century we have largely been concerned with the open loop absolute formulation of computing but it represents a small fraction of the types of computation that we are interested in. Nature provides countless examples of computational systems. They are almost without exception type-2 and they are robust. Now we wish to bring computation into the realm where nature has provided us with so many successful examples and we find that type 1 computation does not scale to the real world where absolutes disappear and guarantees are all but nonexistent.

Programs that have to interpret visual scenes are an extreme example of type-2 computations. The environment cannot be precisely specified and all scenes are legal. We wish to make sense of all visual scenes that the camera can capture.

Until recently we have built the theory of computation around the notion of type-1 computation and developed notions of program correctness around the assumption that the environment can be accurately and completely modeled. We have been able to do this because for the most part computation has occurred within the artificial environment of a computer separated from the outside world except for some input channels whose inputs have been rigorously specified.

While we have seemingly been able to do a lot with type-1 computation it is what we cannot do well that argues for a new theory of computer science that embraces the more broadly general view of computation that includes both type-1 and type-2 computation.

Even the simplest animals exhibit type-2 computations that are extremely robust. By comparison our ability to build robots that explore our unconstrained environment is hampered by a lack of understanding of how to build type-2 computations. Programs that interpret visual scenes suffer for the same reason.

When we have absolutes we can build a system that behaves perfectly. This is conventional programming. This is a rare occurrence in nature.

When success of a computation cannot be guaranteed for any reason we must:

1. Check how well the computation did.
2. Be prepared to take some constructive action if the computation didn't do well.

This formulation of an outline for type-2 computations is similar to a control system. Rather than simply assuming that the computation performs as expected, the result is measured against an expected behavior and when a deviation is detected a corrective force is applied and the computation retried in order to bring the result closer to the expected behavior – the set point. In this formulation type-2 computations consist of type-1 computations encapsulated in a control system. This allows us to benefit from everything that we know about building correct type-1 systems and requires additionally that we have mechanisms for:

1. Knowing the program intent.
2. Measuring how closely the program intent was met.
3. Applying a corrective force to bring the program behavior closer to the program intent.

Self-adaptive software is an attempt to build type-2 computations as systems that apply a corrective force by making changes to the program code. At the simple end of the spectrum of self-adaptive systems this can simply mean adjusting parameters upon which the program operates. At the other end of the spectrum it means re-synthesis of the program code.

The self-adaptive computation loop looks like this:

1. Synthesize a program based on an understanding of:
   (a) What is intended.
   (b) What the environment is like.
   (c) How computational elements are likely to behave.
2. Monitor the behavior of the program measuring deltas.
   (a) Measure how close the program is to achieving its intent.
   (b) Measure what the environment is really like.
   (c) Measure how computational elements are actually behaving.
3. Update our choices for computational elements based on empirical understanding of the system dynamics and re-synthesize (go to step 1 and repeat).

Program intent is described as a specification in some kind of language. A computational element is a module that may be selected to perform some operation called for by the specification. A computational element may have subordinate requirements of its own.

If the program specification is considered to be a theorem, the computational elements can be thought of as axioms or rules of inference (depending upon whether they transform a piece of specification into another or whether they simply satisfy a piece of specification).

The computational elements define a formal system and the goal of program synthesis is to prove that the specification is a theorem of the formal system defined by the collection of computational elements. The proof is actually a selection of the computational elements wired together as an executable program and it is guaranteed to implement the specification.

So far we have introduced the notions of type-2 computation and program synthesis abstractly. Our system attempts to apply these ideas to the task of robust aerial image interpretation. Parts of our solution are necessarily specific to the problem of image interpretation while other parts of our solution take the form of a general architecture, agent language, and synthesis engine that could in principle be applied to other problem domains. By discussing the abstract ideas of type-2 computation and program synthesis we have told only half of the story. A significant part of the problem involves the acquisition and application of models of the domain. Our approach to models and specifications has been corpus based. The model of the environment is extracted from a corpus of representative images while the specification for how the image should be segmented and labeled is learned by example from human expects. A human expert manually annotates an image corpus. The segmentations produced by the system are like those manually produced by the human expert.

Image segmentation is the partitioning of the image into disjoint regions, each of which is homogeneous in some property. It is an essential component of most image analysis systems and has been studied extensively Weska [12] Adams [1]

Zhu and Yuille [13]. Despite its long history of research, current segmentation algorithms are inadequate and unreliable in real world conditions. It has been argued that image segmentation is not a low level process at all Marr [9] and that semantics are necessary in order to produce good segmentations.

We refer to semantic labels assigned to regions as “image content descriptors” and the process of generating them as “image labeling”. An image content descriptor is a hypothesis about the content of that region (such as “urban area”).

The Minimum Description Length (MDL) agent negotiation formulation described below permits us to generalize the region competition algorithm of Yuille and Zhu [13] and to permit agents with differing levels of semantics to participate in the segmentation. This allows segmentation, classification, and image parsing to proceed cooperatively.

By structuring the image interpretation architecture as a feedback loop with dynamic evaluation and circuit synthesis the system can draw upon a variety of capabilities (implemented as agents) and prior knowledge of the world (from an image corpus) in order to provide robust behavior over a range of image conditions.

Our approach to semantics is statistical/information theoretic. A corpus of hand annotated images is used as a source of prior knowledge about semantic relationships in representative images and as a mechanism for importing expert knowledge about the images in the corpus and specifications for the system behavior.

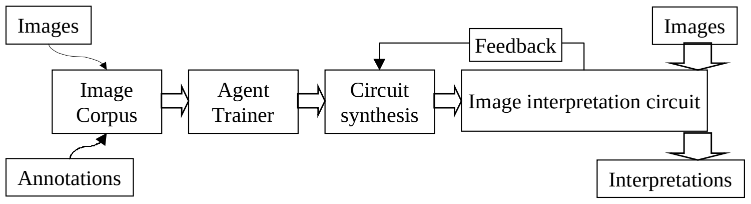

Fig. 1. Program Flow

Figure 1 shows the basic program flow for the system. Semantics are acquired from the image corpus containing representative images and expert annotations. Agents and meta-agents for image segmentation and interpretation are trained on the corpus allowing the circuit synthesis module to produce an image interpretation circuit that will interpret and segment images. Feedback allows agents self-knowledge to be used to monitor performance and to re-synthesize the interpretation circuit in order to track changes in image quality and content.

The corpus allows statistical information to be gathered that supports agent negotiation (agent architecture), expert knowledge extraction (circuit synthesis), and image parsing. We provide an overview of these components below.

In section 2 we describe the problem domain – image segmentation and labeling. Image interpretation is a special case of a general class of problems that we call "Interpretation problems". In section 3 we introduce the idea of interpretation problems. A key piece of the architecture is a novel agent architecture. The agent architecture supports competition among agents that attempt to find the global minimum description length interpretation. Section 4 gives a sketch of the agent architecture used in GRAVA. Section 5 provides an overview of the segmentation algorithm. The segmentation algorithm algorithm demonstrates the power of the MDL based agent architecture. Section 6 introduces the program synthesis engine.

2 The Problem Domain

As we have described above self-adaptive software involves several components:

1. A performance model;
2. A means of comparison; and
3. A mechanism for modifying the computation.

The goal of making self-adaptive software a reality involves finding ways of achieving each of these components and structuring a way of making them work together to produce a robust system.

We chose the problem of producing robust interpretations of aerial images as the test domain for our investigation into self-adaptive software.

Figure 2 shows an aerial image that has been segmented into regions. Each region has a coherent content designation such as *lake*, *residential*, and so on. The description that is formed by the collection of labeled regions is an interpretation of the image.

To produce such an interpretation several cooperative processes are brought into play. First the image is processed by various tools in order to extract texture or feature information. The selection of the right tools determines ultimately how good the resulting interpretation will be. Next a segmentation algorithm is employed in order to produce regions with outlines whose contents are homogeneous with respect to content as determined by the texture and feature tools. The segmentation algorithm also depends upon tools that select seed points that initialize the segmentation. The choice of tools to initiate the segmentation determines what kind of segmentation will be produced. Finally labeling the regions depends upon two processes. The first tries to determine possible designations for the regions by analyzing the pixels within the regions. The second looks at how the regions are related to each other in order to bring contextual information to bear on the process of selecting appropriate labels.

At any point a bad choice of tool for initial feature extraction, or for seed point identification, or for region identification, or for contextual constraints can

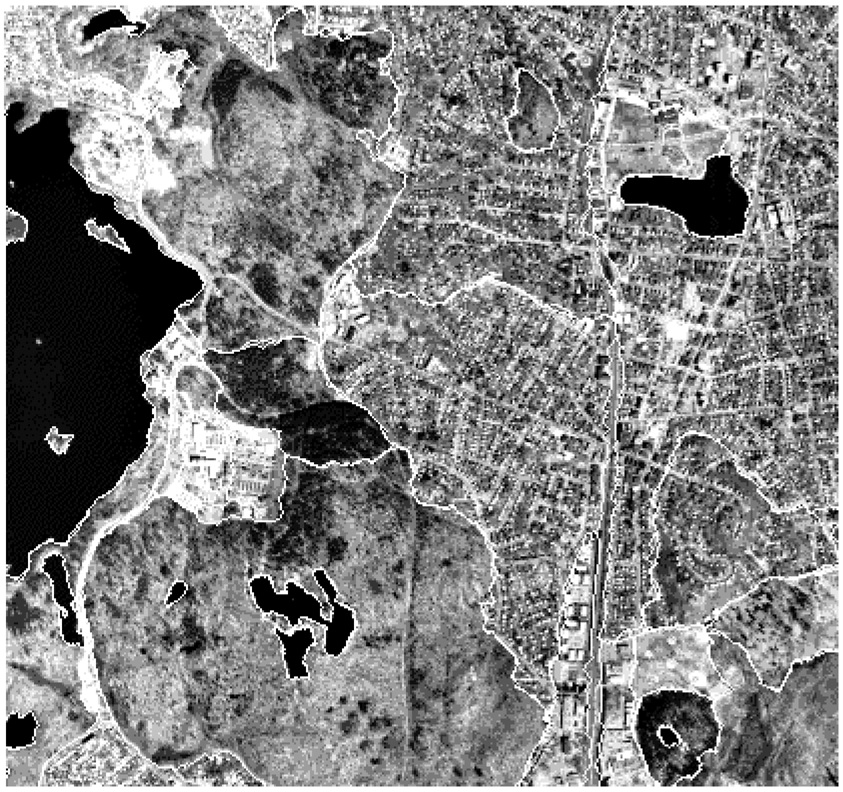

Fig. 2. Segmented Aerial Image

cause a poor image interpretation to result. The earlier the error occurs the worse the resulting interpretation is likely to become. For example a poor choice of tools for extracting textures from the image will result in a poor segmentation. The poor segmentation will result in poor region content analysis and so the resulting interpretation can be very bad indeed. The test for self-adaptive software is to determine how the program that consists of the collection of tools described above can be organized so that when a poor interpretation is produced the program self-adapts into a program that does a better job.

Figure 3 shows the dependency relationship between components of the system that we describe in this paper.

Image understanding requires models of the real world that can be used to produce a description of the image. We must be concerned with where these models come from and how to apply them to the task of image interpretation. In the case of a self-adaptive implementation the models must include an un-

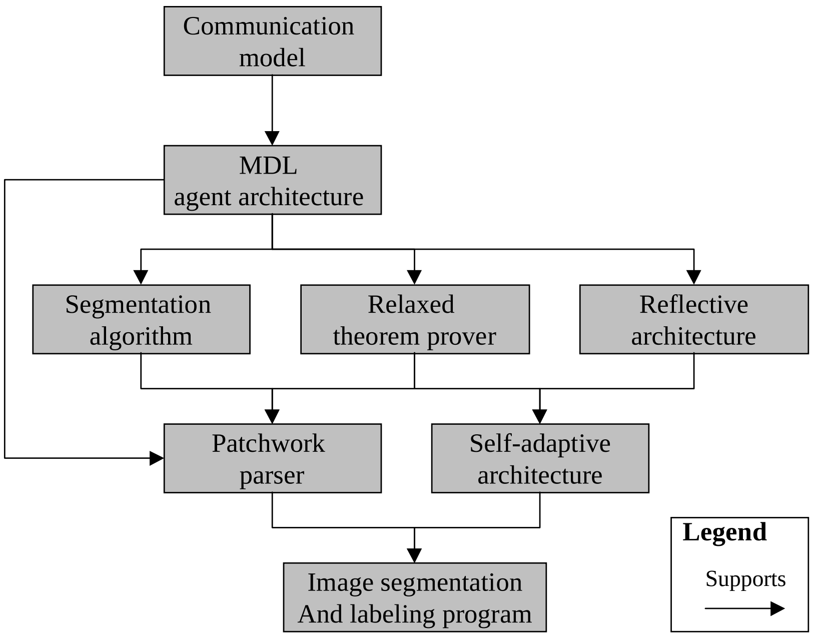

Fig. 3. Logical Components

derstanding suitable for estimating success of the interpretation and a model of how the parts of the program interrelate.

There are many ways of segmenting an image depending upon what is considered important for the task at hand. There is not one single *best* segmentation. Applications that wish to distinguish between different crop types may be interested in segmenting individual fields and assigning different labels to them depending upon their crop type. Other applications that are interested mostly in urban buildups may be happy to segment all fields as fields ignoring subtle differences caused by crop type.

In some applications the desired interpretation may be static but many applications will have different needs at different times changing dynamically as the application proceeds. A search program that attempts to find factories in images may at first be happy segmenting areas into industrial and agricultural regions and then with the industrial regions selected seek finer grained segmentation of the industrial region in order to search for the factory type that it is looking for. Systems involving mobile systems such as missiles or unmanned reconnaissance aircraft may have different interpretation needs depending on the phase of the flight. At other times weather conditions may dictate that only reduced levels of segmentation can be produced dues to deteriorated visibility. At a slower pace – but still dynamic for a satellite based Earth monitoring program – changes in seasons will affect the appearance of the landscape and the appropriateness of the various tools selected to interpret the image.

In summary, there are a large number of dimensions of change that can affect the performance of an image interpretation program of the form described above. To build a static solution to a dynamic program invites brittle program behavior. The self-adaptive software idea is to make the program monitor the various dimensions of change at runtime and when appropriate make changes to the running system so as to always do as well as it can under any given circumstance – and indeed to be able to estimate how well it has done at its task.

A major part of the self-adaptive problem involves mechanisms for implementing the self-adaptation itself but an equally important part of the problem concerns how models are build and maintained and how they are used in the monitoring and adaptation phases. Image interpretation it seemed to us presented an interesting case that could lead to an understanding of how self-adaptive solutions may be developed for image interpretation problems and perhaps interpretation problems in general.

3 Interpretation Problems

The problem of image interpretation belongs to an interesting class of problems that can broadly be charactarized as interpretation problems. Interpreting specifications of a segmentation is another example of an interpretation problem.

Our reflective architecture is based on the observation that for many problems a tower of interpretation problems can be constructed.

Consider the familiar problem of software development. In software development there are several layers of interpretation problem at work. Typically these are not automated, but the process in instructive because it is a familiar problem.

First the programs requirements are established. Typically these take on the form of a requirements document. The requirements say what the program should do at a very general level. It usually doesn't say much about how the program should operate or what it should look like.

The requirements document is *interpreted* to produce a specification that satisfies the requirements. The specification says how the requirements are to be achieved in the form of a program or suite of programs.

The specification document is *interpreted* to produce a design. The design shows at a fairly detailed level how the program components should work.

The design document is *interpreted* to produce a program — or suite of programs — that satisfied the design.

Finally the compiler interprets the program to produce the machine code that satisfies the program and the hardware interprets those instructions to produce the required behavior.

If a change occurs in any of these levels it must satisfy the level immediately superior to it and subordinate levels must be made to be compatible with the change. Since every part of the description was produced as a part of the interpretation of the higher level specification it is possible to *trace* the affects of the change up and down through the structure. The ability to track the path between components of a system and specifications and requirements that gave rise to them is called tracability.

At each stage a specification is interpreted by an interpretation procedure that brings some specialized knowledge to bear on the problem in order to produce a representation or description. The description thus generated becomes the specification of the lower level interpretation problem. The software engineering scenario described is a manual process although it may one day be automated.

The problem of self-adaptive software is to respond to changing situations by re-synthesizing the program that is running. To do this we reify the software development process within the domain of image interpretation programs. In our specific case, we must produce a segmentation that conforms to the specification implicit in the corpus.

The number of levels will vary from problem to problem. In principle there could be an arbitrarily large number of levels but in practice we expect most problems to have a small number of levels.

The level in the software development example described above that deals with the generation of machine code from a high level language description is of particular interest because our intuitions in this domain fit closely with the task of generating code from the corpus.

The compiler is an interpretation program that interprets the high level language source program and produces a description that draws upon knowledge built in to the compiler about the target machine. Nowhere in the high level source code are the details of the target machine represented. Indeed the code may be compiled with different compilers for different target machines. The compiler embodies various kinds of knowledge essential to producing a good representation of the source:

1. Knowledge of the high level language.
2. Knowledge of certain time and space considerations of certain patterns used in the source program and transformations into more efficient forms.
3. Knowledge of the target machine its instructions, registers, and efficiency considerations.

Of course the compiler may consist of several layers of interpretation problem in which the language is successively translated through intermediate languages until the target level is reached.

Typically a compiler is a carefully constructed program of considerable complexity in which various components perform the application of the domain knowledge (knowledge of the high level language and the target machine) to the task of translating the program. The task of the compiler can be presented in proof theoretic terms.

3.1 Compilation as Proof

The typical compiler can be thought of as the composition of several proof problems for example parsing, optimizing and producing machine code. The purpose of this discussion is to draw upon our intuitions of the compilation process and not to carefully model the behavior of a compiler. We restrict our discussion here to a single level of the compiler.

If we think of the task of the compiler as proving that the program can be computed by the target machine we can see that the resulting machine code is the axioms of the proof.

The knowledge in the compiler can be divided simply into two kinds

1. Knowledge of rules of inference and a procedure for recursively applying them in order to arrive at a proof.
2. Knowledge of the relationship between the source code and the target code in the form of rules.

In this model the compiler produces a tree structured proof. The leaves of the proof are machine codes. The machine codes are read off the fringe of the proof tree to produce the target machine language representation.

It is clear that various stages of the compiler fit this model. The parser is simply trying to prove that the source language is syntactically an instance of the set of programs defined by the high level language specification and the proof is the parse tree.

Of more immediate interest to us than the rationale for breaking up a compiler into a number of stages is the observation that each of these stages can be viewed as an interpretation problem as we have defined it above and that the interpretation problems form a tower.

4 Agent Architecture

In order to implement the interpreters necessary for solving the levels of interpretation problem required by our image understanding program we have developed a novel agent architecture.

Since the mid 1980's there has been growing interest in autonomous intelligent agents Giroux [5] Maes [7]. A key idea of these new architectures is that of "emergent functionality". The function of an agent is viewed as an emergent property of the interaction of the agent with its environment. Agents differ from purely algorithmic approaches in that part of their design depends on finding an interaction loop with the environment which will converge towards its goal. Where such interaction loops can be achieved significant robustness can be obtained. Our agent approach provides a way of building vision systems with such properties.

The GRAVA agent architecture is novel in that:

1. It supports meta-level agents that synthesize circuits of agents that mimic expert interpretation based on a corpus of hand annotated images.

2. It supports a novel agent negotiation mechanism based on a Minimum Description Length (MDL) formulation.

4.1 MDL Agent Negotiation

Image interpretation is fundamentally ambiguous. Interpretation involves finding the most probable interpretation. What we “See” is the most probable interpretation.

We have generalized this intuition in the form of a minimum description length or minimum entropy approach. In this approach agents compete to produce parts of a description of the image. The basis for competition is to seek to reduce the global entropy or to minimize the description length. This approach allows a natural interaction between agents of varying levels of sophistication and permits some analysis of the emergent functionality in terms of convergence to a solution.

The idea of minimum description length (Leclerc [6]) is that if a representation is optimally coded, the shortest description is the most probable one. This follows trivially because coding theory tells us that the most frequently occurring symbols should be encoded with the fewest bits. If we are looking for the most probable interpretation of an image it follows that we should seek the minimum description length. Part of the agent protocol for our architecture allows the agents to participate in the construction of the alphabet and to provide information that allows the entropy of the symbols in the working alphabet to be estimated.

There may be many agents at our disposal to generate the required description of the image. The architecture must provide for the agents to be harnessed in order to not only generate a description of the image but also to generate a description that meets the needs of the consumer. Figure 4 shows the components of the system. The agents are engaged in interpreting the image to form a description. The agents operate in an environment provided by a request from the consumer. The architecture provides selection of the appropriate agents as control over which agents are allowed to contribute to the generated description at various stages of its creation. These control activities are implemented as meta-agents whose operation can be summarized by the following control operations.

When the consumer makes (or modifies) a request, the meta-agents implement the following steps:

1. *Synthesize circuit.* The agents that are capable of producing the vocabulary of the description requested in the consumers request are selected from the pool of available agents. These agents themselves have pre-requisite input vocabularies which are used to select supporting agents from the pool. This process is repeated until raw sensor inputs are reached. This process selects from the pool of agents those agents that may contribute to the requested description. This stage is similar to plan generation in other agent systems.

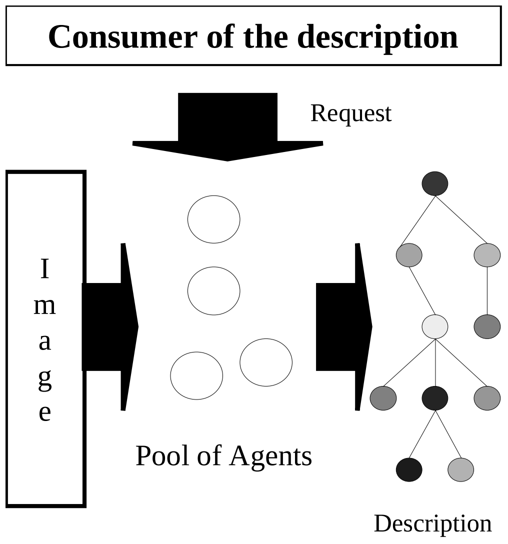

Fig. 4. Multi-Agent Interpretation Architecture

2. *Obtain vocabulary.* Once the circuit has been synthesized the agents in the circuit are asked for information about what vocabulary they consume/produce. The result is a global vocabulary that defines all of the symbols that may occur in the description of the image.
3. *Estimate entropy.* Having obtained the global vocabulary the entropy of each symbol is calculated. The entropy is the (possibly fractional) number of bits that would be required in an optimal coding of the vocabulary given the distribution of such features in real images that the agents have experienced in training (on the corpus). These entropies are used to determine the global description length as the segmentation and interpretation converges.
4. *Assign voting rights.* Once the entropies of the global vocabulary have been calculated the individual agents are informed of their strength when requesting an update of the global description. Agents use these strengths in order to compete to contribute to the global description. The strengths are determined so that agents whose contributions result in the smallest description length are preferred.
5. *Run!* The agents are allowed to operate on the description that begins as raw data and continues until no improvement in the description length has occurred for some number of iterations.

Since the goal of the agents is to produce a description, they operate by looking at the existing description for opportunities to improve (reduce the description length) the description. Often there are multiple agents that believe that they can improve the description at a particular place. If it were always the case that a local improvement in description length would guarantee a global improvement it would be sufficient to simply pick the agent that offers the greatest improvement locally. Generally however the locally best improvement does not guarantee the globally best improvement. As a result we employ a Monte-Carlo agent selection approach in which the agents that are competing for an update of the description are chosen at random weighted by their voting strength calculated above. This process continues until a complete description has been generated. The process then begins from the raw data again and the resulting description is compared to the previous one. The best description is retained. This process repeats until the description length of the resulting description fails to improve after some number of iterations.

5 Semantic Segmentation

The segmentation algorithm is a good example of the MDL agent architecture in action. The algorithm extends work by Leclerc [6] and Zhu & Yuille [13].

Marr [9] argued that segmentation was not a well founded problem for a variety of reasons but principally because it did not seem possible to find low level processes that could account for the apparent ability of human vision to perceive regions.

Our view of segmentation is that contrary to the common view that segmentation is a low-level process we take segmentation to be a cooperative process with all levels of processing. Where other algorithms pick *a priori* methods for determining things like curvature characteristics, we let such things be driven by the semantics.

The image interpretation problem is viewed from the communication theory view as one of intelligent compression. We wish to throw away all information that we are not interested in and then package up the result in such a way as to minimize the message length. Most image compression algorithms decide what to retain on the basis of being able to reproduce a good approximation to the original image as perceived by a human viewer. Lossy compression may however throw away some crucial detail for certain tasks even though the reconstructed image looks like a close approximation to the original. We are not interested in image reconstruction for human viewing but would like to reconstruct a semantic rendition of the image for whatever purpose the interpretation system is engaged.

Compression algorithms like MPEG are designed as architectures that permit special purpose methods to be plugged in to achieve the compression. Our approach to image segmentation is similar. We provide a basic low-level base level segmenter – a knowledge free segmenter – that makes few assumptions and discards nothing. We also provide an architecture that permits semantic agents that cooperate with the segmenter to be plugged in.

The choice of which agents are used in performing the segmentation and what inputs are made available to the segmenter determine the nature of the resulting segmentation.

There are a number of important structural issues regarding segmentation.

1. Region contents.
2. Region size and shape.
3. Region neighbors.

Size and shape includes the characteristics of the outline of the region. Different regions will want different outlines depending on semantic attributes. For example a leaf outline might have a smooth curved outline or an outline with points of high curvature depending on the leaf type. Low level algorithms such as Zhu & Yuille [13] prefer smooth outlines for all region types.

5.1 Architecture for Segmentation

Segmentation should be facilitated by image interpretation and image interpretation is facilitated by segmentation. The idea of the architecture is to allow the two processes of interpretation and segmentation to proceed cooperatively – each influencing the other and finally converging upon the best interpretation and segmentation that can be achieved.

When we know what a feature in an image is, we can know what is salient about that feature – and therefore what must be a part of the description. What is salient is not just a function of the feature. It is also a function of what we want to do with the description. It is semantics that allow us to decide what can be omitted from an image description. Without semantics we cannot discard any information. Our algorithm is called semantic segmentation because it allows semantics to govern what is retained in an image description. Leclerc's list of criteria for image partitioning insisted on a lossless representation. That is because the descriptions had no idea of salience and therefore had to retain everything. Our algorithm and architecture doesn't define what is salient, but provides a framework within which salience can be specified.

The image interpretation problem is one of efficiently communicating a description of the image content that allows sufficient reconstruction of the image for the purposes that the application demands. One way of reducing the description length for communication is on the basis of shared information. If the transmitter and the receiver share knowledge that shared knowledge need not be part of the transmitted message. If an image contains a chair it is not necessary or useful to transmit every detail of the chair. We can represent it as a "chair". To be useful it is probably necessary to parameterize "chair" to specify enough details for a useful reconstruction. We may want to specify what kind of chair it is – swivel office chair, arm chair etc. It may be necessary to provide its size position and orientation too. "Chair" in this example is part of the description language, it is image content semantics, and it is a parameterized model. Sometimes it may be structurally decomposed and other times it may be a single level model.

Where the "chair" model – or models in general – come from is not important. They can be learned from data, or they can be manually constructed. What is important for our purposes is that there is a mechanism for establishing the probability of the parameterized model occuring in the image given whatever prior information is available.

5.2 MDL Agent formulation of the algorithm

A semantic element or model is implemented as an agent that attempts to describe a region of the image in terms of the model that it represents. Such a description consists of the name that uniquely describes the model and a vector that parameterize the model. The agent also produces an estimate of the probability of the parameterized model appearing in the image.

An agent has the responsibility of having regions of the image represented as compactly as possible. To achieve this, once an agent instantiated a representation of a region of the image it begins to search for other agents that can represent the region more efficiently. To do this it invokes feature finders that may infer the applicability of an agent. This is somewhat like the opposite of the region growing algorithms. Rather than regions trying to grab pixels from its neighbors, our algorithm has regions attempt to give away its pixels in order to produce a smaller description.

We want higher level semantics to influence segmentation but we don't want every interpretation agent to have to have built-in support for segmentation. We prefer to have a general segmenter that knows as much as is necessary about maintaining a segmentation and as little as possible about built-in assumptions that lead to a segmentation. For this reason, in order to ground the cooperative process between segmentation and interpretation we define a base level segmenter that gets the process started.

Figure 5 shows the flow of the semantic segmentation algorithm. Initially a single region is initialized that contains the whole image. Its description length is computed based on the null semantics representation of the base segmenter (described below).

The first step (*1) is to attempt to find regions that can be introduced that can reduce the description length. If no such opportunities can be found the algorithm terminates. New regions can be null semantic regions (like the original region) or can be semantic regions.

In the second step (*2) each region attempts to reduce the description length by giving away boundary pixels to neighboring regions. This is repeated until no more reduction can be achieved by giving pixels away to neighbors. If no reduction occurred during this step the algorithm terminates.

The third step (*3) attempts to reduce the description length by merging neighboring regions. When neighboring regions are merged the total boundary cost is decreased but the cost of representing the internal pixels may increase if the regions are not similar.

The above three steps are repeated until the algorithm terminates in step *1 or step *2.

Step *1 is complicated by conflicting opportunities to introduce new regions (the potential new regions overlap). These cases are handled by Monte-Carlo

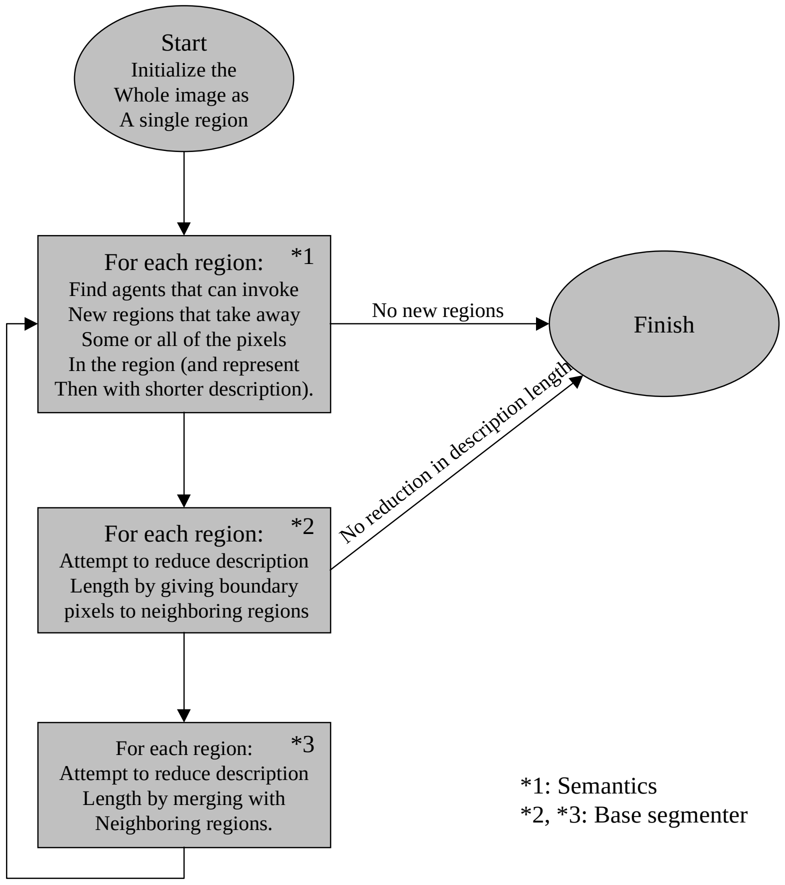
Start
Initialize the
Whole image as
A single region
For each region: *1
Find agents that can invoke
New regions that take away
Some or all of the pixels
In the region (and represent
Then with shorter description).
No new regions
Finish
No reduction in description length
For each region: *2
Attempt to reduce description
Length by giving boundary
pixels to neighboring regions
For each region: *3
Attempt to reduce description
Length by merging with
Neighboring regions.
*1: Semantics
*2, *3: Base segmenter

Fig. 5. High level schematic of the semantic segmentation algorithm

decision points. Step *1 corresponds to seed point selection when the new regions are are null semantics base level regions.

Step *1 introduces semantics by allowing agents that implement semantics to reduce the description length.

Steps *2 and *3 implement the base level segmenter.

5.3 Base Level Segmentation

The problem of defining the base segmenter is to define a compact and useful representation of the image in terms of regions. The purpose of designing the message format is to enable us to compute the message length that would be required to communicate the image. We don't actually need to construct the message – or to communicate it.

The image is to be represented as n regions $R_0..R_{n-1}$. We begin the image representation with n the number of regions. In order to represent a region it is necessary to describe the pixels that the region contains. One way to do that is to represent the boundaries of the region. Given the boundaries of the region, the pixels can be described in order from top left to bottom right staying within the boundaries.

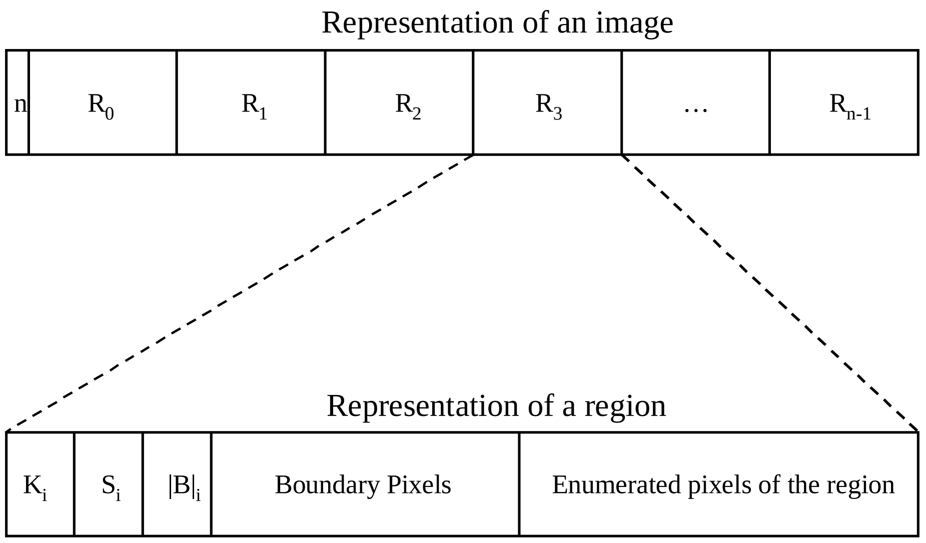
Representation of an image
n
R_0
R_1
R_2
R_3
...
R_{n-1}
Representation of a region
K_i
S_i
$|B|_i$
Boundary Pixels
Enumerated pixels of the region

Fig. 6. Base Description Language for Images

We may have many different approaches to representing regions so the first thing that we represent is the kind K of region that we are going to transmit.

$$DL(K) = -log_2 P(K) \tag{1}$$

We begin by specifying the representation for the base region type that makes no assumptions about the shape or contents of the region. Having determined that the region is a base region we specify how many borders $|R|$ the region R has.

$$DL(|R|) = -log_2 P(|R| = n) \tag{2}$$

Next we describe each boundary of the region. The first boundary is the outer boundary. If there are subsequent boundaries they are all internal and contain embedded (subtractive) regions. For each boundary $B_i \in R$ we define an (arbitrary) starting pixel S_i.

$$DL(S_i) = -log_2 \frac{1}{p} \tag{3}$$

We specify the number of pixels $|B_i|$ that comprise the boundary.

$$DL(|B_i|) = -log_2 P(length(B_i = k)) \tag{4}$$

Then for each boundary pixel in order we describe the position as a relative move from the previous pixel $M_{i,j}$. Since there are 7 possibilities at each step (a boundary cannot double back on itself) the length move can be represented as:

$$DL(M_{i,j}) = -log_2 P(M_{i,j}|M_{i,j-1}, Mi, j-2..., M_{i,j+1}, M_{i,j+2}, ...) \tag{5}$$

The conditional probabilities can be learned from a corpus of representative boundaries. This is a knowledge free estimate of smoothness constraints for the boundary of the region. Making the representation too sparse requires too many outlines to obtain a good estimate. We use the following in our implementation:

$$DL(M_{i,j}) = -log_2 P(M_{i,j}|M_{i,j-2}, Mi, j-1, M_{i,j+1}, M_{i,j+2}) \tag{6}$$

This determines the position and shape of the region as well as the number of pixels in the region. It also allows us to define a unique ordering of the pixels. Given such an ordering we can represent the individual pixels in the region without knowing more about the region than its bits.

Our intuitions about regions are that they are homogeneous.

If they are homogeneous, we mean that all the pixels of a region are taken from the same distribution. We can estimate that distribution by building a histogram of the pixels in the region. Then the coding length of a pixel whose intensity is p is given by:

$$DL(Pixel_{i,m}) = -log_2 P(intensity = p|region = R_i) \tag{7}$$

5.4 Region Splitting

Figure 7 shows a region splitting in action. On the left a region is growing around the two islands. At this point the islands are part of the background region. Twenty iterations later the growing (lake) region has expanded past the islands and the islands have split away from the background and are now separate regions.

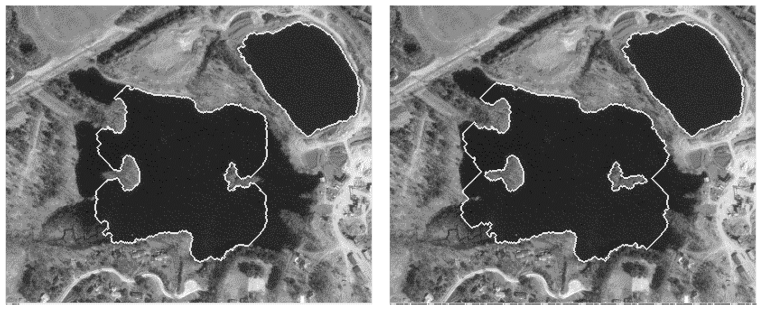

Fig. 7. Region Splitting

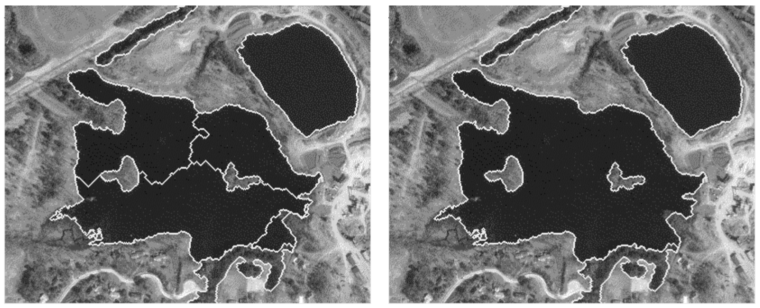

Fig. 8. Region Merging

5.5 Region Merging

Figure 8 shows a region splitting in action. On the left a number of regions have grown to accommodate the lake. In the next iteration (the right image) the regions internal to the lake have been merged. It is more compact to represent them as a single region than as separate regions.

Next we give an overview of the segmentation algorithm.

1. The algorithm begins with the entire image as a single region. The description length is calculated. Each region is represented by an agent.
2. each agent (initially only one) seeks to reduce the description length of its region. There are two ways of doing this.
   (a) The agent can give away pixels of its region to a neighboring or internal region.
   (b) The region can spawn new regions thereby giving up the pixels contained in those new regions. In order to spawn new regions, feature detectors may be run over the region to find support for viable internal regions.

These steps are done in turn as follows. First for each region in the image the agent attempts to give away boundary pixels to neighboring and internal regions. The pixels can be given away if the description length is decreased more for the giving region than it is increased for the acquiring region. This process is repeated until convergence. Convergence occurs when there are no opportunities for any existing region to give away pixels. Initially this point is reached immediately because there is only a single region and therefore no region to give pixels away to.
Upon convergence the agent for each region solicits agents to produce sub-regions. The addition of sub-regions must reduce the global description length. Once new agents and regions have been initialized the prior attempts to reduce global description length are performed. The cycle is repeated until no new agents get introduced.

6 Circuit Synthesis

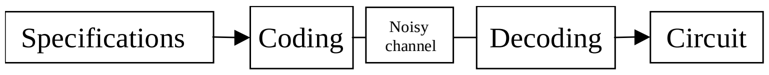

Fig. 9. Image Interpretation Channel

The goal of the system is not simply generate a good description and segmentation of the image but to generate a description and segmentation that is similar to ones generated by a human expert. This situation is modeled as a problem of communicating the expert's intent to the program. the human expert provides specifications in the form of an annotated corpus that defines what the image interpretation programs should produce from the image. The coding of specifications as annotations and subsequent decoding by synthesizing a circuit of agents that can reproduce the annotations given an image is modeled as transmission through a noisy channel (see Figure 9). The goal of the circuit synthesis module is to maximize the mutual information between the behavior of the produced circuit and the specifications.

6.1 Learning by Example

The purpose of the program is to segment an image in a way that is as similar as possible to an expert segmentation. This formulation of the segmentation problem is different from the way segmentation problems are usually defined. Typically segmentation programs are judged subjectively by looking at the resulting segmentation and judging how well major regions have been identified.

For a segmentation to be useful it needs to be able to make distinctions that later stages of processing are interested in because ultimately the success of the image interpretation problem depends upon the usefulness of the segmentation. For our purposes the success of the segmentation is defined as being the closeness of the segmentation to that of a human segmenter.

In general we can think of the segmentation problem as being one of producing a segmentation that satisfies a specification for what the segmentation should be. In our case the specification takes the form of the corpus of hand annotated images. There is a great similarity between the task of building annotations to mimic those of an expert and what is known as learning by example.

6.2 Generalizing the Problem Description

The task of synthesizing a program from a specification is as we have discussed above similar to the problem of compiling a program and the characterization of the problem as one that is essentially a theorem proving problem suggests a way of building a general architecture. Before we can do that however there is one level of relaxation that is necessary.

In compiling a program into machine code, we generally deal with certainty. The high level language that is being compiled is not ambiguous and the machine code can be relied upon to perform as expected. In the case of image interpretation or interpretation problems in general the source specification may be ambiguous and the rules are not guaranteed to succeed. Instead we have a way of characterizing the likelihood of succeeding. Thus the compiler is a very special case of a more general problem. It is this aspect of the problem that makes a self-adaptive software solution important.

In the compiler example we were able to represent the rules as models that take an input and produce a description.

To generalize the problem we add two capabilities:

1. A post-test to test the effectiveness of the generated description.
2. A measure of utility of the rule.

Before, if the code could be generated it was guaranteed to produce the correct result. We must have a way of evaluating the result to establish its effectiveness. Furthermore we must have a way of recovering from cases where the test is negative.

In the simple compiler case any rule that could generate code from the source language was as good as another.

In the deterministic case a proof is a proof. In our generalized version of the problem some proofs are better than others. Specifically the proof that has the highest overall likelihood of succeeding is the best proof. It is ridiculous to call the task theorem proving or the result to be a proof when the result has only a likelihood of succeeding. Henceforth therefor we refer to the generalization of theorem proving a *support generating*, the result of applying such a procedure a *support*, and the procedural embodiment of the process a *support generator*.

Each node has a probability associated with it. This probability is the probability that the node will succeed in its task. Each node contains a test procedure to determine whether it has succeeded in practice. Viewing the support as a program the probability that the entire program will succeed is given as follows.

$$P(rule_n = success|P(sub_0)\&P(sub_1)...\&P(sub_m)) \tag{8}$$

In some systems only terminal nodes (nodes that produce leaves) may have probabilities less than 1.0 while other systems may allow any node to have a fractional probability of success.

We can represent each leaf of the tree as a code of length $-log_2(P(leaf_i))$. The description length of the proof then is given by:

$$DL(support) = \prod_{s \in support} -log_2 P(s) \tag{9}$$

The support generator searches for rules that generate a valid supports and the description length is calculated for each. The Monte-Carlo algorithm selects the best support by taking enough samples to find the shortest description length support. The resulting program is produced trivially from the support tree. At runtime when a test fails probability assignments for the leaves are updated and the synthesis engine runs again with probabilities that reflect the knowledge of the environment known at runtime. The support that has the shortest description length generates the program most likely to succeed based on what is known about the environment at the point in time when it is generated.

7 Conclusions

Many problems in artificial intelligence can be characterized as finding the most probable interpretation from an ambiguous set of choices. Speech understanding natural language understanding, visual interpretation are obvious examples of problem areas with those characteristics. Typically what is the most probable interpretation is something that depends upon context that is only available at runtime. The GRAVE architecture supports a self-adaptive approach to this class of problems.

A reflective tower representing different levels of specification and an interpretation function that converts specifications into executable code by using a support generator (relaxed theorem prover) as a code generator allows reflection to identify precisely the piece of code responsible for a deviation in performance from the intended behavior and allows that piece of code to be dynamically resynthesized so as to produce a modified piece of code that reflects the best that is known about the environment at that point in time.

This model has proven effective at dynamically mutating an image segmentation program so as to produce the best interpretations.

The applicability of this architecture is restricted to problems that can be construed as interpretation problems. Nevertheless we believe that there are a significant number of interesting problems that fit within those confines and that the GRAVA architecture provides a mature architecture for building self-adaptive programs at least for image interpretation problems but likely also for other problems that can be cast as interpretation problems.

Little is known about the stability of systems built this way and there are as yet no known guidelines for building systems that are intended to be stable. More work remains to be done in this area.

8 Acknowledgements

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-98-0056. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

References

1. R. Adams and L. Bischof. Seeded region growing. *IEEE Trans. on PAMI*, 16(6), 1994.
2. Alan Bawden. Reification without evaluation. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 342–351, 1988.
3. C. Smith Brian. Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory for Computer Science, January 1982.
4. Jim des Rivieres and C. Smith Brian. The implementation of procedurally reflection languages. In *Proceedings 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas*, pages 331–347, August 1984.
5. Sylvain Giroux. Open reflective agents. In J. P. Muller M. Wooldridge and M. Tambe, editors, *Intelligent Agents II Agent Theories, Architectures, and Languages*, pages 315–330. Springer, 1995.
6. Y. G. Leclerc. Constructing simple stable descriptions for image partitioning. *Int. J. of Computer Vision*, 3:73–102, 1989.
7. Pattie Maes. Situated agents can have goals. In Pattie Maes, editor, *Designing Autonomous Agents*, pages 49–70. MIT/Elsevier, 1990.
8. Pattie Maes and Daniele Nardi. *Meta-Level Architectures and Reflection.* North-Holland, 1988.
9. D. Marr. Early processing of visual information. *Phil. Trans. R. Soc. Lond. B*, 275:483–524, 1976.
10. P. Robertson. A corpus based approach to the interpretation of aerial images. In *Proceedings IEE IPA99.* IEE, 1999. Manchester.

11. Paul Robertson. On reflection and refraction. In Akimori Yonezawa and Brian C.Smith, editors, *Reflection and Meta-Level Architecture, Proceedings of the 1992 International Workshop on New Models for Software Architecture.* ACM, 1992. Tokyo.
12. J. S. Weska. A survey of threshold selection techniques. *Comput. Graph. Image Process*, 7:259–265, 1978.
13. S. C. Zhu and A. L. Yuille. Region competition: unifying snakes, region growing, and bayes/mdl for multiband image segmentation. *IEEE Trans. on PAMI*, 18(9):884–900, 1996.