GRAVA: An Architecture Supporting Automatic Context Transitions and its Application to Robust Computer Vision

Paul Robertson
Dynamic Object Language Labs
9 Bartlet Street, Suite 334
Andover, MA 01810
email:probertson@doll.com
and
Robert Laddaga
Computer Science and Artificial Intelligence Laboratory
Massacusetts Institute of Technology
Cambridge, Massachusetts, USA
email: rladdaga@ai.mit.edu

ABSTRACT

We describe a software development approach for vision that enhances robustness by making novel use of context. Conventional approaches to most image understanding problems suffer from fragility when applied to natural environments. Complexity in Intelligent Systems can be managed by breaking the world into manageable contexts. GRAVA supports robust performance by treating changes in the program's environment as context changes. Automatically tracking changes in the environment and making corresponding changes in the running program allows the program to operate robustly.

We describe the software architecture and explain how it achieves robustness. GRAVA is a reflective architecture that supports self-adaptation and has been successfully applied to a number of visual interpretation domains. This paper describes the protocols and the interpreter for GRAVA.

1 Introduction

As our software based systems become more complex, and deal increasing with real world systems, the problem of robustness for such systems has intensified. New approaches to software development are clearly needed to deal with this growing problem. In this article we describe a prototype implementation of the Self Adaptive Software approach to software development. This prototype Self Adaptive system is designed to implement robust vision systems, but its applicability is considerably more broad.

Image understanding programs have tended to be very brittle and perform poorly in situations where the environment cannot be carefully constrained. Natural vision systems in humans and other animals are re-markably robust. The applications for robust vision are myriad. Robust vision is essential for many applications such as mobile robots, where the environment changes continually as the robots moves, and robustness is essential for safe and reliable operation of the robot.

The lack of robustness noted above is by no means limited to computer vision—it is a problem observed by all programs that deal directly with the real world and includes such things as speech recognition and mobile robots. We are surrounded by examples from nature of natural systems that interact robustly with the real world but until recently we have not had the tools to engineer such systems.

It is a little appreciated fact that the overwhelming majority of microprocessors (more than ninety eight percent) are use in embedded applications and the trend is for the percentage of such processors deployed in embedded applications to continue to grow. At the same time the power of such processors is growing rapidly and forcing us to consider the problems that come with our growing aspirations for embedded systems.

In the spring of 1998, the US agency DARPA introduced the term "Self adaptive software" to describe a software methodology that aspires to solve exactly the kind of problem described above.

Self adaptation is a model-based approach to building robust systems. The environment, the program's goal, and the program's computational structure must be modeled. In principle, the idea is simple. The environment model and the program goal model both support continuous evaluation of the performance of the program. When program performance deteriorates, the program goal model and the computation model together support modification of the program structure. In this way, the program structure evolves

as the environment changes, even radically, so that the components of the program are always well suited to the environment in which they are running. The conjecture is that robust performances results from having all components operating within their effective range.

Although the complexity of the real world is overwhelming the complexity does not assert itself at the same time. At any instant a programming is operating within a *context* in which the complexity of bounded. In AI we have been fairly successful at building systems that perform robustly within environments of restricted complexity. If we can divide the complexity of the world up into a collection of contexts, each of a bounded and manageable size, we can in principle consider the hard problem of making a robust embedded system as an easier problem formulated as the composition of a collection of manageable parts.

Even if we could know all the different states that the environment could be in, we wouldn't know *a priori* what state the environment would be in at any particular time. Consequently, in order to achieve robust image understanding, programs should determine the state of the environment at runtime and adapt to the environment that is found. In practice it is likely that the set of possible contexts cannot be explicitly enumerated *a priori.*

A premise of the self-adaptive approach is that it should be possible, at runtime, to synthesize context specific systems, to determine the need to change context and to self-adapt the program so that the program's context matches the state of the environment and operates robustly because each of its components is operating well within their optimal range.

The progress of a self-adaptive program as its environment gradually changes can be viewed as a trajectory though the space of contexts. Figure 1 shows the trajectory of such a program through the space of complexity that the environment can exhibit. The ellipses represent the range of operation of various modules. Each module handles only a small fraction of the total complexity that the program operates in but by switching algorithm from time to time as the program trajectory through complexity space moves out of the domain of capability of one algorithm and into another the program can be kept operating robustly.

It may be argued that implementing all of the necessary algorithms is a formidable task but in practice many of these can be *learned* such as from a corpus.

In summary, the idea of self-adaptation is to adapt the program to a particular "context". In order to achieve this adaptation we build structural descriptions that facilitate dividing the model space into contexts and provide a mechanism for determining when a context is a good fit to the environment.

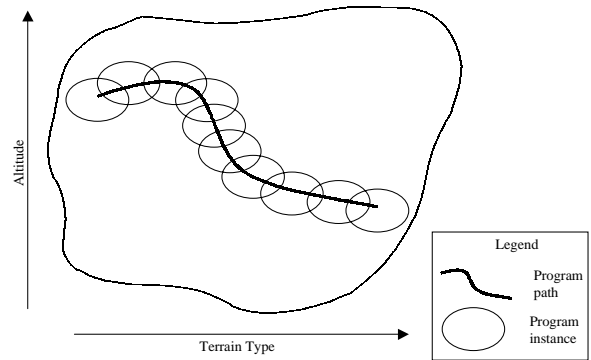In speech understanding systems it is common to have separate grammars for specific contexts. For ex-

Figure 1. Path of a Self Adaptive Program

ample, when a speech understanding system is waiting for a phone number, the probability of a word being one of the digits is much greater than it is in other contexts. One reason for this practice in natural language is that when the vocabulary and the set of parse rules become too large, the HMM methods, that are frequently used in speech processing, become unwieldy. It is also useful, and perhaps necessary, to have such contexts in order to provide enough constraint to make sense of what is often a very noisy signal.

We also note from studies of human perception that we always interpret images within a context that defines our prior expectations about what we expect to see. Psychologists call this "priming". This reaches an extreme form in the case of model-based image analysis, in which programs "hallucinate" [2] one of a small set of models onto images. Typically, the human programmer defines the "context" by providing *a-priori* the (small) set of models that can be hallucinated.

The need for contexts to manage the diversity of the world is no less important for image understanding. AI has long understood the importance of contexts. In 1975 Minsky introduced the notion of *frames* [9] which was essentially an approach to contexts. Frames have been used extensively in AI research, especially for natural language. Riseman's Schemas [3] was a similar idea specifically for Computer Vision.

Man of the ideas prevalent in natural language and speech understanding have direct counterparts in computer vision. The first application of the GRAVA architecture [11] was to the interpretation of satellite aerial images. In that program satellite images were segmented into regions of homogeneous content and the regions were parsed, much as words are in a sentence to form a structural understanding of the image. Different image types are comprised of different kinds of regions, different colors and textures, and different parse rules. Rather than making one huge grammar

that includes all textures and region types, it is better to have grammars, and optical models tailored to the context because tailored contexts provide greater accuracy and constraint. In that program the contexts as well as the grammars and region content models were learned from a corpus of images annotated by a human photo interpreter.

Contexts occur for a variety of reasons, at different levels of processing, and in different parts of the corpus. Given a set of images it is generally not possible to divide the images into separate piles with each pile representing a different context. Contexts for different aspects of the problem can be composed in a variety of ways. The explosion of possible combinations of contexts is one reason why the self-adaptive approach is attractive. That is, rather than generating all possible combinations of contexts in advance—and then having a "big switch" to choose which to use—it is better to generate the particular combination of contexts on demand.

To better understand the idea of contexts, consider the case of optical model contexts and language model contexts.
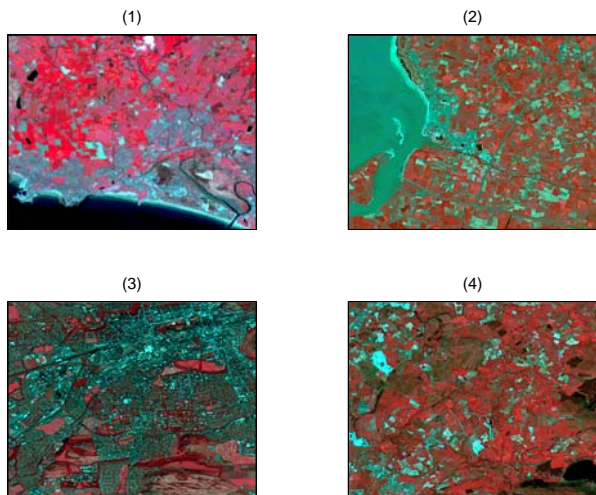

Figure 2. Image Contexts

Figure 2 shows four multi-spectral color SPOT images from the color corpus that demonstrate different contexts. Images (1) and (2) are similar in content (mostly farmland and small towns) but the colors and textures of the regions are very different. In fact, the images are taken under different imaging conditions. In the case of these two images, the major difference is with the optical models, since, grammatically, the two are rather similar. In images (3) and (4) the nature of the terrain is very different. Image (3) shows part of a major city whereas image (4) shows a rural setting with only small villages. The grammar that is suitable for parsing images 3 and 4 is quite different. Attempting to interpret any of these images with

the wrong collection of optical or grammatical models may be expected to produce a poor result especially since knowledge weak segmentation algorithms often give poor results. In this case, the reason for the differences between image (1) and image (2) were changes in the SPOT technology used to image them.

Separating contexts in this way suggests (for example) that grammar learned using one sensor implementation may be usable even when the sensor technology is changed, so long as new optical models are available for the new sensor. One of our original goals was to be able to build vision systems that continue to work well despite changes in the sensor and image preprocessing environment. For that reason, the separation of imaging contexts from language contexts is an appealing idea.

Even without changing the sensor technology, different optical contexts may be called for, for example, variations in optical characteristics due to changes in weather conditions or season. When the seasons change, the optical characteristics of fields and trees vary dramatically but the language of the terrain changes little grammatically because it is defined by the man-made structures and natural constraints of the terrain. San Francisco is not a city one day and a rural region the next.

In the case of the picture grammar, we wanted to automatically produce batches of rules that constitute image grammar contexts. These batches of rules are models of the context in question. Similarly, in the case of optical models, we want to produce batches of optical models that can be grouped into contexts.

In this paper we describe GRAVA, an architecture for building self-adaptive programs, and describe its theory of operation.

2 An Overview of the GRAVA Architecture

Vision (and Robotics) systems lack robustness. They don't know what they are doing, especially when things change appreciably (i.e. in situations where technologies such as neural nets are ineffective).

Reflective architectures—an idea from AI—offer an approach to building programs that can reason about their own computational processes and make changes to them.

The reflective architecture [8, 4] allows the program to be aware of its own computational state and to make changes to it as necessary in order to achieve its goal.

However, much of the work on reflective architectures has been supportive of human programmer adaptation of languages and architectures rather than self-adaptation of the program by itself.

Our use of reflection allows the self-adaptive ar-

chitecture to reason about its own structure and to change that structure.

2.1 Interpretation Problems

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.

2.1.1 Layers of Interpretation: An Example

A key idea in the formulation of our reflective architecture is that problems can often be described in terms of interconnected layers of interpretation forming a hierarchy of interpretation problems. A simple and familiar example of such a layered view is the process of how large software projects are executed.
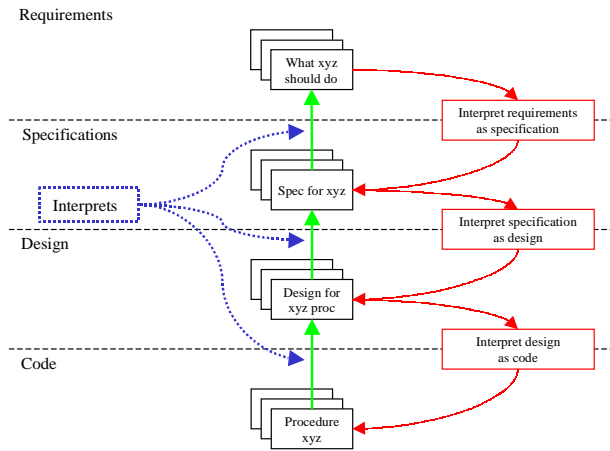

Figure 3. Example of the relationship between levels of interpretation

Large software projects, especially software projects of defense contractors, start out with a requirements document. This document says what the program should do but doesn't say how it should be done. Someone *interprets* the requirements document as a software system and produces a set of specifications for the components of the software system that satisfies the requirements. The specifications are then *interpreted* as a program design. The program design lays out the procedures that make up the program that implements the specification. Finally a programmer interprets the program design to produce a body of code. If care is taken to retain back pointers it is possible to trace back from a piece of code to the part of the design that it interpreted. Parts of design should be traceable to the parts of the specification they interpret and parts of the specification should be

traceable to the parts of the requirements document that they interpret.

Figure 3 shows the relationship between different levels of interpretation in the software development example.

When requirements change, as they often do in the lifetime of a software system, it is possible to trace which pieces of the system are affected. In this example, at each level, an input is interpreted to produce an interpretation that is used as the input at a subsequent level.

Each component of the system "knows" what it is doing to the extent that it knows what part of the level above it implements (interprets).

The purpose of the reflective architecture is to allow the image interpretation program to be aware of its own computational state and to make changes to it as necessary in order to achieve its goal. The steps below provide a schematic introduction to the GRAVA architecture.

1. The desired *behavior* is specified in the form of statistical models by constructing a corpus.

2. The behavior, which covers several different imaging scenarios, is broken down into contexts. Contexts exist for different levels of the interpretation problem. Each context defines an expectation for the computational stage that it covers. Contexts are like frames and schemas; but because the contexts are gathered from the data automatically it is not necessary to define them by hand.

3. Given a context a program to interpret the image can be generated from that context. This is done by *compiling* the context into a program by selecting the appropriate agents.

4. The program that results from compiling a context can easily know the following things:

   (a) What part of the specification gave rise to its components.

   (b) Which agents were involved in the creation of its components.

   (c) Which models were applied by those agents in creating its components.

   (d) How well suited the current program is to dealing with the current input.

5. The division of knowledge into agents that perform basic image interpretation tasks and agents that construct programs from specifications is represented by different reflective levels.

2.2 Reflective Interpreter for Self-Adaptation

The techniques for implementing reflection [15, 8] have become common in modern languages [10, 6] and architectures. Unlike traditional implementations, which have largely been supportive of human programmer adaptation of languages and architectures, we use reflection as a way of supporting self-adaptation of the program *by itself*. There are two principal differences in our use of reflection:

1. We open up the program to itself so that by knowing what it knows it can use what it knows to alter itself in order to respond to changes in the real world.

2. We do not wish to change the semantics of the program/language, we wish to change the program itself.

A reflective layer is an object that contains one or more "interpreter". Reflective layers are stacked up such that each layer is the meta-level computation of the layer beneath it. In particular each layer is generated by the layer above it. The face identification application described in this paper uses two layers.

Each layer can reflect up to the layer above it in order to self-adapt. The prototype GRAVA implementation is written in Yolambda [7] a dialect of Scheme [5]

```
(defineClass ReflectiveLayer
  ((description) ;; the (input) description
                 ;; for this layer
   (interpreter) ;; the interpreter for this
                 ;; layer's description
   (knowledge)   ;; a representation of world
                 ;; knowledge
   (higherlayer) ;; the meta-level above this
   (lowerlayer)));; the subordinate layer
```

A reflective layer is an object that contains the following objects.

1. *description:* the description that is to be interpreted.

2. *interpreter:* a system consisting of one or more cascaded interpreters that can interpret the description.

3. *knowledge:* a problem dependent representation of what is known about the world as it pertains to the interpretation of the subordinate layer. For the face identification application knowledge consists of evidence accumulated from agents supporting each of the contexts (age, race, sex, lighting, and pose).

4. *higherlayer:* the superior layer. The layer that produced the interpreter for this layer.

5. *lowerlayer:* the subordinate layer.

The semantics for a layer are determined by the *interpret*, *elaborate*, *adapt* and *execute* methods which we describe in turn below.
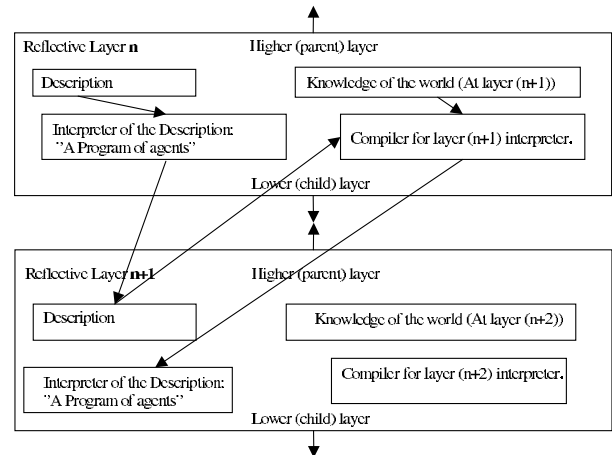

Figure 4. Meta-Knowledge and Compilation

Figure 4 shows the relationship between reflective layers of the GRAVA architecture.

Reflective Layer "n" contains a description that is to be interpreted as the description for layer "n+1". A program has been synthesized either by the layer "n-1" or by hand if it is the top layer. The program is the interpreter for the description. The result of running the interpreter is the most probable interpretation of the description—which forms the new description of the layer "n+1". All the layers (including "n") also contain a compiler. Unless the layer definition is overridden by specialization, the compiler in each layer is identical and provides the implementation with a theorem prover that compiles an interpreter from a description. The compiler runs at the meta level in layer "n" and uses the knowledge of the world at layer "n+1" which resides in level "n". It compiles the description from level "n+1" taking in to account what is known at the time about level "n+1" in the *knowledge* part of layer "n". The compilation of the description is a new interpreter at layer "n+1".

Below we describe the meta-interpreter for layers in GRAVA.

The interpret method is the primary driver of computation in the reflective architecture. The reflective levels are determined by the program designer. In order for the self-adaptive program to "understand" its own computational structure, each layer describes the layer beneath it. In self-adapting, the architecture essentially searches a tree of meta-levels. This is

best understood by working through the details of the architecture.

In the simplest of situations the top level application of "interpret" to the top layer results in the recursive descent of "interpret" through the reflective layers finally yielding a result in the form of an interpretation. Along the way however unexpected situations may arise that cause the program to need to adapt. Adaptation is handled by taking the following steps:

1. Reflect up to the next higher layer (parent level) with an object that describes the reason for reflecting up. It is necessary to reflect up because the higher level is the level that "understands" what the program was doing. Each level "understands" what the level directly beneath it is doing.

2. The world model (knowledge) that is maintained by the parent level is updated to account for what has been learned about the state of the world from running the lower level to this point.

3. Given updated knowledge about the state of the world the lower level is re-synthesized. The lower level is then re-invoked.

Armed with that conceptual overview of the interpret procedure we now explain the default interpret method.

```
1:(define (interpret
           ReflectiveLayer|layer)
2:  (withSlots (interpreter
                description
                lowerlayer) layer
3:    (if (null? interpreter)
4:      description
5:      (begin
         (elaborate layer))
6:          (reflectProtect
             (interpret
              lowerlayer)
7:           (lambda (layer gripe)
              (adapt layer gripe))))))

8:(define (reflectionHandler
           ReflectiveLayer|layer
           gripe)
9:  (adapt layer gripe))
```

Line 3 checks to see if the layer contains an interpreter. If it does not the result of evaluation is simply the description which is returned in line 4. This occurs when the lowest level has been reached.

If there is an interpreter, the elaborate method is invoked (line 5). "elaborate" (described below) constructs the next lower reflective layer.

"reflectProtect" in line 6 is a macro that hides some of the mechanism involved with handling reflection operations.

(reflectProtect *form handler*) evaluates *form* and returns the result of that evaluation. If during the evaluation of *form* a reflection operation occurs the *handler* is applied to the layer and the gripe object provided by the call to reflectUp. If the handler is not specified in the reflectProtect macro the generic procedure reflectionHandler is used. The invocation of the reflection handler is not within the scope of the reflectProtect so if it calls (reflectUp ...) the reflection operation will be caught at the next higher level. If reflectUp is called and there is no extant reflectProtect the debugger is entered. Therefore if the top layer invokes reflectUp the program lands in the debugger.

When the reflection handler has been evaluated the reflectProtect re-evaluates the *form* thereby making a loop. Line 7 is included here to aid in description. It is omitted in the real code allowing the reflectionHandler method to be invoked. The handler takes care of updating the world model based on the information in *gripe* and then adapts the lower layer. The handler therefore attempts to self adapt to accommodate the new knowledge about the state of the world until success is achieved. If the attempt to adapt is finally unable to produce a viable lower level interpreter it invokes reflectUp and causes the meta level interpretation level to attend to the situation.

```
1:(define (elaborate
           ReflectiveLayer|layer)
2:  (withSlots (lowerlayer) layer
3:    (let ((interpretation
            (execute layer))
4:          (llint (compile
                    layer
                    interpretation)))
5:      (set! lowerlayer
            ((newLayerConstructor
              layer)
6:            higherlayer: layer
7:            description: interpretation
8:            interpreter: llint)))))
```

The purpose of the elaborate layer is to build the initial version of the subordinate layer. It does this in three steps:

1. Evaluate the interpreter of the layer in order to "interpret" the layer's description. The interpretation of $layer_n$ is the description of $layer_{n+1}$.

   Line 3 invokes the interpreter for *layer* with *(execute layer)*. This simply runs the MDL agent interpreter function defined for this layer. The result of executing the interpreter is an interpretation in the form of a description.

2. Compile the layer. This involves the collection of appropriate agents to interpret the description of the lower layer.

Line 4 compiles the new layer's interpreter. Layer n contains knowledge of the agents that can be used to interpret the description of layer $n + 1$. The description generated in line 3 is compiled into an interpreter program using knowledge of agents that can interpret that description.

3. A new layer object is instantiated with the interpretation resulting from (1) as the description and the interpreter resulting from *compile* in step (2) as the interpreter. The new layer is wired in to the structure with the bi-directional pointers (lowerlayer and higherlayer).

In line 5, (newLayerConstructor layer) returns the constructor procedure for the subordinate layer.

The adapt method updates the world state knowledge and then recompiles the interpreter for the lower layer.

```
1:(define (adapt
          ReflectiveLayer|layer
          gripe)
2:   (withSlots (updateKnowledge) gripe
3:     (updateKnowledge layer))
        ;; update the belief state.
4:   (withSlots (lowerlayer) layer
5:     (withSlots (interpreter) lowerlayer
6:       (set! interpreter
              (compile layer)))))
```

The representation of world state is problem dependent and is not governed by the reflective architecture. In each layer the world state at the corresponding meta level is maintained in the variable "knowledge". When an interpreter causes adaptation with a reflectUp operation an update procedure is loaded into the "gripe" object. Line 3 invokes the update procedure on the layer to cause the world state representation to be updated.

Line 6 recompiles the interpreter for the lower layer. Because the world state has changed the affected interpreter should be compiled differently than when the interpreter was first elaborated.

```
1:(define (execute
          ReflectiveLayer|layer)
2:   (withSlots (description
                 interpreter
                 knowledge) layer
3:     (run interpreter
          description
          knowledge)))
```

2.3   Protocol for Interpreters

An interpreter is a special kind of computational agent that contains agents which it sequences. To support

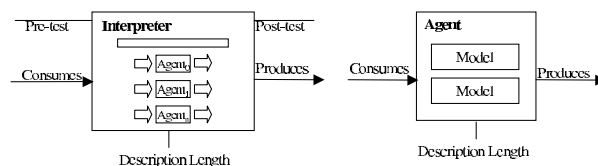those activities the interpreters support a protocol for meta-information shown in Figure 5(left).

Figure 5. Protocols for Interpreter and Agent Meta-Information

1. *(pretest anInterpreter anInput)* – Returns *true* if the input is suitable for the interpreter and *false* otherwise.

2. *(posttest anInterpreter anOutput)* – Returns *true* if the output is acceptable and *false* otherwise.

3. *(descriptionLength anInterpreter anInput)* – Returns the description length of the interpreter. The description length is $-log_2(P(success))$ where $P(success)$ is the probability that the interpreter will successfully interpret the scene.

2.4   Protocol for Agents

In order for agents to be selected and connected together by the theorem prover/compiler they must advertise their semantics. The purpose of the compiler is to select appropriate agents and connect them together to form a program. To support those activities the agents support a protocol for meta-information shown in Figure 5(right).

1. *(consumes anAgent)* – Returns a list of types that the interpreter expects as input.

2. *(produces anAgent)* – Returns a list of types that the interpreter produces as output.

3. *(descriptionLength anAgent)* – Returns the description length of the agent. The description length is $-log_2(P(correct))$ where $P(correct)$ is the probability that the agent will diagnose the feature in the same way as the specification.

3   Conclusion

The GRAVA software architecture is a prototype of a new approach to software development, which we call Self Adaptive Software. It promises to be an excellent approach to producing more robust, self-checking systems. The general Self Adaptive approach is clearly applicable to more than just interpretation problems.

The GRAVA architecture has been successfully applied to a number of problems including the aerial

image interpretation problem discussed in this paper and a person/face recognition and tracking project (ongoing) [13, 14]. Although the architecture doesn't depend upon corpus based methods the problem of generating the large number of models required for such a system to operate robustly makes corpus based systems particularly attractive.

We have developed methods for automatically inducing contexts from annotated corpora [12].

Since contexts are not random but are structurally related, transitions between contexts can be modeled as hidden Markov models (HMM) [16, 1]. We are currently extending the architecture described in the paper to use HMM reasoning to optimize the context switching mechanism. We will present the results of this research at a later date.

Although GRAVA was developed as an architecture for building robust vision programs and so far has only been applied to vision problems there is no reason why, in principle, that GRAVA could not be applied to other interpretations including the interpretation of speech and natural language.

References

[1] L.E. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of a markov process. *Inequalities*, 3:1–8, 1972.

[2] M. Clowes. On seeing things. *Artificial Intelligence*, 2:79–116, 1971.

[3] B. Draper, R. Collins, J. Brolio, A. Hansen, and E. Riseman. The schema system. Technical Report COINS TR88-76, Computer and Information Science, Univ. Massachusetts at Amherst, 1988.

[4] S. 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.

[5] IEEE. Ieee standard for the scheme programming language. IEEE Standard 1178-1990, IEEE Pisscaataway, 1991.

[6] G. Kiczales, J. des Rivieres, and G. Daniel. *The art of the Metaobject Protocol*. MIT Press, 1993.

[7] R. Laddaga and P. Robertson. *Yolambda Reference Manual*. Dynamic Object Language Labs, Inc., 1996.

[8] P. Maes and D. Nardi. *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[9] M. Minsky. A framework for representing knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, New York, 1975.

[10] A. Paepcke. Pclos: Stress testing clos experiencing the metaobject protocol. In *ECOOP/OOPSLA '90 Proceedings*, 1990.

[11] P. Robertson. *A Self-Adaptive Architecture for Image Understanding*. PhD thesis, University of Oxford, 2001.

[12] P. Robertson and R. Laddaga. Principle component decomposition for automatic context induction. In *Proceedings Artificial and Computational Intelligence, Tokyo 2002*. ACI, 2002.

[13] P. Robertson and R. Laddaga. A self-adaptive architecture and its aplication to robust face identification. In *PRICAI 2002, Trends in Artificial Intelligence*, pages 542–551. Springer Verlag, LNAI 2417, 2002.

[14] P. Robertson and R. Laddaga. An agent architecture for information fusion and its application to robust face identification. In *Proceedings of the 21st International Conference on Applied Informatics, Innsbruck Austria*, pages 132–139, 2003.

[15] B.C. Smith. Reflection and semantics in lisp. In *Proceedings 11th Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah*, pages 23–35, January 1984.

[16] A.J. Viterbi. Error bounds for convolution codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, 13:260–269, 1967.