Confidence From Self Knowledge and Domain Knowledge

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

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

Abstract. The GRAVA architecture supports building self-adaptive applications. An overview of the GRAVA architecture, its agent language and its reflective protocol are presented with illustrations from the aerial image interpretation domain.
Keywords: Aerial Image Analysis, Reflection, Code Synthesis, Agent Architecture.

1 Introduction

GRAVA is an architecture for building self-adaptive applications. In this paper we give a overview of the architecture and its protocols. The architecture is designed around an agent language embedded within a reflective architectural framework.

Autonomous agents are expected to operate in a decentralized manner without the intervention of a central control mechanism. This involves distributed algorithms for selecting which agents to run and when as well as dividing resources among the agents.

One approach to the agent selection problem that has been the focus of considerable attention, is the notion of a market based approach. The idea is that when an agent wishes to delegate a subtask to another agent capable of performing the subtask agents that are candidates to perform the subtask compete by bidding a price. This often works well, producing efficient solutions. However, two problems arise in such systems:

1. Selecting an appropriate basis for cost computations so that the bidding is fair.
2. Because the bidding is piecewise local, such systems are prone to find local minima and miss the global minima.
3. Agents not designed to work together can behave incoherently resulting in thrashing behavior. Ultimately multi-agent systems tend to work well only as a result of excruciatingly careful design. This makes implementing multi-agent systems a very complex and error prone programming exercise.

Our approach addresses these problems as follows:

1. The basis for cost computation is description length. Description length is the correct measurement in an interpretation problem because it captures the notion of likelihood directly: $DL = -log_2(P)$.
2. Monte Carlo sampling allows us to avoid the problem of finding unwanted local minima.
3. The problem of incoherent agents is addressed by dividing agents into contexts and then using reflection and self-adaptation to select an appropriate set of agents that are suited to the current state and which provably leads to a solution.

1.1 The Role of Reflection

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

1.2 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.

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.

Large software projects, especially software projects of defense contrators, 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.

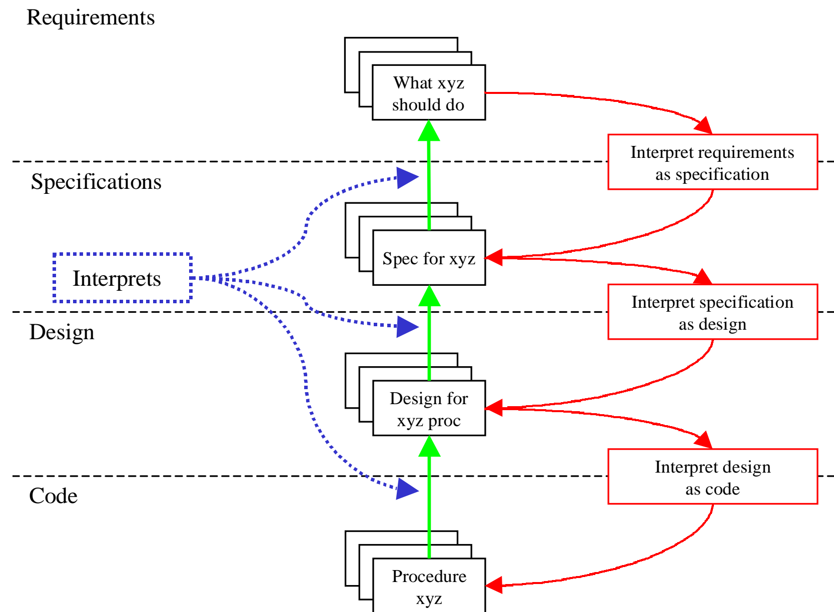

Fig. 1. Example of the relationship between levels of interpretation

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 1 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).

2 Objects in the GRAVA Architecture

The architecture is built from a small number of objects: Models; Agents; Interpreters; Reflective Levels; and Descriptions.

All of these terms are commonly used in the literature to mean a wide range of things. In the GRAVA architecture they have very specific meanings. Below, we

describe what these objects are and how they cooperate to solve an interpretation problem.

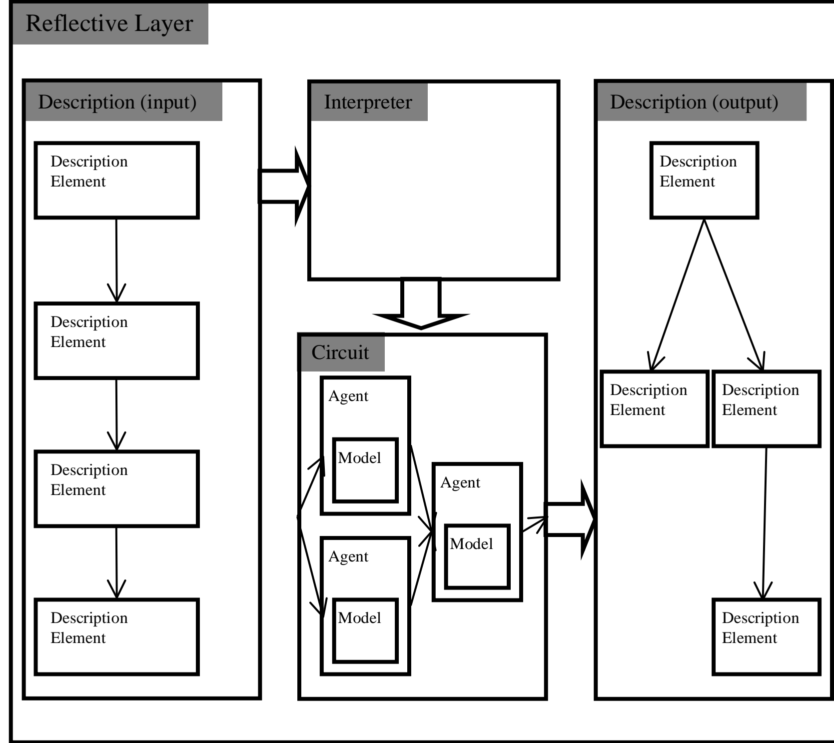

Fig. 2. Objects in the GRAVA Architecture

Figure 2 shows the objects that make up the architecture. A reflective layer takes an input description Δ_{in} and produces an output description Δ_{out} as its result. A description consists of a collection of description elements $< \epsilon_1, \epsilon_2, ..., \epsilon_n >$. The output description is an interpretation ($I \in Q(\Delta_{in})$) of the input where $Q(x)$ is the set of all possible interpretations of x.

$$\Delta_{out} = I(\Delta_{in}) \tag{1}$$

The goal of a layer is to find the best interpretation I_{best} which is defined as the interpretation that minimizes the global description length.

$$arg \min_{I_{best}} DL(I_{best}(\Delta_{in})) \tag{2}$$

The interpretation function of the layer consists of an interpretation driver and a collection of connected agents. The interpretation driver deals with the

formatting peculiarities of the input description (the input description may be an array of pixels or a symbolic description). The program is made from a collection of agents wired together. The program defines how the input will be interpreted. The job of the interpreter/program is to find the most probable interpretation of the input description and to produce an output description that represents that interpretation.

The GRAVA architecture allows for multiple layers to exist in a program and there are [reflective] links between the layers.

Below, we describe in greater detail the purpose, protocol, and implementation of the objects depicted in Figure 2. We maintain a dual thread in the following. On the one hand, we describe the GRAVA architecture abstractly, on the other, we also describe the actual implementation that we have developed.

Description A description Δ consists of a set of description elements ϵ.

$$\Delta =< \epsilon_1, \epsilon_2, ..., \epsilon_n > \tag{3}$$

Agents produce descriptions that consist of a number of descriptive elements. The descriptive elements provide access to the model, parameters, and the description length of the descriptive element. For example, a description element for a face might include a deformable face model and a list of parameters that deform the model face so that it fits the face in the image. A description element is a model/parameters pair.

The description length must be computed before the element is attached to the description because the agent must compete on the basis of description length to have the descriptive element included. It makes sense therefore to cache the description length in the descriptive element.

The description class implements the iterator:

```
(for Description|des fcn)
```

This applies the function "fcn" to every element of the structural description, and this enables the architecture to compute the global description length:

$$DL(\Delta_{out}) = \sum_{i=1}^{n} DL(\epsilon_i) \tag{4}$$

To get a better fix on notation, this is implemented as:

```
(define (globalDescriptionLength Description|des)
  (let ((dl 0))
    (loop for de in des
      (set! dl (+ dl (descriptionLength de))))))
```

DescriptionElements Description elements are produced by *agents* that *fit models* to the input.

Description elements may be implemented in any way that is convenient or natural for the problem domain. However the following protocol must be implemented for the elements of the description:

```
(agent <Element>)
```

Returns the agent that fitted the model to the input.

```
(model <Element>)
```

Returns the model object that the element represents.

```
(parameters <Element>)
```

Returns the parameter block that parameterizes the model.

```
(descriptionLength <Element>)
```

Returns the description length in bits of the description element.

Implementations of description elements must inherit the class DescriptionElement and implement the methods "agent", "model", "parameters", and "descriptionLength".

For readability we print description elements as a list:

```
(<model name> . <parameter list>)
```

Models Fitting a model to the input can involve a direct match but usually involves a set of parameters.

Consider as input, the string:

```
``t h r e e   b l i n d   m i c e''
```

We can interpret the string as words. In order to do so, the interpreter must apply word models to the input in order to produce the description. If we have word models for "three", "blind", and "mice" the interpreter can use those models to produce the output description:

```
((three) (blind) (mice))
```

The models are parameterless in this example. Alternatively we could have had a model called "word" that is parameterized by the word in question:

```
((word three) (word blind) (word mice))
```

In the first case there is one model for each word. In the case of "three" there is an agent that contains code that looks for "t", "h", "r", "e", and "e" and returns the description element "(three)". In the second case there is one model for words that is parameterized by the actual word. The agent may have a database of words and try to match the input to words in its database.

Consider the two examples above. If the probability of finding a word is 0.9 and the probability of the word being "three" is 0.001 the code length of "(word three)" is given by:

$$DL(wordthree) = DL(word) + DL(three) = -log_2(p(word)) - log_2(p(three)) \tag{5}$$
$$= -log_2(0.9) - log_2(0.001) = 0.1520 + 9.9658 = 10.1178 bits \tag{6}$$

The second approach, in which a separate agent identifies individual words would produce a description like "(three)". The model is "three" and there are no parameters. The likelihood of "three" occurring is 0.001 so the description length is given by:

$$DL(three) = -log_2(p(three)) = -log_2(0.9 * 0.001) = 10.1178 bits \tag{7}$$

That is, the choice of parameterized vs. unparameterized doesn't affect the description length. Description lengths are governed by the probabilities of the problem domain. This allows description lengths produced by different agents to be compared as long as they make good estimates of description length.

For a more realistic example, consider the case of a principle component analysis (PCA) model of a face [1]. A PCA face model is produced as follows. First a number n of key points on a face are identified as are their occurrences on all of the images. The shape of the face ψ_i is defined by a vector containing the n points. A mean shape is produced by finding the average position of each point from a set of example face shapes.

$$\bar{\psi} = \frac{1}{n} \sum_{i=1}^{n} \psi_i \tag{8}$$

The difference of each face from the mean face $\bar{\psi}$ is given by:

$$\delta\psi_i = \psi_i - \bar{\psi} \tag{9}$$

The covariance matrix S then is given by:

$$S = \sum_{i=1}^{n} \delta\psi_i \delta\psi_i^T \tag{10}$$

The eigenvectors p_k and the corresponding eigenvalues λ_k of the covariance matrix S are calculated. The eigenvectors are sorted in descending order of their eigenvalues. If there are N images, the number of eigenvectors to explain the

totality of nN points is N, typically large. However, much of the variation is due to noise, so that $p << N$ eigenvectors suffices to account for (say) 95% of the variance. The most significant of the eigenvector-eigenvalue pairs are selected as the principal components.

The resulting face model consists of a mean face shape $\bar{\psi}$ and a set of eigenvectors and weights such that any face shape ψ_p can be approximated by:

$$\psi_p = \bar{\psi} + \mathbf{Pb}, \tag{11}$$

where $\mathbf{P}$ is the vector of eigenvectors and $\mathbf{b}$ is the vector of weights. The weights are a measure of how much the model must be distorted in order to match the face ψ_p.

The above formulation of a face shape model describes a parameterized model. The weights are the parameters and the mean shape and vector of eigenvectors is the model. Algorithms exist for fitting such shape models to data. These algorithms first identify a key component and then, using the mean shape model, search for the other features (often edges) near the place where the mean suggests it should be. When the feature is found, its actual location is used to define a distance from the mean. This is repeated for feature points in the model. A set of weights is calculated which represents the parameterization of the model.

Agents The primary purpose of an agent is to fit a model to its input and produce a description element that captures the model and any parameterization of the model.

We implemented the atomic computational elements in GRAVA as agents. The system manipulates agents and builds programs from them but does not go beneath the level of the agent itself. The agent allows conventional image processing primitives to be included in the GRAVA application simply by providing the GRAVA agent protocol. We might have used methods if we were building a language rather than an architecture. GRAVA agents are not autonomous agents. They depend upon other agents to reason about them and to connect them together to make programs.

An agent is a computational unit that has the following properties:

1. It contains code which is the implementation of an algorithm that fits its model to the input in order to produce its output description.
2. It contains one or more models [explicitly or implicitly] that it attempts to fit to the input.
3. It contains support for a variety of services required of agents such as the ability to estimate description lengths for the descriptions that it produces.

An agent is implemented in GRAVA as the class "Agent". New agents are defined by subclassing "Agent". Runtime agents are instances of the appropriate Agent class. Generally Agents are instantiated with one or more models.

The protocol for agents includes the method "fit" that invokes the agent's model fitting algorithm to attempt to fit one or more of its models to the current data.

```
(fit anAgent data)
```

The "fit" method returns a (possibly null) list of description elements that the agent has managed to fit to the data. The interpreter may apply many agents to the same data. The list of possible model fits from all applicable agents is concatenated to produce the candidate list from which a Monte Carlo selection is performed.

Interpreters An *interpreter* is a *program* that applies *agents* in order to produce a structural description output from a structural description input.

A scene interpretation program may include agents for face recognition—such as the PCA face shape agent described above—and may include other agents that recognize other things that would be found in an image such as trees, buildings, and roads. The interpreter could be hand-assembled or it could be generated.

Monte Carlo Agent Selection A recurring issue in multi-agent systems is the basis for cooperation among the agents. Some systems assume benevolent agents where an agent will always help if it can. Some systems implement selfish agents that only help if there is something in it for them. In some cases the selfish cooperation is quantified with a pseudo market system.

Our approach to agent cooperation involves having agents compete to assert their interpretation. If one agent produces a description that allows another agent to further reduce the description length so that the global description length is minimized, the agents *appear* to have cooperated. Locally, the agents compete to reduce the description length of the image description. The algorithm used to resolve agent conflicts guarantees convergence towards a global MDL thus ensuring that agent cooperation "emerges" from agent competition. The MDL approach guarantees convergence towards the most probable interpretation but it does not guarantee that the most probable interpretation will be found.

When all applicable agents have been applied to the input data the resulting lists of candidate description elements is concatenated to produce the candidate list.

The *monteCarloSelect* method chooses one description element at random from the candidate list. The random selection is weighted by the probability of the description element.

$$P_{elem} = 2^{-DL(elem)} \tag{12}$$

So, for example, if among the candidates, one has a description length of 1 bit and one has a description length of two bits, the probabilities of those description lengths is 0.5 and 0.25 respectively. The *monteCarloSelect* method would select the one bit description twice as often as the two bit description.

The monteCarloSelect algorithm is given below:

```
(define (probability DescriptionElement|de)
  (expt 2.0 (- (descriptionLength de))))
```

```
(define (monteCarloSelect choices)
  (callWithCurrentContinuation
    (lambda (return)
      (let* ((sum (apply + (map probability choices)))
             (rsel (frandom sum)))
        (dolist (choice choices)
          (set! rsel (- rsel (probability choice)))
          (if (<= rsel 0.0) (return choice)))))))
```

3 Reflective Interpreter for Self-Adaptation

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.

A system can have an arbitrary number of levels. The example described in the introduction (Figure 1) has four levels. Most systems will have a small number of levels. Experience to date suggests that three levels is usually sufficient.

Each layer can reflect up to the layer above it in order to self-adapt.

```
(defineClass ReflectiveLayer
  ((description) ;; the (input) description for this layer
   (interpreter) ;; the interpreter for the description of this layer
   (knowledge)   ;; a representation of world knowledge at this level
   (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. In the software development example the requirements level would contain a description of the requirement that is to be interpreted by the layer.
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. The knowledge gets updated as the subordinate layer attempts to interpret its description. The knowledge is used in the synthesis of the interpreter for the subordinate layer.
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.

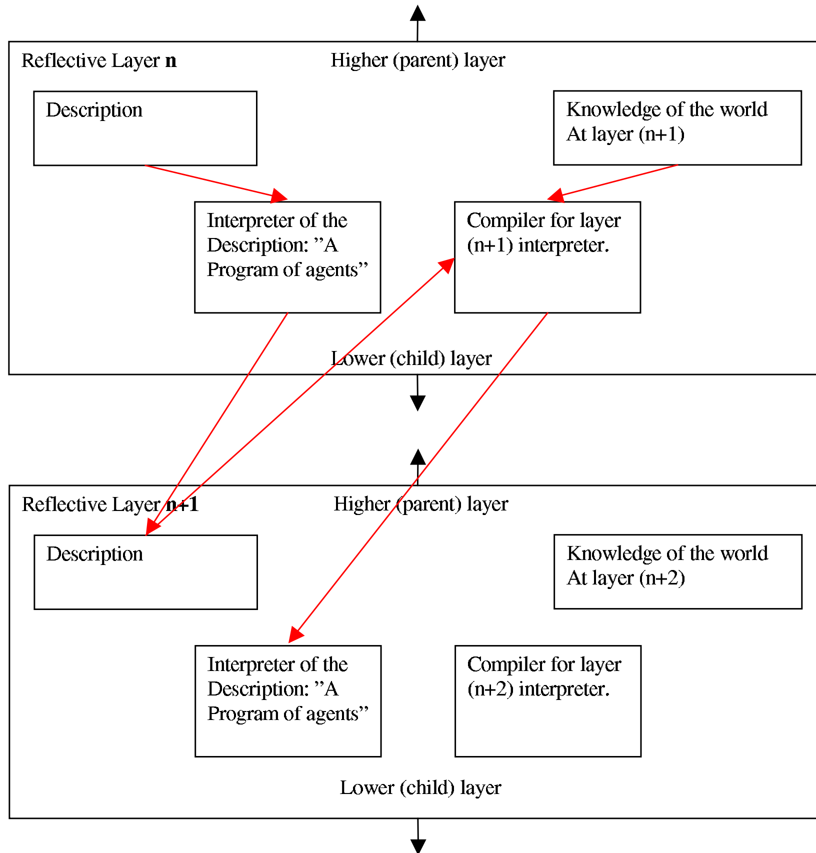

Fig. 3. Meta-Knowledge and Compilation

Figure 3 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. That interpreter is run. The result of running the interpreter is the most probable interpretation of the description—which forms the new description of the layer "n+1". Layer "n" also contains a compiler. Actually all layers contain a compiler. Unless the layer definition is overridden by specialization the compiler in each layer is identical and provides the implementation as 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 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.

The top level layer is manually constructed by the program designer. It must be because there is no higher level to defer to. That level defines its goal in the form of a description that must be interpreted. The collection of agents that interpret that description are provided by the human programmer. The program that those agents constitute is charged with the responsibility of producing the reflective layer immediately below. The lower level, once constructed, is then interpreted in order to bring about the desired behavior.

At some point the layers bottom out in a lowest level below which there is no further elaboration of layers. The lowest level layer has a description but no interpreter. The description at the lowest level is the result of the top level application of “interpret”.

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 this far.
3. Armed with updated knowledge about the state of the world the lower level is re-synthesized. The lower level is then re-invoked.

We now explain the default interpret method.

```
1:(define (interpret ReflectiveLayer|layer)
2:  (withSlots (interpreter description lowerlayer) layer
3:    (if (null? interpreter)
4:       description        ;; Return the description
5:       (begin
6:         (elaborate layer);; Create/populate subordinate layer.
7:         (reflectProtect (interpret lowerlayer)
8:           (lambda (layer gripe) (adapt layer gripe)))))))
```

```
9:(define (reflectionHandler ReflectiveLayer|layer gripe)
10: (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 6). “elaborate” (described below) constructs the next lower reflective layer.

“reflectProtect” in line 7 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 is be caught at the next higher level. If reflectUp is called and there is no extant reflectProtect the debugger is entered. So 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 8 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)) ;; ll description
4:          (llint (compile layer interpretation)))
5:      (set! lowerlayer ((newLayerConstructor layer)
6:                        higherlayer: layer
7:                        description: interpretation
8:                        interpreter: llint)))))
```

The purpose of the elaborate method 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)))
```

4 Program Synthesis

The purpose of the "compile" method described in Section 3 is to produce a collection of interpreters connected together that performs the function of interpreting the description that belongs to the layer. In this section, we explore the idea of compilation-as-proof, and then use the idea in our self-adaptive architecture.

An interpreter (as described above) is a coherent computational unit. It includes code for sequencing agent's over the input description and making Monte

Carlo samples of the agents results. An interpreter therefore contains a collection of agents that the interpreter controls. In this section we develop the protocol for interpreters that permits them to be automatically selected, sequenced, and populated with agents by the compiler.

4.1 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 so although such an exercise could be of interest in its own right, 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 leaves of the proof tree.

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

1. Knowledge of rules of inference and a procedure for 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 view, the compiler produces a tree-structured proof. The leaves of the proof are blocks of machine code. The machine codes are read off the fringe of the proof tree to produce the target machine language representation.

Consider the problem of interpreting a piece of source code as an assembly language program. Models of how to represent a source program fragment in assembly language can be applied in order to interpret the source code as assembly language.

Using the GRAVA agent/interpreter architecture the interpreter would sequence parts of the language over a collection of agents that can deal with the parts of the language. The interpreter part therefore is a code walker and the agents of the interpreter are the collection of agents that deal with each syntactic construct in the language.

Consider the problem of interpreting the expression **C=A+B**.

Compiling that expression involves proving that the expression is a part of the language. The proof looks like this:

1. C=A+B is an assignment (rule: assign-1) where the location is C and the expression is A+B.
2. C is a location (rule: location-1).
3. A+B is an addition (rule: addition-1) where A is a sub-expression and B is a sub-expression.
4. A is a de-reference (rule: dereference-1).
5. B is a de-reference (rule: dereference-1).

Agent: assign-1	
Consumes:	(A1=(location X) A2=(expression Y))
Produces:	(A3=(expression X=Y))
Fit:	(define (fit assign-1|agent) (withTemporaryRegister (R1) (let ((L1 (ask A1 `(location ,X)) (E1 (ask A2 `(expression ,Y ,R1)))) (tell A3 (list L1 E1 (make store-1 val: R1 loc: L1))))))
Model: store-1	
	(define (emit store-1|mod output) (withSlots (val loc) mod (assemble output 'STORE val loc))

Fig. 4. Agent and Model for Assignment

The simplest way of implementing this example is to treat it as a one step problem of interpreting the source expression as a sequence of machine instructions. However, in order to use this example to illustrate the components of the reflective architecture we develop this example here as a two step process. For readability and to avoid getting bogged down in unnecessary details we illustrate the essential components using pseudo code. Figure 4 shows the relevant portions of the assignment agent and its associated model.

The assignment agent is connected to two agents A1 and A2 from which is asks about the location of the left hand side (LHS) and the value of the expression. It tells its result to agent A3. It fits the model "store-1" which supports an "emit" method that assembles instructions to an instruction stream. Figures 5 and 6 show implementation templates for addition and de-reference respectively which are required for this example.

The proof tree is shown in Figure 7.

The resulting description that results from running the agents is shown below.

```
((location-of C) ; from assign-1
 ((location-of C)) ; from location-1
 ((register tmp1) ; from addition-1
  (deref-1 tmp1 (location-of A)) ; emits (LOAD tmp1 (location-of A))
  (deref-1 tmp2 (location-of B)) ; emits (LOAD tmp2 (location-of B))
  (add-1 tmp1 tmp2))             ; emits (ADD tmp1 tmp2)
 (store-1 tmp1 (location-of C)))); emits (STORE tmp1 (location-of C))
```

Agent: addition-1	
Consumes:	(A1=(expression X) A2=(expression Y))
Produces:	(A4=(expression X+Y) A3=(result))
Fit:	(define (fit addition-1|agent) (withTemporaryRegister (R1) (let ((R2 (ask A3 `(result)))           (E1 (ask A1 `(expression ,X ,R1))) (E2 (ask A2 `(expression ,Y ,R2)))) (tell A4 (list R2 E1 E2 (make add-1 val1: R1 val2: R2)))]
Model: add-1	

	(define (emit add-1|mod output)
	  (withSlots (val1 val2) mod
	    (assemble output 'ADD val1 val2)))

Fig. 5. Agent and Model for Addition

Running through the description in sequence applying the "emit" method on each entry results in the following instructions being assembled:

```
(LOAD tmp1 (location-of A))
(LOAD tmp2 (location-of B))
(ADD tmp1 tmp2)
(STORE tmp1 (location-of C))
```

In the example given above we included only a single agent for each operation (assign-1 location-1 addition-1 dereference-1). We could provide arbitrarily many agents for each operation in which case we would have to choose which one to select on the basis of the description length of the description elements.

A compiler is an interpretation program that interprets a 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.

Agent: dereference-1	
Consumes:	(A1=(location X))
Produces:	(A2=(expression X))
Fit:	(define (fit dereference-1|agent) (let ((R1 (ask A1 `(location ,X)))) (tell A2 (list R1 (make load-1 reg: R1 loc: L1)))))
Model: deref-1	
	(define (emit deref-1|mod output) (withSlots (reg loc) mod (assemble output 'LOAD reg loc)))

Fig. 6. Agent and Model for Dereference

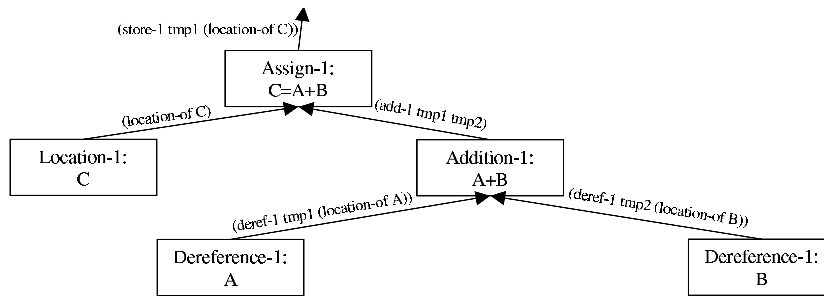

Fig. 7. Example Proof Tree for Compilation Example

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.

The purpose of the above example was to motivate the idea that compilation can usefully be viewed as a theorem proving activity. By adding “produces” and “consumes” to the protocol for agents and interpreters we can treat them as rules of inference and use a theorem prover to connect them up onto programs—just as we did above.

In compiling a program into machine code, we generally deal with certainty. The language that is being compiled is not ambiguous and the machine code can be relied upon to perform as expected. Computers are designed to operate reliably and high level languages are designed to be unambiguous.

The real world does not offer such certainty and programs that must interact with the real world inherit that uncertainty. 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. Conventional compilers are a special case of a more general problem.

5 Uncertain Information and MDL

Because the real world doesn't offer us the kind of guarantees that we have managed to build for ourselves in the form of closed world computers programs are brittle when they attempt to operate in an unconstrained environment such as the *real world*.

The theorem prover developed below is "relaxed" in that the theorems it produces are guaranteed to be programs that produce the desired effect but only if the program terminates. The program may not terminate because (reflectUp ...) may be invoked prior to completion. Since there may be many agents and interpreters that can be connected up to be a valid program that satisfies the representation from which it was compiled there are many different proofs. We wish to find the compilation/proof that has the highest likelihood of completing without invoking reflectUp.

$$DL(interpreter) = -log_2(1 - P(invokesReflectUp)) \tag{13}$$

For each interpreter the description length is determined by Equation 13. An individual agent can't invoke reflectUp because an individual agent may be unsuitable for a particular application and its failure is not cause to adapt the program. Instead another agent should do better. The interpreter then selects the agent that did best. It is the interpreter object that is in a position to invoke reflectUp.

We provide three ways for reflectUp to be invoked and cause self-adaptation of the program to occur:

1. An interpreter *pre-test* fails.
2. An interpreter *post-test* fails.
3. No agents are successful in interpreting part of the input that is being interpreted.

We add *pre-test* and *post-test* to the protocol for interpreters along with *produces* and *consumes*.

5.1 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 8.

1. *(pretest anInterpreter anInput)*

   Returns *true* if the input is suitable for the interpreter and *false* otherwise.

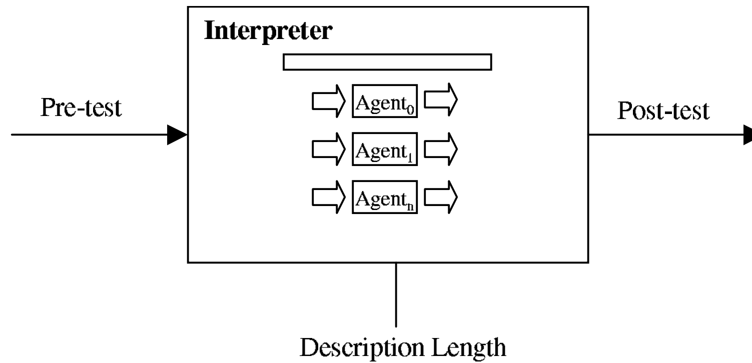

Fig. 8. Protocol for Interpreter Meta-Information

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.

5.2 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 9.

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.

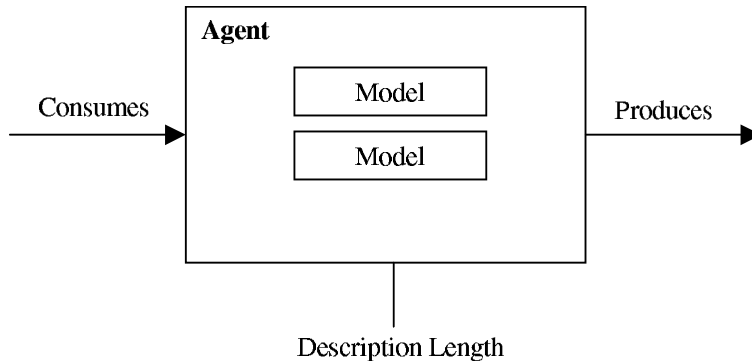

Fig. 9. Protocol for Agent Meta-Information

6 Conclusion

The GRAVA architecture is based on two ideas:

1. That for interpretation problems global MDL is the goal;
2. That global MDL can be approximated by Monte-Carlo sampling.

MDL provides a common currency or "gold standard" for use in market model agent systems. Because the architecture is specialized to solving interpretation problems there are problems for which it is not appropriate. Nevertheless a great many interesting problems can be cast in the form of interpretation problems.

Because the foundations upon which the architecture is built are well understood the behavior and performance of the architecture can yield to some level of analysis—such as convergence analysis. The architecture has some interesting characteristics:

1. Cooperation between agents at different semantic levels is an emergent property of the architecture.
2. Robustness within the limits of the programs domain is realized by virtue of measuring how local choices affect global description length.
3. There is an implicit information fusion model.

The third point is worthy of greater discussion. Most attempts at reasoning about uncertainty [4, 3, 2] attempt to bring together contributions so as to make a local decision. When local evidence is sufficient these methods work well but when the sources of evidence become less straightforward the approaches get bogged down. Numerous problems occur—most notably that required probabilities are often not available and that in order for the approaches to be tractable it is necessary to assume conditional independence. In any complex system the practical issues are immense.

The implicit information fusion model in the GRAVA architecture depends upon the effect that local decision has upon the global interpretation. A less

probable interpretation of a feature will be selected if it gives rise to a shorter global description.

Conventioal architectures such as Schemas, Blackboards, Rule Systems, and Subsumption all attempt to find a single path towards a solution and manual hard wiring of control information is one of the major problems with those approaches. In GRAVA we have chosen a method that avoids those problems at the cost of having to pursue multiple paths. When multiple agents are available we choose one based on a Monte-Carlo sample.

It can be objected that our approach is computationally expensive compared to conventional approaches. There are a number of important observations on this issue:

1. Conventional architectures search for the first solution that can be found. They hoped that by making local decisions a reasonably good solution will result. They do not attempt to find the *best* solution. For interpretation problems it is often important to find the best solution (or one very close to it).
2. Processing is cheap. It is reasonable to have multiple parallel computations occurring at once. It is reasonable to have multiple processors, perhaps as many as one per agent or one per model. These are problems that scale well. You really can have one processor per model and derive benefit from the parallelism. There are of course within the single processor view numerous opportunities to build optimization methods. For example for certain problems we can do better than using Monte-Carlo sampling.

There are a few comments that should be made regarding the use of Monte-Carlo sampling to estimate the global MDL description. Most uses of Monte-Carlo sampling attempt to estimate a PDF by counting frequencies of occurrences. For complex situations a great many samples would be required before anything like an accurate estimate of the PDF would result. In our use we are able to measure the global description length by adding up the description lengths of the components. Since we only want the most probable solution and not the whole PDF we can get by with far less iterations than would be required to estimate the PDF with any accuracy. As successive samples are taken we always know which is the best solution so far and can terminate the search at any point. This allows us to trade off interpretation accuracy against computational cost.

In some agent architectures an agent is a wholly autonomous entity that determines its own applicability and negotiates to participate in the ongoing computation. The agents described in this architecture are woven together into a program. The choice of which agents may participate and where has already been made when the program was constructed (at run time). Agents in GRAVA are computational units that support the protocol for agents. The way that the agents are woven together into a program is similar to the way that methods are woven together in CLOS. The MDL approach to agent selection described here should be equally effective in other agent architectures as long as they are solving interpretation problems.

The architecture described in this paper has the novel attribute that it knows what it knows and it knows how to apply what it knows. As with Smith's [5] reflective architecture each reflective layer implements the layer beneath it. In Smith's architecture each level of interpreter implemented the level of interpreter beneath it. In the GRAVA architecture each layer implements not only the interpreter but also the program beneath it. The inclusion of a compiler in the reflection loop allows the meta-interpreter to respond to the changing state of the environment by recompiling lower layers as appropriate. Each layer contains the knowledge of:

1. What the layer beneath is trying to do.
2. What the current belief state is as to the state of the world as it pertains to the operation of the layer beneath it.

When the belief state changes at some level the code for lower levels is automatically updated. If new agents are added they are automatically used where appropriate.

7 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. T.F. Cootes, G.J. Edwards, and C.J. Taylor. Active appearance models. In H. Burkhardt and B. Neumann, editors, *Proceedings, European Conference on Computer Vision 1998*, volume 2. Springer, 1998. pages 484-498.
2. A.P. Dempster. A generalization of bayesian inference. *Journal of the Royal Statistical Society, Series B*, 30:205–247, 1968.
3. W.B. Mann and T.O. Binford. Probabilities for bayesian networks. In *Proceedings Image Understanding Workshop*. Morgan Kaufman, San Francisco., 1994.
4. G. Shafer. *A Mathematiccal Theory of Evidence*. Prinston University Press, 1976.
5. 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.