# CONTROLLER DEVELOPMENT WITH A RAPID PROTOTYPING SHELL

PUSHKAR PIGGOTT AND PHILLIP J. MCKERROW[1]

***Abstract*—The advantages of rule-based techniques are well known, but traditional expert system shells are unsuited to controller development. In this paper we clarify the differences between the controller development task and traditional applications of expert system shells, and describe a shell we have developed for rapid-prototyping development of controllers. To deal with the differences the shell uses fuzzy inference, a single inference step per time step, a special architecture for symbol grounding, and vector variables. We describe the script language, and two example applications. One is to control a simulation of a truck backing up, the other is as part of a sonar-sensing mobile robot control system.**

***Keywords*—Control, expert system shell, fuzzy inference, robotics.**

***Source of Publication*—The Australian Journal of Intelligent Information Processing Systems, Vol. 4, No. 2 Winter 1997.**

## 1 INTRODUCTION

Fuzzy systems have shown themselves to be highly successful controllers in a wide variety of applications [21]. This success can be interpreted as the result of the well known benefits of rule-based development techniques, with fuzzy inference the key to applying them to controllers. We have therefore developed a *shell* for controller development that uses fuzzy inference as just one technique for overcoming the differences between controller development and more traditional applications of expert system shells. Our shell differs from existing fuzzy development environments in that it integrates easily into a control system to create a fuzzy relationship prototyping tool that minimizes the re-compilation step [17]. It also has a special syntax to simplify the definition of multiple fuzzy subsets.

The first half of the paper introduces the differences between controller development and more traditional applications of expert systems, and describes our shell. We present the script language syntax, with emphasis on the importance of script simplicity. The second part briefly describes two example applications. The first is the control of a simulated truck backing-up to a dock. The second uses two simpler rule-sets as part of a larger system for real-time control of a sonar-sensing mobile robot. Both are illustrations only, and use relatively simple control relationships. More complex examples where a prototyping shell would be useful are common in the literature (e.g., [19]).

## 2 RULE-BASED CONTROLLER DEVELOPMENT

### 2.1 MATHEMATICAL MODELLING AND RULES

The benefits of rules over mathematical modelling are frequently stated in the fuzzy literature (e.g., [21]) and are dealt with only briefly here. Conventional control system design requires the derivation of a mathematical model of the system to be controlled. Some generic analytical models are available, but in most cases a special-purpose model must be derived. This model must be based on a detailed knowledge of all the variables, and obtaining this is time consuming, and often impossible (e.g., [11]). Alternatively, a model may be derived by fitting curves to logged data and using linear regression and series approximation techniques, but this is also difficult. In practice, mathematical models are often inflexible and have to be adapted on-line to deal with parameter variations that are not captured by the equations. They are also hard to understand and maintain, and may be too complex to compute in real time.

Rule-based systems can be model-free [21]. The rule-set may be derived from a model or from rules-of-thumb, or developed through prototyping. By not implementing the model explicitly, the rule-set can be simple to compute and robust, and it may be effective where no mathematical model is possible. Rules are easier to understand than equations, simplifying both knowledge acquisition and rule-base maintenance.

1. Robotics Research Lab. University of Wollongong NSW 2522

Rule-based system design can also be simplified by the use of a *shell* to separate the representation of knowledge from the mechanism of inference. The shell incorporates an inference engine that performs inferences defined in a *shell script*. The script author can ignore the implementation details and focus on the structure of the knowledge.

In short, a rule-based system can be applicable where mathematical modelling is impossible or impractical, and the rule script is composed of linguistic terms that are readily comprehensible. Controller development differs from traditional symbolic applications of expert system shells in the following four significant ways, however.

- An expert system deals with discrete symbols, while a controller deals with *continuous domains*.
- A controller must respond in *real time*.
- The symbols in the script that defines a controller must be *grounded* to control variables.
- The *data structures* supported must be stream-lined so as not to compromise execution speed.

The following subsections discuss these differences, and how they affect our shell.

## 2.2 CONTINUOUS DOMAINS

A controller relates continuous domains. Boolean rules can take subranges as antecedents and yield discrete values as consequents, but the results are poor. As the match between such a rule-set and a control curve is improved, the number of rules increases rapidly towards infinity.

A traditional expert system is not restricted to rules. It can comprise both rules and arithmetic functions, and the consequence of a rule can be a function that derives an output from the inputs. Thus, rules can be used to map an assortment of curved patches to distinct regions of the control curve. This may be an improvement over mathematical modelling in some cases, for example where a complete model is not feasible, but with the use of functions it retains many of the disadvantages. It also introduces the problem of matching the edges of adjacent patches [14].

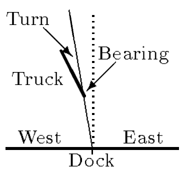
The inference method used by *fuzzy logic* [8] provides the alternative approach that we use. Fuzzy inference generates a continuous output by applying multiple rules to relate a set of inputs to an output. Each rule defines an output point, and the fuzzy inference mechanism generates a continuous curve by interpolating between them.

Each variable is represented by a fuzzy membership graph, with the variable's range on the *x* axis, and membership (μ) on the *y* axis (eg., *Bearing* in Figure 1). A *fuzzy subset* maps a subrange of the variable to membership values (eg., *West* and *East* in Figure 1). A rule takes subsets as its antecedents, and the strength of a rule changes as the inputs move across its subsets, changing their memberships.

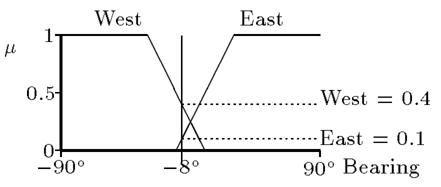
Antecedent terms are combined using *min* for *and* and *max* for *or*, and negation is the subtraction of the value from 1. A consequent of a rule is a subset whose weight is varied by the strength of the rule. The value of an output variable is determined by taking the *centre of gravity* of the subsets defined on its range. The system developer can fine-tune the relationship between input and output by adjusting the shapes of the input subsets and the weights of the output subsets.

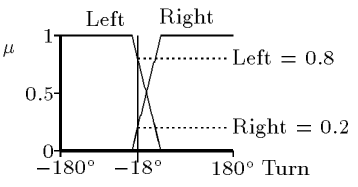
Figure 1 shows a simple example where four rules are used to control a truck backing up to a dock. It will be extended in the Section 5. The input variables are the truck's *Bearing* from the target point, and its *Turn* relative to the bearing. The figure shows the rules converting a *Bearing* of -8° and a *Turn* of -18° to an output *Steer* value of -10°. The output subsets are represented as shapeless weights for reasons discussed in Section 4.2.

Fuzzy controllers traditionally use a matrix representation for rules that requires a distinct rule for every combination of input subsets. This structure reflects the implementation and is not a very flexible format for rule-set design. It may also result in many unnecessary rules. By using a shell we allow each rule to sample an arbitrary selection of input variables and affect an arbitrary number of output variables. This flexibility, in conjunction with rapid compilation, allows the rule-set designer to focus on and test particular control relationships, and to gradually extend the rule-set, retaining only those variables, subsets and rules that demonstrate improved performance. The resulting economy minimizes controller time complexity for serial implementation without compromising the potential for parallel implementation.



**1:** if Bearing is West and Turn is Left then Steer is Left
**2:** if Bearing is West and Turn is Right then Steer is Right
**3:** if Bearing is East and Turn is Left then Steer is Left
**4:** if Bearing is East and Turn is Right then Steer is Right





**R1:** min(0.4,0.8) = 0.4, **R2:** min(0.4,0.2) = 0.2,
**R3:** min(0.2,0.8) = 0.2, **R4:** min(0.2,0.2) = 0.2

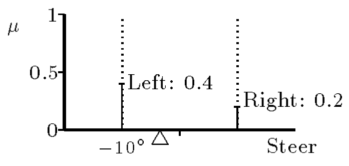


Figure 1: Fuzzy Inference Applied to a Truck Backing up to a Dock

## 2.3 REAL-TIME RESPONSE

The inference engine of an expert system is a *production system* that is obliged to follow chains of inference of unknown length, which may even fail to terminate. It is therefore unsuited to control because, to be safe, a control system must be deterministic and guarantee to meet hard deadlines. Time-sensitive production systems suitable for complex control are being developed [5].

For simpler controllers, the outcome of the complete rule-set can be equated with the firing of a single production rule. In a production system's chain of deductions, the firing of one rule changes the database and forces the re-application of the entire rule-set. Similarly, each computation of controller output from inputs is a step in a chain of computations as the controlled system changes its world towards a desired state. Such a controller responds directly to the world. It uses the world as its own model, as Brooks [3] has urged that a robot should.

It is appropriate, then, that a controller perform only a single step of inference at each time step, and our shell supports the development of such controllers. To make maximal use of its inputs, the controller should act on the combined result of all its rules, and fuzzy inference make this possible. The restriction on inference depth does not defeat the termination problem because the controlled system still may not reach its goal. It will continue to be responsive though, and therefore to act safely, as each inference step is guaranteed to complete within specific time limits.

A small amount of internal state can be maintained by *intermediate* variables that hold results over from one time step to the next, but planning is impossible. We do not deny its utility, but consider it a distinct issue that operates on a different time scale. The flexible architecture of our shell allows planners to be integrated with it.

## 2.4 SYMBOL GROUNDING

Traditional expert systems are symbolic. Their inputs and outputs are text strings that are interpreted by a user. The buffering effect of this interpretive stage has helped obscure their limitations for real-world applications. *Symbol grounding* is the name given to the grounding of the symbols manipulated in a computer system to objects (or at least perceptions) and actions in the real world. It is an area in which practical control applications have a large part to play [10].

The variable names used in a controller script are more than just names for control data, they name conceptual entities central to the model manipulated in the mind of the system developer. Symbol grounding is the creation of these entities, and they may not correspond directly to quantities offered by the hardware. The mapping between hardware control signals and script variables may require arbitrary computation, and the grounding code is therefore most conveniently developed in an established programming language.

The choice of appropriate control variables at an appropriate conceptual level is part of knowledge acquisition for control. Sugeno and Yasukawa [23] propose a fuzzy-logic-based approach to qualitative black-box modelling that provides us with a model of the fuzzy control equivalent to the knowledge acquisition process. Black box modelling involves identifying a model system that has the same input/output characteristics as the modelled system. Since nothing is known of the *actual* mechanism, the designer must propose one that exhibits the same behaviour.

Sugeno and Yasukawa divide model identification into sequential stages: *Structure Identification Stages I & II* and *Parameter Identification*. Figure 2 relates these stages to the structure of a control system using a shell. Structure Identification Stage I is the identification of the control variables, specifically the choice of input and output quantities effective for control. Input/output relations are identified in Stage II, and Parameter Identification tunes the parameters (vertex locations) of the fuzzy subsets.

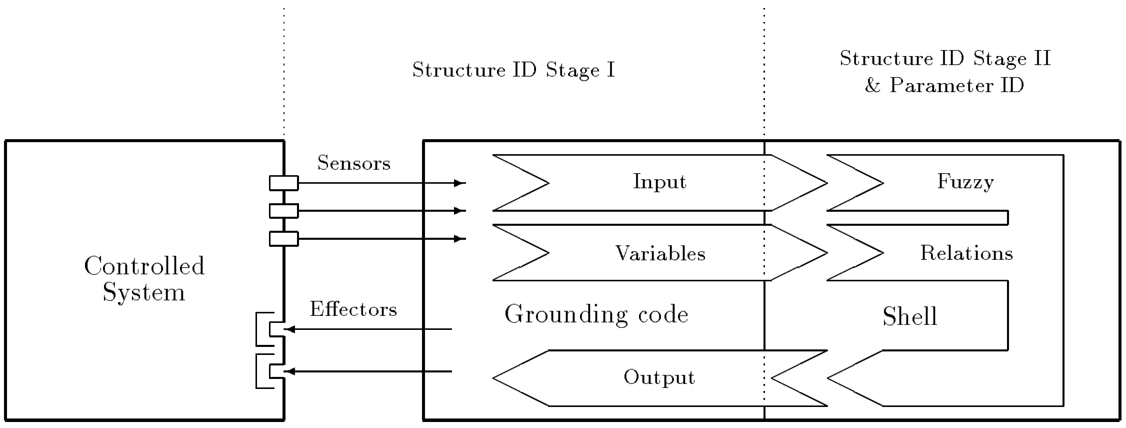


Figure 2: The Parts of a Fuzzy Control System and Stages of System Indentification

## 2.5 DATA STRUCTURE

Data structures are a powerful language tool but complex constructs such as the *frames* [12] common in knowledge representation risk a cost in both compilation and execution time. On the other hand, simple constructs that match the structure of the available data can reduce computation and considerably simplify the control script. Our shell compromises by implementing *vector variables* that allow subsets and rules that apply equally to every element to be defined just once. In the case of a sonar ring, for example, a single vector variable can represent all the sensors.

The number of elements may vary at run time. For example, if a robot system identifies plane wall segments for use as navigation beacons, then the number of segments currently detected may vary. Our shell handles this situation by having an arity count associated with the vector that can be set at each time-step.

## 3 THE CONTROLLER DEVELOPMENT TOOL

FLOPS [4], FEST [25] and LIFE FEShell [24] are expert system shells that have fuzzy extensions to provide continuous output. They rely on production systems for inference, and the resulting multi-step inferencing leads to substantial delays, as described above. Wood & Schneider [25], for example, report a 4.5 second time step for a FEST application, equivalent to a distance of 200 metres travelled by the controlled helicopter. We require a much tighter control loop to safely control autonomous robots.

A tool that supports the development of high performance controllers must allow the developer to create control variables grounded to control signals, and to define relationships between them. Grounding can be achieved with a conventional programming language, and it would be possible to extend such a language to include a syntax for defining the relationships. However, although these two tasks deal with the same variables, they are quite different conceptually, and it is preferable to use separate languages. Thus, a controller development system must incorporate two languages, and provide cross-language linking of corresponding variables.

Fuzzy-C[1] and Fide[2] are controller development environments of this sort. They are complex, using multiple script-file types, multiple editors and proprietary linkers. Fuzzy-C supports rule tuning using a simulation of the task, or from a remote machine during execution. Fide supports rule tuning only with a simulation.

Our shell is simpler because it is designed primarily as a prototyping tool, and because it exploits the system development task structure identified by Sugeno & Yasukawa. No special environment is required because it is a C++ object that uses the normal language mechanisms to integrate with the control system. We chose C++ because it provides the necessary object support and is also very suitable for writing low-level grounding code.

The control system developer identifies the control variables in Structure Identification Stage I, and uses C++ to develop the low-level grounding code that creates them from input and output signals. This code will not require much subsequent alteration. The shell object is linked in to form a complete prototyping control system that compiles and executes a script file of relationship definitions at run-time. No source-level re-compilation or re-linking is required, and the update-compile-test loop for Structure Identification Stage II and Parameter Identification is therefore fast and appropriate for rapid prototyping.

In its simplest form (Figure 3) the control system consists of an initialization section that initializes the controlled system and compiles the script, and an operating loop that generates control outputs from its inputs in discrete time steps. The shell has four methods with which the control system manages it. The initialization part calls **`variableIn`** or **`variableOut`** for each control variable to declare it to the shell, and then calls **`compile`** to compile the script. The arity and bounds are optional arguments to **`variable`**. At each iteration, the operating loop generates values for the input variables from the input signals, calls **`infer`** to update the output variables according to the fuzzy relationships, and uses them to generate the output signals.

A control system is not limited to this simple form. A complex system may require multiple sets of fuzzy relationships, applicable in different situations. It may also use the output variables of one script as input to higher-level relationships defined by another script.

## 4 THE SHELL SCRIPT

To ensure consistency, or in case the script and grounding code authors are not the same person, the control system passes on the variable declarations that were provided by the calls to **`variable`** to the script author as an initial script. The control system creates this initial script and exits if it cannot find the script file. The script author edits and develops the initial script and re-runs the control system to prototype it.

### 4.1 VARIABLES

A script variable definition consists of a declaration and zero or more trailing subset specifications. The declaration comprises all the arguments to **`variable`** except the data address. Figure 4 shows the script declarations generated from the variables exported to the shell in Figure 3. The ancillary words, such as **`is`**

1. Fuzzy-C is a trademark of Togai InfraLogic Inc.

2. Fide is a trademark of Aptronix Inc.

```
Sim sim;
Shell shell;
float bearing, turn, steer;
const float bounds[2] = {-90.0, 90.0};
int arity = 1;
shell.variableIn(“Bearing”, &bearing,
    &arity, bounds);
shell.variableIn(“Turn”, &turn);
shell.variableOut(“Steer”, &steer);
shell.compile(“steer.scr”);
while (!sim.finished()) {
    bearing= sim.bearing();
    turn= sim.turn();
    shell.infer();
    sim.step(steer);
}
```

Figure 3: Using the Shell’s Program Interface

and **`from`** are optional. The arity comes before the IO direction but it is optional, and omitted, if it is **`1`**. The IO direction declaration is optional because any variable not declared by the grounding code is assumed to be intermediate. The bounds can be specified by the grounding code or the script author. If given, they are used to check all subset vertices.

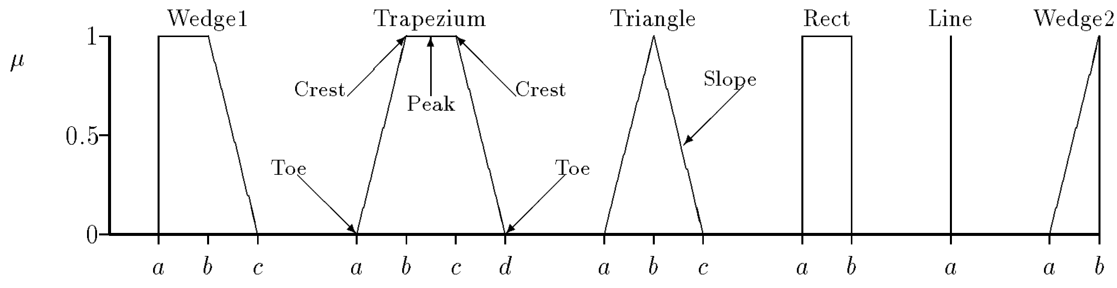
```
Bearing is input float from -90 to 90
Turn is input float
Steer is output float
```

Figure 4: The Declarations in the Initial Script

## 4.2 SUBSETS

The use of linguistic terms is one of fuzzy control’s prime claims to simplicity and ease of use. In practice, these terms must be tied numerically to subset shapes, and the potential for variety of shape is a major loophole where complexity can be re-introduced. The standardization and simplification of subset shapes is therefore an important area of research. Some work suggests Gaussian curves and other complex shapes (e.g., [1,20]), but practical real-time systems (e.g., [13]) commonly use trapezoids. Trapezoids are suitable for control because they lead to simple computations without appreciable degradation of performance.

A subset specification syntax should be simple and minimize the potential for confusion and error in fuzzy relation definition. Figure 5 illustrates the syntax we use for specifying individual subsets. It allows output



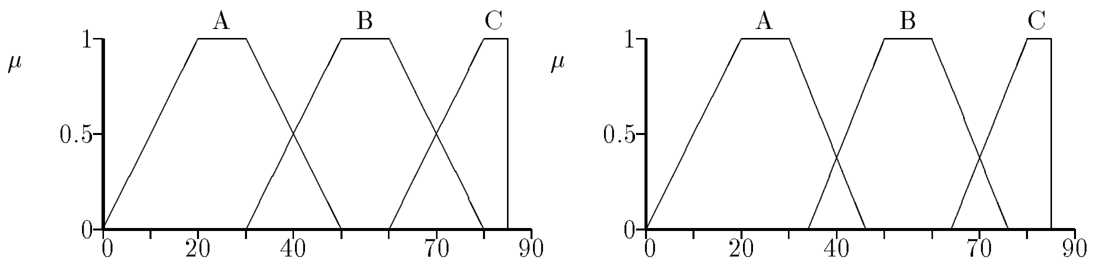
| | |
|---|---|
| .Wedge1 | is $a$ to $a$ to $b$ to $c$ |
| .Trapezium | is $a$ to $b$ to $c$ to $d$ |
| .Triangle | is $a$ to $b$ to $c$ |
| .Rect | is $a$ to $b$ |
| .Line | is $a$ |
| .Wedge2 | is $a$ to $b$ to $b$ |

Figure 5: Trapezoidal Shape Specification

subsets to also have trapezoidal shape, but we recommend the use of weighted *singletons*—shapeless vertical lines. They are simpler, and can often be further simplified by using a default weight of one throughout. The shell implements the correlation-product inference method, which does not use the shapes of output subsets because the use of that extra information complicates the developers understanding of the inference.

A fuzzy subset does not operate in isolation. A variable's subsets are usually intended to provide a complete *cover* over its range. There are instances in the literature of subset covers with irregular distributions and irregular regions of slope that do not overlap (e.g., 8, 22]), but commonly subsets and their overlap are regular (e.g., [2, 7, 15]) and often slope only occurs at overlap (e.g., [6, 13, 15]). A cover of subsets with regular overlap between neighbours is hard to represent and maintain with a collection of individual subset specifications because the simple relationship between the vertices of successive subsets is obscured by their textual independence. We therefore define a multiple subset cover to consist of a series of subsets with slope only at overlap, and our shell has special syntax for specifying them.

We divide covers into the *semi-regular* and *fully regular* types shown in Figure 6 & Figure 7. The subsets of a semi-regular cover are specified explicitly in a *shape list*, but the toe and crest of neighbouring subsets are initially assumed to match, and a single value is given for both. The left-most subset in a shape list is represented by two vertices followed by its name and one or two more vertices. The next subset assumes the last two vertices of the previous subset to be its first two vertices, so its name comes immediately after the last toe vertex of the previous subset, and is followed by one or two more vertices.



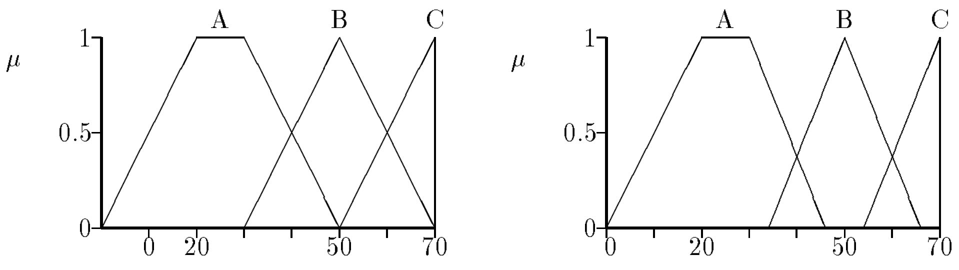
[0 A B C 85 85] 10
[0 A B C 85 85] 40%
a.

[0 A B C 85 85] 10 4
[0 A B C 85 85] 40% 40%
b.

Figure 6: Semi-regular Cover Specification

In Figure 6a for example, *A* is specified with toe 0 and crest 20 before the name, and crest 30 and toe 50 after. *B* uses the 30 and 50 from *A* as its first toe and crest, and is followed by 70 for its final toe. *C* takes the preceding 50 and 70 as its first toe and crest, and ends the definition with a repeated 70 for its final toe, making it a wedge.

An optional toe retraction quantity following the closing bracket reduces the overlap. It can be specified either as a fixed value in the variable's units, or a real number followed by % that specifies the percentage of



[0 20 A 30 50 B 70 C 70]
a.

[0 20 A 30 50 B 70 C 70] 4
[0 20 A 30 50 B 70 C 70] 40%
b.

Figure 7: Fully Regular Cover Specification

each gulf between the crests that does not overlap. In Figure 6b all toes are retracted by 4 units leaving 8 units, or 40%, of the distance between the crests not overlapping.

The subset shapes of a fully regular cover are calculated automatically, and only its bounds and the names are supplied in a *name list*. Each non-wedge subset in the cover is identical and symmetrical, and overlap between neighbours is identical but not necessarily complete. An end wedge subset is a standard subset cut in half, and it is specified by duplicating the first or last bound. A value representing the proportion of each subset width that is at a membership of 1 (the peak width) can follow a name list. It defaults to 0 for triangular subsets. A toe retraction quantity can be appended.

With this special syntax, subset covers can be specified with ease, and with confidence that the subsets are regular and properly matched.

### 4.3 RULES

A rule consists of the key word **`if`**, a condition part, the key word **`then`** and a consequent part. The condition part is a straightforward expression tree of input terms, and the consequent is a list of output terms. Negated output terms are assigned the negation of the rule's value. Sub-expressions present in more than one rule are evaluated only once, making them effectively script author defined higher-level variables. The special subset names *True* and *False* may be omitted in the condition part of a rule, as they are deduced from the context. A term with no subset specified is assumed to refer to a subset *True*, a negated one to *False*. If *True* is assumed, but only *False* defined, then *not False* is assumed. The inverse works for an assumption of *False*.

Variables with arity greater than one are called *vector variables* (VVs). They require careful handling in rules. In the condition part of a rule, a *vector term* is either a VV term, or two vector terms of the same arity joined by a conjunction. Conjunctive operations within a vector term are performed element by element. A *scalar term* is anything else: a non-vector variable term, the conjunction of two scalar terms or the conjunction of two terms of differing arity. A vector term can also be forced scalar by suffixing the conjunction with **`1`**.

A vector term is converted into a scalar term by selecting the element that will give the rule its highest activation. If the rule were duplicated to create one instance per element then the instance with the highest activation would over-ride the rest, and therefore only the element that gives this activation need be considered. Selection of this element depends on the negation status of the term. If the term is not negated (or is negated an even number of times in the expression tree) then the largest element is chosen. If the term is negated an odd number of times then the smallest element is chosen.

If the condition part of a rule is a vector term and a consequent term has the same arity then, again, the operation is performed element by element. The rule becomes effectively a family of rules relating corresponding elements of condition and consequent terms. If the arity of a consequent term differs then the condition is made scalar, and the consequent's elements are treated as distinct terms. Again, the condition part can be forced scalar by suffixing the *then* with **`1`**.

## 5 A SIMULATION APPLICATION

For the first application we use the truck backing-up problem given by Kosko [8]. This allows us to demonstrate rules scripted from a matrix-based controller, and prototyping a controller from scratch using different control variables. The task is to back a truck from any location and orientation in a rectangular yard up to the middle of a dock that forms one of its sides. Here the grounding code does not ground terms to sensors and actions, but is itself a simulation of the problem.

Fuzzy rule matrices are easily translated into text rules for the shell, and Figure 8a shows the results obtained from a shell implementation of Kosko's rule set [8]. It has 35 two-term rules that take orientation and lateral position as inputs, and give steering angle as output. With this rule-set the truck follows the same path irrespective of its perpendicular distance from the dock, and it can therefore have trouble when this distance is small. For example, the lower path in the figure reaches the dock before the truck is aligned.

Figure 8b shows the result of making these rules boolean. Note the over-shooting of the goal, and the knees formed when the control point moves from one rule to another. The continuous control output derived from fuzzy rules does a much better job.

The lower two trials are the result of our own prototyping development. We identified the input variables shown in Figure 1 because they are closer to the driver's view than the global *x* coordinate and orientation used by Kosko. Such values should be easier to obtain from sensors and they can be the source of substantial

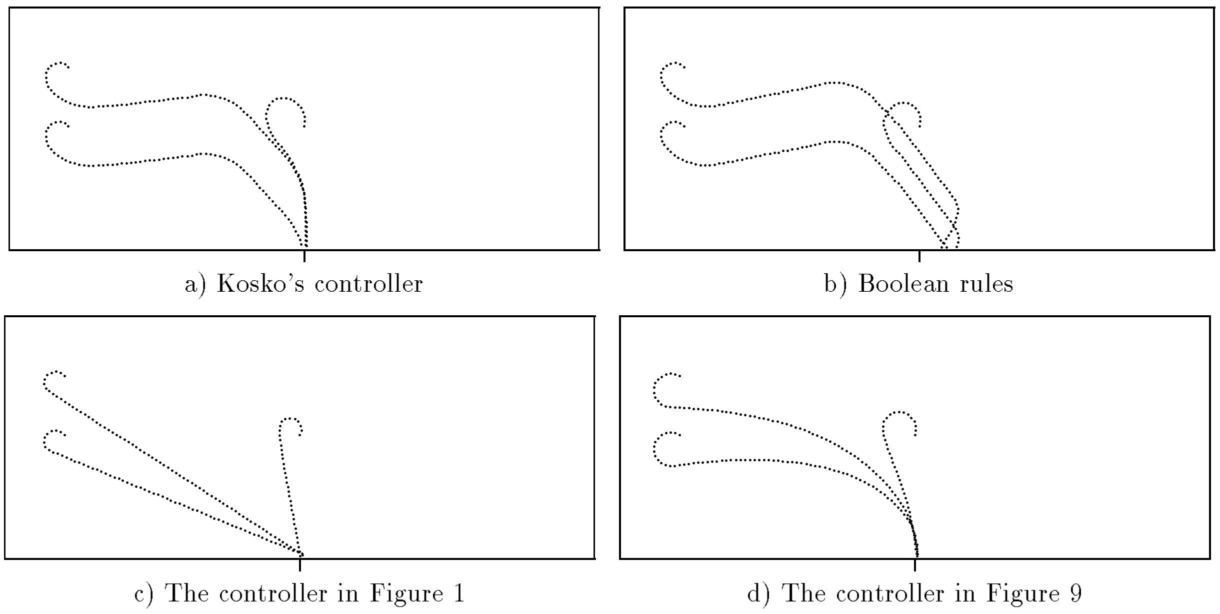


Figure 8: Truck Paths Using Different Rule Sets

economies in problem representation [16]. They improve performance here by loosening the turn for starting points with small perpendicular offset so that the truck can reach the mid-line further from the dock and has space to align itself before reaching it (Figure 8d).

The shell allows us to develop the rule-set incrementally, adding subsets and rules only as required. Figure 8c shows the behaviour obtained from the tiny first-cut rule set of Figure 1. The truck turns and heads for the dock, but does not align with it. It should correct its orientation by first heading for the mid-line back from the dock, aligning and then approaching. To achieve this it must steer *across* the bearing, and this is achieved by the rule set shown in Figure 9. It is significantly simpler than Kosko's matrix of 35 rules.

```
Bearing is input float
[-90 -90 Left -48 -8 CentreLeft -4 4
    CentreRight 8 48 Right 90 90]
Turn is input float
[-180 -180 Left -30 0 Zero 30 Right 180 180]
Steer is output float
.Left  is -20
.CentreLeft  is  -2
.CentreRight is   2
.Right is  20

if Turn is Left and Bearing is not Right
    then Steer is Left
if Turn is Left and Bearing is Right
    then Steer is CentreLeft
if Turn is Zero and Bearing is Left
    then Steer is Left
if Turn is Zero and Bearing is CentreLeft
    then Steer is CentreLeft
if Turn is Zero and Bearing is CentreRight
    then Steer is CentreRight
if Turn is Zero and Bearing is Right
    then Steer is Right
if Turn is Right and Bearing is Left
    then Steer is CentreRight
if Turn is Right and Bearing is not Left
    then Steer is Right
```

Figure 9: The Rule-set for Truck Backing Up

## 6 A MOBILE ROBOT APPLICATION

In the second application, two instances of the shell are used in a control system for wall tracking for a mobile robot with a ring of 16 sonar sensors [18]. The control system is based on a *sector* model. The sensors divide the circumference of the robot into 16 equal sectors of 22.5°, the *sector angle* (Figure 10). The robot

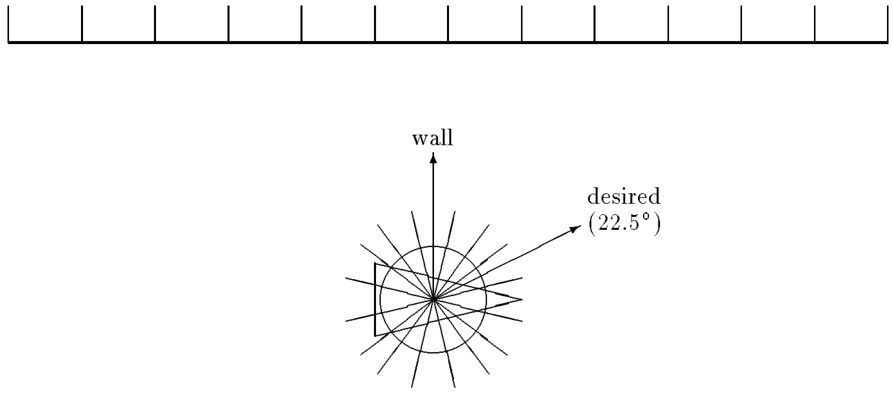


Figure 10: The Wall and Desired Sectors

initially identifies a sector containing an orthogonal wall nearby, the *wall sector*. It then turns a multiple of the sector angle such that the *desired sector* for the current range faces the wall. Then, as it approaches the wall, the robot turns in steps, changing the wall sector at preset range thresholds such that it converges to a parallel track (Figure 11).

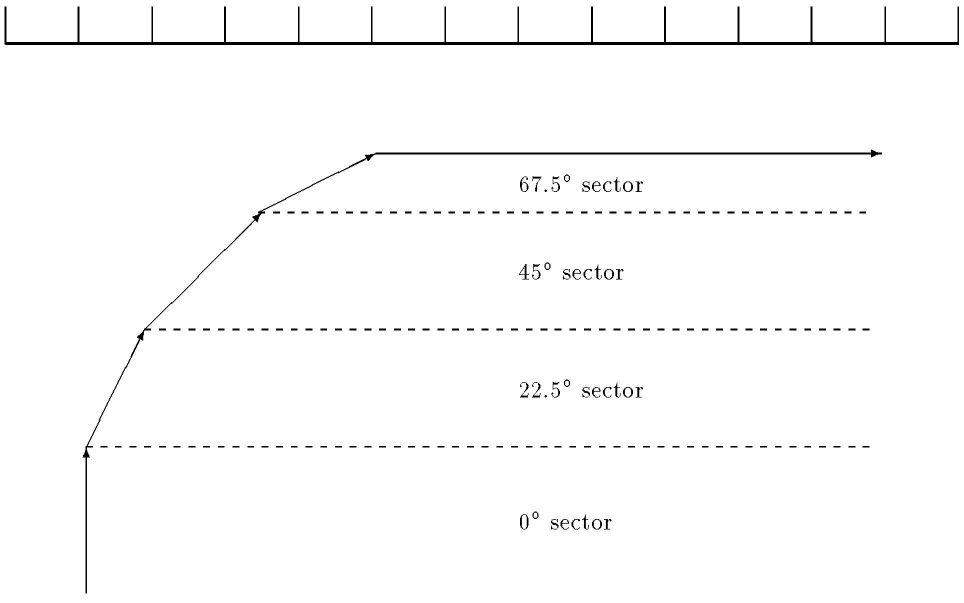


Figure 11: The Path to Approach the Wall

The first instance of the shell was used to develop the initial wall sector identification. The script is shown in Figure 12. We identified nearness and flatness as positive indicators. Nearness is desirable in a sonar return because it indicates a close wall, and also that the return is unlikely to be the result of multiple reflections. To establish flatness, the robot moves in a straight line, performing three complete scans of the sonar ring. *ddRange* is the double differential of the three ranges: ($R_3$ - $R_2$) - ($R_2$ - $R_1$). It is a measure of the co-linearity of the three returns, and hence of the flatness of the surface perceived. The maximum *Sector* value indicates the most promising sector. Initially, we used only the first rule, and used prototyping to find effective subset points. We then introduced the second rule to increase the appeal of strongly co-linear returns.

```
Range is 16 input int
     .Near is 0 0 2000
ddRange is 16 input int
     .Flat is 0 0 200
     .VeryFlat is 0 0 50
Sector is 16 output int
     .Useless is 0
     .Wall is 500
     .StronglyWall is 1000

if Range is Near and ddRange is Flat then Sector is Wall
     and Sector is not Useless
if ddRange is VeryFlat then Sector is StronglyWall
```

Figure 12: The Wall Identification Script

As the robot moves towards the wall, it must keep the sensor orthogonal to the wall to preserve the accuracy of the return [9]. The second instance of the shell was used to develop the controller for this. The script is shown in Figure 13. The neighbouring sensors are far from orthogonal to the wall, so their returns cannot be used to establish its angle. We therefore identified the difference between successive returns from the wall sensor as the controller input. The raw difference changes with the angle of approach to the wall, so the grounding code subtracts the *expected closing distance* for the current wall sector from it before it is used by the script. The output, *Turn* is the rate of turn.

```
Difference is 1 input int
     [-30 -30 Negative Positive 30 30]
Error is 1 input int
     [-80 -80 Negative Positive 80 80]
Turn is 1 output char
     .Out      is   65
     .SmallOut is   15
     .Zero     is    0
     .SmallIn  is  -15
     .In       is  -65

if Difference is Negative then Turn is Out
if Difference is Positive then Turn is In
if Error is Positive then Turn is SmallIn
if Error is Negative then Turn is SmallOut
```

Figure 13: The Orientation Control Script

We first obtained good results through prototyping using only *Difference*, and then we introduced *Error* to make the robot converge on the exact tracking distance once it is parallel to the wall. When the error is outside the range defined for *Error*, it has no effect. The two new rules only fire when the 90×° sector faces the wall and the robot is ready to home-in on the exact tracking distance.

Figure 14 shows an example run. Wall identification occurs during the initial straight section moving away from the wall. The robot then turns, approaches the wall and tracks it. Note that the robot corrects its orientation to the wall slightly after each turn, and moves in to the exact tracking range once it is parallel. The system has been used to handle corners by monitoring the forward-facing sensor and switching control to that sensor when its range return makes it the desired sensor. The robot then performs its stepped turn to align with the new wall.

## 7 CONCLUSION

We have outlined the differences between traditional applications of expert system shells and the controller development task, and described the architecture of the shell we have developed for rapid prototyping development of controllers. The shell uses fuzzy inference to obtain continuous output, a single step of inference at each time step to guarantee real-time response, allows script symbols to be grounded via a base programming language, and implements vector variables to take advantage of structure in the control

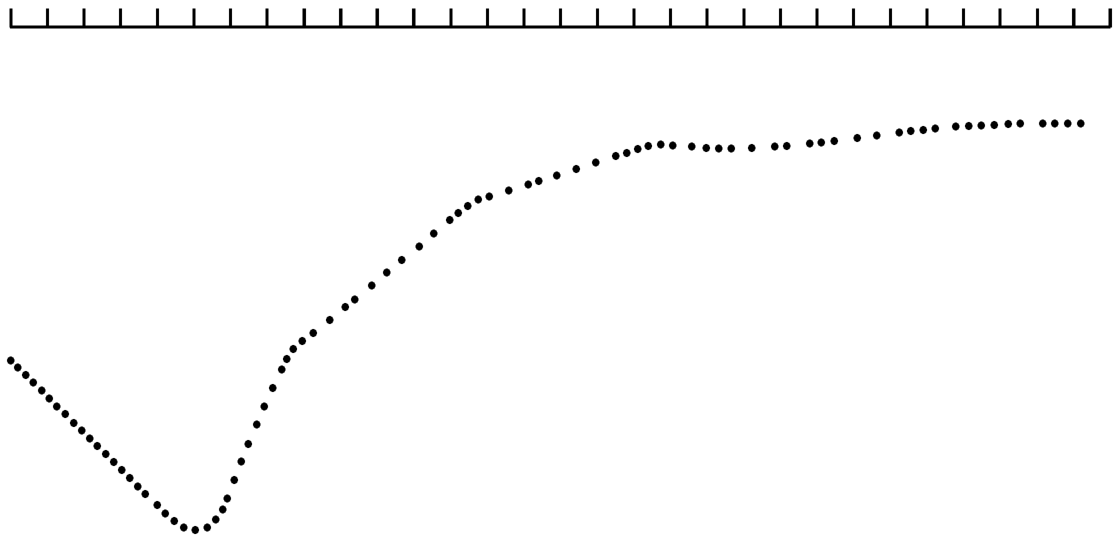


Figure 14: The Robot’s Actual Path

data. We have briefly described two example applications of the shell that demonstrate its effectiveness in differing tasks.

Fuzzy inference allows us to take a rule-based approach to controller development. By prioritizing the identification of control variables and the development of control relations, we have been able to create a rapid prototyping environment for the latter, while allowing the control variables to be implemented in an established programming language as part of a larger control system. The shell is very easy to integrate into a control system, and multiple instances can be used to handle different parts of the control system’s task.

## REFERENCES

[1] Nahrul K. Alang-Rashid and A. Sharif Heger, A general purpose fuzzy logic code, In *Proc. IEEE Int. Conf. on Fuzzy Systems*, pages 733–742, Piscataway, NJ, March 1992.

[2] Jason Birch, Self-learning and adaptive fuzzy controllers, In *Proc. 1993 Dept. Research Conf.*, Department of Computer Science, University of Western Australia, July 1993.

[3] Rodney A. Brooks, Intelligence without reason, In *Proc. 12th Int. Joint Conf. on AI*, pages 569–595, San Mateo, CA, 1991. Morgan Kaufmann.

[4] J. Buckley, W. Siler, and D. Tucker, FLOPS, a fuzzy expert system: Applications & perspectives, In C. V. Negoita and H. Prade, editors, *Fuzzy Logics in Knowledge Engineering*, pages 256–274. Verlag, TUV, Cologne, 1986.

[5] F. F. Ingrand, M. P. Georgeff, and A. S. Rao, Architecture for real-time reasoning and system control, *IEEE Expert*, 7(6):34–44, 1992.

[6] Pratap S. Khedkar and Srinivasan Keshav, Fuzzy prediction of timeseries, In *Proc. IEEE Int. Conf. on Fuzzy Systems*, pages 281–288, Piscataway, NJ, March 1992.

[7] Bart Kosko, Fuzzy systems as universal approximators, In *Proc. IEEE Int. Conf. on Fuzzy Systems*, pages 1153–1162, Piscataway, NJ, March 1992.

[8] Bart Kosko, *Neural Networks and Fuzzy Systems*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[9] John J. Leonard and Hugh F. Durrant-Whyte, *Directed sonar sensing for mobile robot navigation*, Kluwer Academic Publishers, Cambridge, MA, 1991.

[10] Chris Malcolm and Tim Smithers, Symbol grounding via a hybrid architecture in an autonomous assembly system, In Pattie Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 123–144. MIT Press, Cambridge, MA, 1990.

[11] Phillip J. McKerrow, Computer controlled galvanizing, *Computers in Industry*, 4(1):19–30, March 1983.

[12] Marvin Minsky, A framework for representing knowledge, In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*, pages 245–262. Morgan Kaufmann, San Mateo, CA, 1985.

[13] Ashok Nedungadi, A fuzzy robot controller—hardware implementation, In *Proc. IEEE Int. Conf. on Fuzzy Systems*, pages 1325–1331, Piscataway, NJ, March 1992.
[14] Anil Nerode and Wolf Kohn, Multiple agent autonomous control: A hybrid systems architecture, In J. N. Crossley *et al.*, editor, *Logical Methods*, Birkhäuser, 1993.
[15] Peter J. Pacini and Bart Kosko, Adaptive fuzzy systems for target tracking, *Intelligent Systems Engineering*, 1(1):3–21, 1992.
[16] Pushkar Piggott and Abdul Sattar, Reinforcement learning of iterative behaviour with multiple sensors, *Applied Intelligence*, 4(4):351–365, October 1994.
[17] Pushkar K. Piggott and Phillip J. McKerrow, A shell for fuzzy control system prototyping, In *Proc. 17th Annual (Australian) Computer Science Conf. (ACSC-17)*, volume B, pages 489–497, January 1994.
[18] Pushkar K. Piggott and Phillip J. McKerrow, Mobile robot control with a fuzzy control shell, In *Proc. of Nat. Conf. of Australian Robot Assoc.*, pages 422–430, Sydney, July 1995. ARA.
[19] Francois G. Pin and Yutaka Watanabe, Navigation of mobile robots using a fuzzy behaviourist approach and custom-designed fuzzy inference boards, *Robotica*, 12:491–503, 1994.
[20] Fabrizio Russo, A user-friendly research tool for image processing with fuzzy rules, In *Proc. IEEE Int. Conf. on Fuzzy Systems*, pages 561–568, Piscataway, NJ, March 1992.
[21] Kevin L. Self, Designing with fuzzy logic, *IEEE Spectrum*, 27(11):42–105, November 1990.
[22] R. Simutis, I. Havlik, and A. Lübbert, A fuzzy-supported extended Kalman filter: A new approach to state estimation and prediction exemplified by alcohol formation in beer brewing, *J. of Biotechnology*, 24:211–234, 1992.
[23] Michio Sugeno and Takahiro Yasukawa, A fuzzy-logic-based approach to qualitative modeling, *IEEE Trans. on Fuzzy Systems*, 1(1):7–31, February 1993.
[24] M. Tsutomu, K. Hiroshi, and F. Satoru, Fuzzy expert system shell LIFE FEShell—working environment, In *Proc. Int. Symp. on Uncertainty Modeling & Analysis*, pages 153–160, Los Alamos, April 1993. IEEE Comp. Sci. Press.
[25] D. Wood and M. Schneider, Fuzzy expert system control for naval helicopters, In *Proc. IEEE Int. Conf. on Fuzzy Systems*, pages 1051–1056, Piscataway, NJ, January 1994.