A Numberless, Tensed Language for Action Oriented Tasks

David Alan Bourne

CMU-RI-TR-82-12

The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

13 October 1982

This research was sponsored by the Robotics Institute, Carnegie-Mellon University, and, in part, by the Westinghouse Corporation.

Table of Contents

[3]The language description is informal here, however, the formal details are available elsewhere [Bourne 82b].

List of Figures

Abstract

Action oriented languages are number intensive. Graphics languages are centered around *where* to draw something rather than *what* to draw. The "where" involves a tedious numeric description of vertices. Robotics languages are also dominated by a "where" description, but now the "where" specifies a robot motion. The result is an array of numbers that obscures the meaning of the program to its reader.

This paper shows how a number of linguistic devices can be used to eradicate the plethora of numbers from action oriented descriptions. Functions or verbs can be tensed (e.g., past tense) to modify their meaning without duplicating the root function. The result is an English-like description of a control structure. Arguments or nouns can be modified in name, like the use of a **GENSYM** function in Lisp which generates a unique variable name from a character string, and in number (e.g., singular vs. plural). The result is an English-like description of bound and quantified variables. The remaining quantitative description of action tasks can be relegated to a database whose management system is specialized for *number management.*

The resulting language is a formal variant of a natural language with a Lisp-like syntax (i.e., lists with functions in the first position). The programs approach the readability of a natural language without the cost of ambiguity that is inherent in natural descriptions. Finally, the programs can be easily *pretty printed* in English so that they can be read by non-programmers.

1. Introduction

Educated people and computing machinery both communicate with languages and yet there remains a gap between people and machines. The natural languages used by people seem not to be suited for communicating with machines and the existing computer languages seem not to be suited for communicating with people. The verbosity and ambiguity in English sentences can obscure the simple and yet precise ideas that are required for man machine communication. On the other hand, computer languages tend toward obscurity. Each language comes with a set of programming tricks that are foreign to a non-programmer. This project is an attempt to formalize a set of language tricks that are familiar to any English speaker while avoiding the weaknesses inherent to natural language. A byproduct of choosing English tricks is the ability to easily paraphrase programs into English text.

Programming is inherently difficult for many reasons. People are not used to specifying a solution in *every* detail. In personal communication this is usually not necessary since the listener often has the information to fill in the gaps. If he is missing information, it can be systematically obtained from a sequence of questions and answers. Similarly, programming would be greatly simplified if the machine was already an *expert* in the area of discourse. That is, the system would already have all of the details of *how* to execute its basic functions. The remaining task left to the programmer is to describe *what* operations need to be performed to accomplish his goals.

Most present day computer languages are designed for sequential data manipulations and are not amenable to coordinating simultaneous operations. Action tasks can involve manipulating several objects simultaneously to the satisfaction of a programmer's goal; these tasks are characteristic in manufacturing. Raw parts are formed and assembled into final products by machine manipulations that proceed in parallel. For example, one application is the control and coordination of nine machine tools, two of which are industrial robots. Each of these machine tools is operated by its own controller and they are linked together into a star configuration with a supervisor at the center. A program implicitly describes what machine functions can be performed and when they can be applied.

Control functions must be decoupled to such a degree that they can be scheduled for execution parallel to other operations. Unfortunately, the machine tools cannot be relied upon to operate in harmony, so asynchronous activities have to be accepted as the standard mode of operation. The description of these tasks could easily become mired in details, specifying such things as communication line numbers, line speeds, protocol types, hexadecimal addresses and every other imaginable and obscure computerism.

Numbers are abundant in most programs and yet they have little or no meaning when they are taken out of context. In fact, programs would be much easier to understand if there were no explicit numbers present in the text. Unfortunately, they are a necessary evil. Numbers describe exactly where something is or exactly how much something should be done; these details must be present at some level. Therefore, for the sake of the programmer and the completeness of the task description, the numbers remain, but they are condemned to a separate system which knows how to *manage* this database of godless creatures. The numbers are boxed up into relational tables and can be indirectly referenced from within a program.

To satisfy these constraints, the resulting language and operating system isolate the description of the task from the description of the equipment. The task description is *numberless* and uses word constructions which are already familiar to non-programmers. The statements in a program are decoupled into independent rules that can be scheduled for execution over a distributed architecture.

1.1. New Applications Breed New Languages

Each new area of computation brings with it a wave of new programming languages and robotics is no different. AL [Mujtaba 79,82], VAL [Unimate 80], RAPT [Popplestone 78], AML [Grossman 82a,82b] and RAIL [Automatix 81] are but a few. For the most part, each language is a spin-off of another well known computer language: ALGOL, BASIC, APT, PL/1 and PASCAL respectively. The robotics languages are new by virtue of including special task oriented features. These features facilitate solutions to robotic problems and remain couched in a stylistic framework that is already familiar to an experienced programmer. Some features include built-in subroutines (e.g., homogeneous transformations, 'Draw,' and 'Grasp') that are specialized for a particular problem area (see Paul's work in robotics [Paul 77,81] and the proposal for a graphic's standard [SIGGRAPH79]). Other features include new data structures for organizing information like the aggregates (i.e. nested sets of arbitrary types) found in AML. These languages are designed for an already expert programmer to quickly assimilate.

The effect a new language provides is an organizational view that simplifies a class of task descriptions. Of course, this class of descriptions determines how interesting any particular language is to a consumer. Like other products, the language designer often wants to generalize his system of notation until it can be sold to a large market. This amounts to extending the language's applicability to many different kinds of computational problems; thus there are often premature claims to universality. As long as there are new kinds of problems, there will be new ways to express their solutions.

1.2. Language Review

Every good language reflects a familiar structure. *Low-level* languages reflect low-level structure. For example, assembly language is a representation of hardware that performs computational instructions. Similarly, low-level robot programs represent the functions that the machine can perform at its lowest level such as joint movements. The ML program segment in Figure 1-1 is an example of a typical operator, operand program. In this case, an operator is the name for a device function and the operands are used as its input.

```
100 Sensor      7 100 500
110 Sensor      14 -200 300
120 Move        0 2000 0 0 -4000 650
130 Motor       3 4000
140 Dmotor      2 -100
```

Figure 1-1: ML Joint Level Control [Ardayfio 82, page 62]

A programmer who is familiar with a robot and its devices can precisely control them with a language like ML. Another example of a low-level language is APT which is also an operator operand language for controlling machine tools. It has become deeply entrenched in industry partly because it allows for the direct control of the low-level machinery. Apparently, this control is emotionally difficult to relinquish in the face of a computer program. With the advent of new technologies in Robotics, new opportunities are becoming available for younger generations. They are not yet committed to antiquated systems because they have not yet committed their egos to their machines. This is the time to introduce high-level languages into manufacturing.

A *high-level* language directly represents the algorithm at its level. Therefore, programs that manipulate

algebraic expressions have statements which perform algebraic operations. A robot's work is done with its end effector and so sensibly, a high-level language allows a programmer to direct its control. The RAIL program in Figure 1-2 is an example of how a PASCAL-like programming language can describe the cleaning of a welding torch. By embedding the robot primitives in a familiar computer language, programmers will find it comfortable to program these new machines.

Programming robots in a high-level language is essentially programming by side effect. For example, the statement 'Brush = On' in Figure 1-2 is a variable assignment that also turns on a brush as its side effect.

```
Function Clean-Torch
    Begin
;
; Brush out the torch nozzle, then spray it.
;
        Approach 2.0 From Cleaner-Brush
        Brush = On
        Move Cleaner-Brush
        Depart 2.0
        Brush = Off
;
        Move Cleaner-Spray
        Spray = On
        Wait 2 Sec
        Spray = Off
        Depart 2.0
    End
```

Figure 1-2: RAIL program for cleaning torch [Franklin 82, page 404]

This language and others like it (e.g., VAL, AML, AL) are very effective if the people using them are familiar with the language in which they are embedded and want to control the process at this operational level.

AUTOPASS is a *very high-level* language for describing assembly operations [Lieberman 77]. The English-like description in Figure 1-3 is an AUTOPASS program that describes the assembly of a support bracket. From a distance this project looks like it addresses many of the questions involved in this paper. However on closer inspection the AUTOPASS system offers many features not discussed in this paper (e.g., geometrical modeling, grasp calculations and path planning) and *vice versa*. This paper addresses the following issues which are restrictions in AUTOPASS.

1. The English-like sentences in AUTOPASS are made up of a fixed set of verbs and qualifiers (in **bold**) which operate on their subjects (in *italics*). Unfortunately, different applications require different action words to effectively describe a task.

2. The AUTOPASS statements are translated into motion commands one at a time. As the statements are being compiled, they are used to update the state of a geometrical database which illuminates some semantic errors. Unfortunately, variations in the environment are not detected and used to update the state of the database.

3. Parallel computations which are prevalent in action oriented tasks with multiple machines are difficult to describe. This problem is enhanced by the way the statements are compiled one at a time.

4. The level of English-like description is still very low. The descriptions degrade into quantitative measurements and the structure of the statements is limited to declarations.

5. AUTOPASS programs are embedded in a pseudo PL/1 language so that the PL/1 control structures can be fully utilized. The same philosophy of English-like description is not employed for both control and statement definition.

1. **Operate** *nutfeeder* **With** *car-ret-tab-nut* **At** *fixture.nest*
2. **Place** *bracket* **In** *fixture* **Such That** *bracket.bottom*
   Contacts *car-ret-tab-nut.top*
   And *bracket.hole* **Is Aligned With** *fixture.nest*
3. **Place** *interlock* **On** *bracket* **Such That**
   interlock.hole **Is Aligned With** *bracket.hole*
   And *interlock.hole* **Contacts** *bracket.top*
4. **Drive In** *car-ret-intlk-stud* **Into** *car-ret-tab-nut*
   At *interlock.hole*
   Such That Torque Is Eq 1.20 In-Lbs Using *air-driver*
   Attaching *bracket* **And** *interlock*
5. **Name** *bracket interlock car-ret0intlk-stud car-ret-tab-nut*
   Assembly *support-bracket*

Figure 1-3: AUTOPASS program for support bracket assembly [Lieberman 77, page 329]

Despite the many drawbacks of AUTOPASS it is probably still the most impressive system for describing an assembly of parts.

There are several other research systems which use complex models of the system to plan actions. LAMA a system at MIT is LISP based and has many of the features found in AUTOPASS. In particular, manipulator programs are generated automatically from assembly plans. The task description which assembles a piston sub-assembly uses English-like words that are also LISP function names. The initial assembly plan in Figure 1-4 must be translated by hand to a more explicit assembly plan of the same form. A strategy is then chosen to make the final translation to a manipulator program while considering the geometrical constraints of the working world.

The next program segment (Figure 1-5) is an example of production system rules that are written in OPS2 [Forgy 79]. A production system is an interpreter and a set of rules each with left and right hand sides. The left side of every rule is evaluated as TRUE or FALSE and every rule that is satisfied is gathered together into a *conflict set*. One rule is finally chosen for execution by heuristically resolving the conflict.

A few explanations are required before these rules can be understood. The left hand sides are essentially

```
(Insert      Obj1: [Piston-Pin]
             Obj2: [Piston Pin-Hole]
                        Such-That: (Partly (Fits-In Obj1 Obj2))
(Insert      Obj1: [Piston-Pin]
             Obj2: [Rod Small-End-Hole])
(Push-Into   Obj1: [Piston-Pin]
             Obj2: (And      [Piston Pin-Hole]
                             [Piston-Rod Small-End]))
```

Figure 1-4: Initial Assembly Description [Lozano-Perez 79, page 255]

patterns which are being matched to a database. The symbol '=' marks a variable which becomes bound to an object during the matching process. If '=Object' is bound to 'Banana' when the first rule is satisfied then the '=Object' in '(High =Object)' is also bound to 'Banana.' Finally, the '=' that stands alone in the last rule can be bound to anything.

```
((Want (Monkey Holds =Object))          → (Want (Ladder Near =Place)))
(High =Object) (=Object Near =Place)

((Want (Monkey Holds =Object))          → (Want (Monkey On Ladder)))
(High =Object) (=Object Near =Place)
(Ladder Near =Place)

((Want (Monkey Holds =Object))          → (Want (EmptyHanded Monkey)))
(High =Object) (=Object Near =Place)
(Ladder Near =Place) (Monkey On Ladder)

((Want (Monkey Holds =Object))          → (<Write> "The Monkey Grabs The " =Object)
(High =Object) (=Object Near =Place)      (Monkey Holds =Object)
(Ladder Near =Place) (Monkey On Ladder)   (<Delete> (Want (Monkey Holds =Object))))
(Not (Holds Monkey =))
```

Figure 1-5: A partial rule set to reach high objects [Forgy 79, pp. 11-12]

A production system can be used to schedule computations on a star computer network simply by passing along the satisfied rules to the correct processors. Unfortunately, there are a few dangerous pitfalls. For example, if two rules are executed which move two robots then the robots may collide. This problem results from a hidden dependence in the rules which must be either eliminated or one of the rules must be discarded during conflict resolution [Bourne 82].

One of the main reasons for developing a very high-level language is to make the system accessible to those who have never programmed. On appearance alone, both OPS and LAMA would scare off the uninitiated.

The language in this paper is a very high-level language that is specialized for action oriented tasks. These tasks are executed on a star computer network with machine tools (e.g., robot arms, vision systems, machining centers ...) at the points of the star. The people programming are familiar with their equipment but not with any particular programming methodology. Therefore, this language uses many features of English rather than features which are typical to computer programming languages.

2. Syntax: Combining Atomic Terms[1]

The syntax is very similar to LISP and several production systems [Waterman 78]. Complex terms are composed of functions followed by their arguments where each of the arguments in turn can be another complex term.

(Function Arguments) (1)

Rules are constructed from these terms by pairing boolean functions with command functions.

(Boolean Arguments) → (Command Arguments) (2)

The resulting rule's right hand side is executed whenever the left hand side is TRUE. A program is a set of rules which can be executed asynchronously. However, to limit the size of the rule set, the right hand side can also be another set of rules (i.e., predicate - action pairs).

(Boolean Arguments) → {Rule-Set} (3)

Nested rules reduce the amount of computation required to find the set of satisfied ones, since the embedded rules are essentially invisible. Once an outer rule is satisfied, the inner rules become accessible and their left hand sides must then and only then be continually checked. In addition, rule nesting is a programming device which can be used to logically structure the rule set, thus making the program easier to understand.

{Name Rule-Set} (4)

Finally, a program is any named rule set. This resolves many problems in formatting large programs that are deeply nested because any named rule set can be invoked on the right hand side of any rule, thus making the program easier to read.

3. Words: The Atomic Terms

The readability of a program is directly related to the atomic terms or words in a language and the order in which they occur. The more closely aligned these words are with already familiar words the less there is to learn, thus making it easier to assimilate. The more concise the notation the less that has to be read, thus making it easier to absorb in a glance. The fewer ambiguities in expression the less context has to be analyzed, thus making it easier to understand. These are the design goals and the reasons for choosing English words.

3.1. Functions/Verbs

English has a very rich underlying structure. For example, functions are deeply embedded in sentences and usually manifest themselves as either modifiers (e.g., adjectives and adverbs) or verbs. A unary function is hidden in a simple sentence usually in the form of an adjective.

---

[1]The language description is informal here, however, the formal details are available elsewhere [Bourne 82b].

The first robot on the assembly line is broken. (5)

(Broken First-Robot) (6)

Whereas, there are many occurrences of more complicated functions with many arguments.

The red-robot presented to the blue-robot a turbine-blade. (7)

(Presented Red-robot Blue-robot Turbine-blade) (8)

Functional representations of English have been studied extensively by logicians [Quine59] and linguists [Montague74]. However, the structure of English is *not* the point of this paper other than to appreciate what would be commonly familiar to non-programmers. Rather, words and a few linguistic devices are borrowed from English and are used unambiguously to describe the *action oriented tasks.*

Typical tasks have at least three components. For example, suppose you are hungry and undertake the process of satisfying your hunger. You must first of all purchase the ingredients that are needed to prepare the meal and locate yourself in an appropriate place, such as a kitchen. These are at least some of the task's *pre-conditions*, because the conditions must be TRUE before the process can begin. In addition, you must have cooked and eaten the meal in order to have resolved the hunger. The meal having been cooked and eaten are some of the task's *post-conditions*, because those conditions are TRUE after the task is complete. And finally, the whole process should be enjoyable. This is one of the task's *while-conditions*, because you continue to eat only as long as you are enjoying the meal. Restated, there is a test to see whether the meal is possible and if it is possible, the meal is consumed as long as it remains enjoyable. These condition classes are pervasive throughout task oriented computations and therefore need to be represented in a concise and elegant way.

The conditions are paramount to functions and the condition class can be conveniently indicated by special function markers. Again, English-like devices can be easily employed as function markers.

The *past tense* of a verb indicates that some action has already taken place and is used to mark the function as a boolean (i.e., it returns TRUE or FALSE). In other words,a function in the past tense is a natural way to express a pre-condition.

(Grasped Turbine-blade) (9)

The *present tense* of a verb naturally reads as an imperative and is used to command the system to make the verb's past tense TRUE. The result is a convenient way of representing commands which double as post-conditions.

(Grasp Turbine-blade) (10)

The *active tense* of a verb describes an action which is in process and is also used to mark a boolean function. Active tense descriptions accurately describe the while-conditions of a task.

(Grasping Turbine-blade) (11)

The active tense is distinguished from the past tense by the duration of its truth value. Once something has been "grasped" it continues to have been "grasped" within the context that is defined by the nesting of rules. On the other hand, a robot is only "grasping" something during the actual operation. This distinction is valuable for describing a program's control structure and can be used much as the IF and WHILE statements are used in a typical structured programming language.

Regular verbs are decomposed into their appropriate parts, root and ending, by a very simple procedure [Winograd 71] which is augmented with a dictionary to manage the common irregular verbs. In 1971 this was considered an application of Artificial Intelligence because Winograd was developing these routines within the context of natural language understanding. However, here the routines are just used to provide supplemental information to the lexical analysis phase of compiling within the scope of a formal programming language.

3.2. Connectives and Logical Completeness

A programming language should encompass more than simple concatenations of function calls. Boolean connectives (i.e., 'And', 'Or' and 'Not') are essential for representing complex conditions, such as: 'A robot should move to the furnace, only if it is ready *and* there are *not* any obstructions.' Furthermore, notions of variables and quantifiers are necessary to provide the complete mechanism of reference. As an example, there must be a mechanism for referencing the subject in a previous clause. This logical completeness is available in the first order predicate calculus though many lay-people find it overly technocratic. In addition, there is no widely accepted means of representing anything other than declarative sentences in the first order predicate calculus which dismisses imperative and interrogative sentences.

3.3. Arguments/Nouns

A previous section discussed linguistic devices for modifying functions, so that the resulting clauses are easy to read. This section shows how a function's arguments can be modified in number, so that the expressive power of quantification is captured without the loss of readability. The examples illustrate how an English sentence is translated to the predicate calculus and then how that sentence is translated to our new language. The purpose of these translations is to unambiguously relate the meanings of these sentences to an already familiar language.

The first example shows an English sentence (12) with a hidden universal quantifier (13). The intended meaning of this sentence is that '*all* of the billets have been moved to the furnace,' and this interpretation is triggered by the use of the *plural* noun 'billets.' The first order predicate calculus expresses a plural noun somewhat differently. Rather, than modifying the arguments themselves the predicate calculus represents a plural noun (e.g., 'billets' in (12)) as a quantifier, variable and predicate (13). This clarifies many issues including the scope of the quantifier, which in turn simplifies problems concerning reference (e.g., "What object(s) are referred to by the word 'it' in (18)?"). The conditional in (13) is used in place of a conjunction so that the resulting sentence is true even if there were no billets to be moved. The sentence undergoes its final translation to 14 and uses the plural noun form to explicitly signal the quantifier's presence. (14).

The billets have been moved to the furnace. (12)

$\forall x$ (Billet (x) $\rightarrow$ Moved (x,Furnace)) (13)

(Moved Billets Furnace) (14)

The second example shows an English sentence (15) with a hidden existential quantifier (16). The *singular* form of 'billet' is a general term that in this sentence indicates that at least one billet has been moved to the furnace. It doesn't matter which billet has been moved or if many of them have been moved. Again in (17) the quantifier has been redisguised as a singular noun. So far, the notation in (14) and (17) is relatively simple compared to the predicate calculus without any apparent loss of representational power.

A billet has been moved to the furnace. (15)

$\exists x$ (Billet (x) $\wedge$ Moved (x,Furnace)) (16)

(Moved Billet Furnace) (17)

The beauty of the predicate calculus is only apparent in the third example (18) where the reference of a pronoun must be resolved to understand the sentence. This example is easily understood by a person because only the billets are likely to be moved in the context of this sentence. Unfortunately, knowledge of this sort is not always so useful.[2] The predicate calculus cleanly resolves this problem with the quantifier since it binds the variable 'x' and the scope of the quantifier is unambiguously determined by the parentheses (19). That is, there is an 'x' that is referred to by 'it,' and that *same* 'x' is a billet, has been found and has been moved to the furnace. It is tempting at this point to throw up your hands and say that the predicate calculus solves all of the problems and that no improvements can be made. However, the fact remains that sentences in the predicate calculus are difficult for the layman to understand for the very reasons that make it unambiguous: the additional unfamiliar symbols and their structure are confounding to the uninitiated. Again, we can use a familiar linguistic trick and provide names for the subjects. What was a general term 'billet' in (17) now becomes a singular term 'Billet1' (20) which denotes a specific object. 'Billet1' refers to the same billet within the same, or lower levels of parenthetical structure; this is the scope of its binding. Numbers are used as suffixes because they are easy to generate and easy to compare. The hope is that this naming convention doesn't take on a technical appearance subjecting it to the same disapproval encountered by the predicate calculus. However, there are other alternatives. For example, unique descriptions can replace the names (21) which makes the functional analysis more complicated and increases the level of parenthesis. Both of these devices are included for the sake of completeness.

---

[2]The sentence 'The businessman bought a company with his friend because he was rich.' is quite ambiguous. Who was rich?

A billet has been found in the rack and it has been moved to the furnace (18)

$\exists x$ (Billet (x) $\wedge$ Found (x,Rack) $\wedge$ Moved (x,Furnace)) (19)

(And (Found Billet1 Rack) (Moved Billet1 Furnace)) (20)

(And (Found (Closest Billet) Rack) (Moved (Closest Billet) Furnace)) (21)

Plural nouns are filling in for universal quantifiers and their linguistic machinery. And now, numerals have been added to singular terms to mark that variables with the same numeral refer to the same object. Unfortunately, the thought of using these two ideas together is somewhat repugnant. There is nothing natural about saying either 'Billets1' or 'Billet1s.' In fact this leads us to realize that the plurals do not indicate a general notion of universal quantification because there is no notion of a variable. The analysis in (12-14) is still correct but it fails when it is extended to a compound clause, because the variables are not really represented at all. The sentence (22) is not represented equivalently by (23) and (24). Equation (23) correctly asserts that the same billet has been moved to the furnace and has been heated. While equation (24) asserts that all of the billets have been moved and heated without regard to their individual identity. The named nouns operate as an existential quantifier and its variables. Steps must be taken to assure that these mechanisms can be used to fill in for the universal form.

The billets have been moved to the furnace and they were heated. (22)

$\forall x$ (Billets (x) $\rightarrow$ Moved (x,Furnace) $\wedge$ Heated (x)) (23)

(And (Moved Billets Furnace) (Heated Billets)) (24)

3.4. Adjectives and Adverbs

Adjectives and adverbs can also be used as functions, however in practice, they are used sparingly. An adjective takes as its argument a single noun and it returns as a result a single noun. Similarly, an adverb takes as its argument a single verb and it returns a single verb. The effect of executing either an adjective or an adverb is to modify the target function's definition in the database. This function's modification is only active within the scope of the rule which initiated it. Before the action clause (25) can be executed, the adverb 'Quickly' and the adjective 'Hot' must be evaluated and the updated function names returned (26). The details of what happens in the database are reserved for the next section.

((Quickly Move) (Hot Billet) Swage) (25)

(Move-quickly Billet-hot Swage) (26)

3.5. Rounding Out The Language with An Adverb

One last problem remains: completing the power of the language with respect to the first order predicate calculus. It is well known that universal and existential quantifiers can be freely inter-translated. For example, (27) and (28) are equivalent sentences. It has already been determined that the universal quantifier is only partially represented and so it becomes necessary to fully utilize the power of the existential mechanisms. The 'Not,' introduced earlier, only operates outside of the quantifier's scope. Therefore,

another form of 'Not' must be introduced to modify a function's meaning within the quantifier's scope. A 'Not' used as an adverb fills this obligation and completes the translation between (29) and (30), and completes the language with respect to the first order predicate calculus.

$$\forall x\ (\text{Billet}(x) \rightarrow \text{Moved}(x,\text{Furnace})) \tag{27}$$

$$\neg\exists x\ (\text{Billet}(x) \wedge \neg\text{Moved}(x,\text{Furnace})) \tag{28}$$

(Moved Billets Furnace) (29)

(Not ((Not Moved) Billet1 Furnace)) (30)

3.6. The Language Tightrope

The appeal for using linguistic tricks in a formal language is very seductive and even begins to take on airs of being trivial and obvious. It is neither. The problem of completing the language illustrates how the objective is a tightrope of peril. One slip to the left and the language slips into mountains of ambiguity that is inherent to natural language, and one slip to the right and the language loses its expressive power. However, the advantages of crossing the tightrope seem to outweigh the perils.

4. Where the Numbers Belong

Numbers have names just like people have names. These names are called numerals when they look like '3.' But there is nothing special about these particular names other than their conventional use and their one-to-one correspondence with their distant cousins, the numbers. Other names for the numbers might be Furnace-temperature, Age and Four-bytes. These names don't have to be used in context, but it would be confusing if Four-bytes referred to the number '3.' In addition to referring to numbers, these names can refer to arbitrary sets of values, numeric or otherwise. For example, a set of values to represent the furnace temperature is shown in Figure 4-1. Fortunately, we can talk about 'Furnace Temperature' without speaking directly of '(2200,2300,2258,3,177506).'

Furnace	Min	Max	Current	Line #	Address
Temperature	2200	2300	2258	3	177506

Temperature	Min	Max
Steel	2200	2300
Titanium	2350	2400
Idle	1000	1100

Figure 4-1: Two relational tables that describe details of the furnace.

The values that are needed to describe the low level details of action oriented tasks are stored in relational tables. These tables are accessed and manipulated with a relational algebra and the result of these manipulations is always another table. The rows and columns of the tables all have symbolic names that correspond with the words in a rule set. When a rule is executed, it triggers a set of relational operations that make the appropriate changes in the database. The relational operators make up a majority of the database management system. Suppose the following clause were executed.

(Adjust Furnace Temperature Idle)

This clause would update the minimum and maximum furnace temperature by selecting the 'Idle' row in the table 'Temperature' and overwriting the appropriate slots in the table 'Furnace'; they are determined by the row ('Temperature') and column ('Min' and 'Max') names. The result of these operations is shown in Figure 4-2.

Furnace	Min	Max	Current	Line #	Address
Temperature	1000	1100	2258	3	177506

Figure 4-2: The resulting table after database operations.

Database updates trigger consistency checks that verify the correctness of related information. Simply, the minimum and maximum furnace temperatures are directly related with the current temperature by a procedure's definition. If the current temperature remains within the bounds, then nothing happens; but if it lies outside of the bounds, a message is constructed which is sent to the furnace driver. The furnace driver in turn packages the message in the appropriate protocol and sends it off to the furnace controller. The furnace controller receives the request and adjusts the level of electiric current to the heating elements which directly changes the furnace temperature. Currently, the data relations are built into the system but research is actively underway to generate them automatically [Bourne 80].

Names appear in the text of a program; they talk about numbers and other objects held in a database. This separation of descriptive machinery is a powerful linguistic device which is used both by people in natural contexts (i.e., 'Do what I mean and not what I say.') and logicians in formal contexts (i.e., logic vs. model theory). Traditionally, computer languages mix syntactic and semantic mechanisms and this lack of separation fosters confusion (e.g., error detection in compiling theory).

5. The Runtime System and The Separation of Power

The runtime system is stratified into three layers and is shown in Figure 5-1. The top layer interprets the rules and is responsible for *planning* what actions should be undertaken to accomplish the system's goals. The core of the system is a dynamic database that reflects the state of the task and its constituent machinery. The integrity of the database is maintained by its management system. In effect, the database management system is directly responsible for *maintaining* a consistent and up to date model of the task. Often, the state of the

task degrades independently from any actions within the scope of control. For example, consider the task of taking a shower and maintaining the temperature of the water. Without touching the hot and cold water knobs, the temperature can change drastically due to the thoughtless behavior of an occupant in the adjacent bathroom. The maintenance of the water temperature is the direct responsibility of the database management system which prepares a request to turn down the hot water. Finally, the bottom layer is responsible for *communicating* with the external task functions. That is, it sends the commands to the water valve controller in the appropriate format.

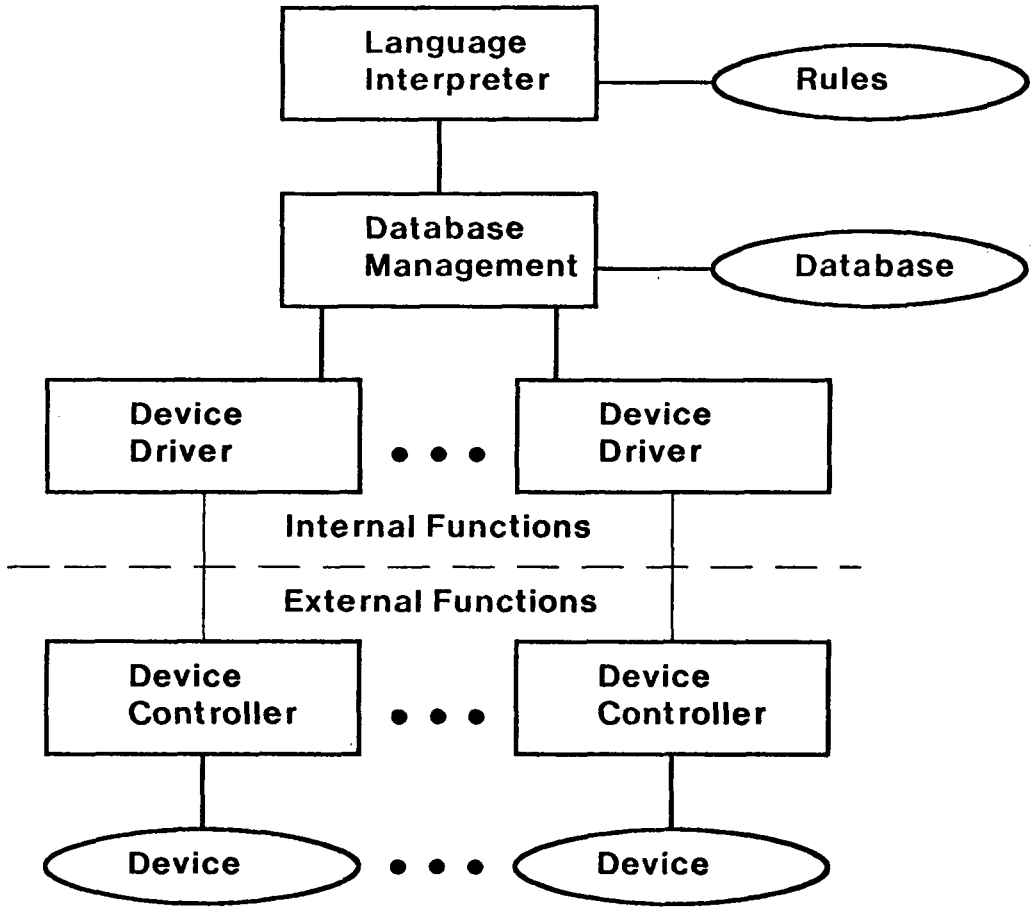

Figure 5-1: Three layers of control internal to the supervisor: top to bottom

The following two rules show how this could be accomplished. Rule-1 can be executed repeatedly after the shower paraphernalia is in place. If the water is not the right temperature then the database management system builds a request that is passed along to the hot water valve driver. The driver packages the message into the appropriate protocol and sends it out to the controller with direct access to the valve actuator. Once the water is warm, Rule-2 can be executed. Again, the advantage of a non-procedural language is illustrated by the fact that Rule-1 may be executed whenever the temperature of the water is unsatisfactory.

(And (Moved Soap ToShower) (Moved Shampoo ToShower))	→	**{(Adjust Water Warm)}**	Rule-1
(Adjusted Water Warm)	→	**{(Get InShower)}**	Rule-2

A simple graphics example shows how a model of an airplane can be animated under the control of a joystick. Of course, the datapoints that define the airplane are kept in the database. This example also shows how an adverb can be used to define the new plane position 'Relatively' to its current position.

(Moved Joystick)	→	**{(Erase Plane) ((Relatively Move) Plane Joystick) (Draw Plane)}**	Rule-3

Rule-4 and Rule-5 demonstrates how a clamp can be loaded when its precise position is not known in advance. The move to 'ToClamp' is initiated in Rule-4. Now the active tense of 'Move' is true in Rule-5 and so the rules internal to it are accessible. When a strain gauge mounted on the robot wrist has encountered a significant load, the internal rule becomes true and its consequent stops the robot from moving further. At this point the entire context of Rule-5 is left and Rule-6 is ripe for execution.

(Gripped Billet1)	→	**(Move Robot ToClamp)**	Rule-4
(Moving Robot ToClamp)	→	**{(Strained Gauge) → (Stop Robot)}**	Rule-5
(Moved Robot ToClamp)	→	**(Release Billet)**	Rule-6

Rule sets can be invoked by using one of the few built-in keywords. 'Perform' is a function which activates a rule set such as 'Preventive Maintenance' in Rule-7. Other forms of the verb are also legal. 'Performed' and 'Performing' and are useful for controlling recursive rule sets and for testing whether a rule set is active.

(Not (Performing Manufacturing)) → (Perform Preventive Maintenance) Rule-7

{Preventive Maintenance

(Fouled Robot Filter)	→	**(Schedule Maintenance Robot Filter)**	Rule-8
(Drifted Robot Positions)	→	**(Calibrate Robot Servos)}**	Rule-9

6. Pretty Printing

After a set of rules has been written, it is straightforward to *pretty print* the programs as English text. By removing the parentheses and adding the appropriate syntactic sugar to the clauses, very readable text can be generated. It can then be further improved by applying a few simple syntactic transformations which compress redundant text into single compound clauses. For example, the two shower rules (Rule-1 and Rule-2) can be pretty printed as the following pair of sentences. 'When' is used to flag the consequent part of the rules. Helper verbs have been added to the verbs and prepositions and articles have been added to the nouns. Programs are *not* written in this form because it would illusively appear as if any English sentence could be interpreted correctly; they are *not* natural sentences but rather sentences in a very simple formal grammar.

When the soap and the shampoo have been moved to the shower

adjust the water to be warm.

When the water has been adjusted to be warm

get in the shower.

Pretty printing simple rules is a matter of printing isolated sentences. This becomes much more difficult when there are active terms, nested rules and named objects. All of these constructions require intersentential relationships. For example, a nested rule like Rule-5 could easily become prohibitively complex. Therefore, the phrase 'consider the following case(s)' stands for the right hand side of the rule and the nested rules can be translated in the standard way.

While the robot is moving to the clamp

consider the following case.

When the gauge has been strained

stop the robot.

Named rule sets are convenient logical segments that can be used to break up text into sections. For example, the rules Rule-8 and Rule-9 make up a program that can be paraphrased as a text segment with its own title.

When the cell is not performing manufacturing tasks

it is time to perform preventive maintenance checks.

PREVENTIVE MAINTENANCE -

Preventive maintenance checks are defined as the following conditional operations.

When the robot's filter has fouled

schedule maintenance for changing it.

When the robot positions have drifted

calibrate the robot servos.

Natural language understanding (e.g., English understanding) may not progress to the point where it is practical to communicate with machines for many years. However, there is no reason why our programs can't be *read* in English today. The whole programming industry has developed into a *write only* society. When was the last time you took home a program just to read? It may turn out that a new generation of programmers that read may be more thoughtful about what programs they write.

7. Some Notes On Implementation

There are two pressing goals in this implementation: readability and execution speed. Task oriented descriptions are specialized for human consumption and are translated into a form that is appropriate for fast execution speeds.

Task descriptions are developed on a VAX 11/780 and compiled into machine readable symbolic expressions which are then downloaded to a DEC PDP 11/23. The compiler and runtime system are written in OMSI PASCAL and many low level Lisp-like primitives make up their basic programming tools. The result are PASCAL programs that read more like LISP than PASCAL.

The Database is also developed on a VAX, and it is also compiled into a machine readable memory structure. The English words which are defined by the database are input to the rule's compiler so that they can be replaced by machine pointers before they are passed along to the 11/23.

Finally, the task descriptions are being used to control a complex manufacturing cell in a Westinghouse factory [Wright 82]. The cell manufactures turbine blade pre-forms from cylindrical bar stock using nine machine tools, a supervisory computer and ten machine controllers.

8. Summary

An application program has two basic parts. A set of relational tables that describe the physical system and a set of rules that update its state. This separation of power simplifies both the management of information that is needed to model a physical system and the description of its task. The numbers and other details that can make a program so difficult to read are not present in the final task description. This simplification, together with a language that utilizes familiar linguistic devices results in a program which is readable to the uninitiated.

It is usually time consuming to gather together this database of facts but its structure is very simple and automated tools have been built to further aid in the database's construction. Once the database has been built, writing the necessary rules is fairly easy. Many of the details can be left out of the description and the description that is necessary parallels the information that would have to be given to an human apprentice. 'Under these conditions, perform those actions.' Finally, a novice programmer can look at an existing set of rules and understand the primitive words since they are based on English. He can then mimic the syntactic forms and write new rules to extend the functionality of his application program. Not only can a novice programmer update the rule set, but now his boss can read his pretty printed work without having to learn anything about programming. This unlocks the door to the intelligence of a whole group of bright people who have never been trained as programmers and yet can make valuable contributions to the logic of programs. For the first time programs are *readable.*

9. Future Interests

This project has a wealth of future paths which are being actively pursued. The programs in this language are extremely easy to construct because of its simple syntax. However, a valid criticism is that the available functions and arguments must be known to the programmer at the time of writing. Therefore, this burden should be removed from the programmer by giving him access to the database while his program is being written. For example, the programmer should be able to make the following request during a session with the editor.

Show me all of the functions and arguments related to robots.

This request would result in a list of robot functions (e.g., Move, Grip and Emergency Stop) and their parameters.

The day to day operation of a manufacturing cell is a problem not usually considered as part of CAD/CAM. However, many of the techniques employed in production should be found useful in product development (CAD) and process development (CAM). An expert system should be able to generate a family of designs that satisfy a set of user design constraints. The resulting shapes and knowledge of machining technologies should produce a series of part programs capable of producing the final product. This process can be viewed as a series of language translations: product constraints to part geometries to machine tool operations to the final production of parts. Concise languages that help describe each phase of development will make the final translation from design constraints to production a tractable problem. This research and others like it are just the beginning.

10. Acknowledgment

I would like to thank my student Paul Fussell for helping me sound out many of these ideas and Peter Angeline for programming support. In addition, I would like to thank Paul Wright and the members of Westinghouse Turbine Components Plant for providing the moral and monetary support needed to complete a project of this magnitude.

References

[Ardayfio 82] Ardayfio, D. D. and Pottinger, H. J.
On The Computer Control of Robotic Manipulators.
In G. D. Gupta (editor), *Computer In Engineering 1982*, pages 59-64. ASME, August, 1982.

[Automatix 81] Automatix.
RAIL Reference Manual.
Automatix Inc., Burlington MA 01803, (617)-273-4340, 1981.

[Bourne 80] Bourne, D.A.
On Automatically Generating Programs for Real Time Computer Vision.
Proceedings of the 5th International Conference onPattern Recognition 1:759-764, December, 1980.

[Bourne 82a] Bourne, D.A. and Fussell, P.S.
Designing Languages for Programming Manufacturing Cells.
In *Proceedings of Electro/82.* IEEE, Boston, MA, May, 1982.

[Bourne 82b] Bourne, D. A. and Mashburn, H.
Cell Programming: A User's Guide.
Technical Report, Robotic's Institute, Carnegie Mellon University, 1982.

[Forgy 79] Forgy, C. L.
On The Efficient Implementation of Production Systems.
PhD thesis, Carnegie-Mellon University, February, 1979.

[Franklin 82] Franklin, J. W. and Vanderbrug, G. J.
Programming Vision and Robotics Systems with RAIL.
In *Robots VI*, pages 392-406. Robotics International of SME, March, 1982.

[Grossman 82a] Grossman, D. D.
Robotics Software At IBM.
In G. D. Gupta (editor), *Computer In Engineering 1982*, pages 73-75. ASME, August, 1982.

[Grossman 82b] Grossman, D. D.
Decade of Automation Research at IBM.
In *Robots VI*, pages 535-543. Robotics International of SME, March, 1982.

[Lieberman 77] Lieberman, L. I. and Wesley, M. A.
AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly.
IBM Journal of Research and Development 21(4):321-333, July, 1977.

[Lozano-Perez 79]
Lozano-Perez, T.
A Language for Automatic Mechanical Assembly.
In Patrick H. Winston, Richard, H. Brown (editor), *Artificial Intelligence: Artificial Intelligence An MIT Perspective*, , pages 245-271. The MIT Press, Cambridge, MA, 1979.

[Montague 74] Montague, R.
Formal Philosophy: The Selected Papers of Richard Montague.
Yale University Press, New Haven, 1974.

[Mujtaba 79] Mujtaba, S. and Goldman, R.
AL User's Manual.
Technical Report Memo AIM-323, Stanford University, January, 1979.

[Mujtaba 82] Mujtaba, M. S., Goldman, R. and Binford, T.
The AL Robot Programming Language.
In G. D. Gupta (editor), *Computer In Engineering 1982*, pages 77-86. ASME, August, 1982.

[Paul 77] Paul, R. P.
"WAVE: A Model-Based Language for Manipulator Control,".
The Industrial Robot 4(1):10-17, March, 1977.

[Paul 81] Paul, R.P.
Robot Manipulators: Mathematics, Programming and Control.
The MIT Press, Cambridge MA, 1981.

[Popplestone 78] Popplestone, R.J., Ambler, A.P. and Bellos, I.
RAPT: A Language for Describing Assemblies.
The Industrial Robot 13:131-137, September, 1978.

[Quine 59] Quine, W. V. O.
Methods of Logic.
Holt, Rinehart and Winston, New York, 1959.

[SIGGRAPH 79] SIGGRAPH Standard's Committee.
A Quarterly Report of SIGGRAPH-ACM.
SIGGRAPH 13(3):759-764, August, 1979.

[Unimate 80] Unimate.
User's Guide to VAL: A Robot Programming and Control System.
Unimation Robotics, Danbury, CT 06810 (203)-744-1800, 1980.

[Waterman 78] Waterman, D.A. and Hayes-Roth, F.
An Overview of Pattern-Directed Inference Systems.
In Waterman, D.A. and Hayes-Roth, F. (editor), *Pattern-Directed Inference Systems*, , pages 3-22. Academic Press, New York, 1978.

[Winograd 71] Winograd, T.
An A.I. Approach to English Morphemic Analysis.
Memo 241, Artificial Intelligence Laboratory, M.I.T., February, 1971.

[Wright 82] Wright, P.K., Bourne, D.A., Colyer, J.P., Schatz, G.C. and Isasi, J.A.E.
A Flexible Manufacturing Cell for Swaging.
In *Manufacturing Cells and Their Subsystems.* 14th CIRP International Seminar on Manufacturing Systems, Trondheim, Norway, June, 1982.