NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A Numberless, Tensed Language for Action Oriented Tasks

David Alan Bourne
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

13 October 1982

Abstract

Action oriented languages are number intensive. Graphic's languages are centered around *where* to draw something rather than *what* to draw. The "where" involves a tedious numeric description of vertices. Robotic's 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.

Tills paper shows how a number of linguistic devices can be used to eradicate the plethora of numbers from action orienccd descriptions. Functions or verbs can be tensed (e.g., past tense) to modify their meaning without duplicating the root ftinction. 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 ftinction in Lisp which generates a unique variable name from a character string, and in number (e.g, singular vs. plural). The result is an Fnglish-likc description of bound and quantified variables. The remaining quantitative description of action asks 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.

Copyright @ 1982 David Alan Boeroc

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

629. 2?2
C28r
8?- ' I o?
c. 2

Table of Contents

1. Introduction 1
    1.1. New Applications Breed New Languages 2
    1.2. Language Review 2
2. Syntax: Combining Atomic Terms[3]
3. Words: The Atomic Terms 6
    3.1. Functions/Verbs 7
    3.2. Connectives and Logical Completeness 8
    3.3. Arguments/Nouns 8
    3.4. Adjectives and Adverbs 10
    3.5. Rounding Out The Language with An Adverb 11
    3.6. The Language Tightrope 11
4. Where the Numbers Belong 11
5. The Runtime System and The Separation of Power 13
6. Pretty Printing 15
7. Some Notes On Implementation 16
8. Summary 16
9. Future Interests 17
10. Acknowledgment 17

[3]The language description is informal here, however, the formal details are available elsewhere [Bourne 82b]. 6

List of Figures

Figure 1-1: ML Joint Level Control [Ardayfio 82, page 62] 2
Figure 1-2: RAIL program for cleaning torch [Franklin 82, page 404] 3
Figure 1-3: AUTOPASS program for support bracket assembly [Lieberman 77, page 329] 4
Figure 1-4: Initial Assembly Description [Lozano-Perez 79, page 255] 5
Figure 1-5: A partial rule set to reach high objects [Forgy 79, pp. 11-12] 5
Figure 4-1: Two relational tables that describe details of the furnace. 12
Figure 4-2: The resulting table after database operations. 12
Figure 5-1: Three layers of control internal to the supervisor: top to bottom 13

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 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 conrrol 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 in parallel with 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 [Unimatc 80], RAFF [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 robotic's 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 die 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        7100 500
110 Sensor         14-200300
120Mo?e            0200000-4000650
130 Motor          34000
140Dmotor          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 t language like ML Another example of a low-level language is APT which is also an operator operand language for controlliig machine tools, It has become deeply entrenched in industry partly became it allows for the direct control of Hie tow-level machinery. Apparently, this control is emotionally difficult to, relinquish in the fee 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

```
(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]

[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 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) (<Delete>
(Ladder Near =Place) (Monkey On Ladder)   (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 network simply by passing along the satisfied rules to the correct processors. Unfortunately, there are 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 rales which can be executed asynchronous^. However, to limit the size of the rule set, the right hand side can also be another set of rules (Le», 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 rale set This resolves many problems in formatting large programs that are deeply nested because any Earned 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 die order in which they occur. The more closely aliped 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

[1]Hie fangmgc dc$cnption a mfenaai *hem,* k m e w , Hie formal details are available elsewhere [Bourne 82b}.

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.

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-mles. On the other hand, a robot is only "grasping^1* 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 ui>ed 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 calk. Boolean connectives (Le., 'And*, *'Of* 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 *no!* 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 die first order predicate calculus which dismisses imperative and interrogative sentences.

3*3. Arguments/Nouns

A previous section discussed linguistic devices for modifying ftinctions, so that the resulting clauses are easy to read- This section shows how a ftinction*s arguments can be modified in number, so chat the expvessbe power of quantification is captured without the loss of readability. The examples illustrate how an Englii 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 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

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

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.

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 cxample-(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 die translation between (29) and (30), and completes the language with respect to the first order predicate calculus.

Vx (Billet(x) -* Moved(x,Furnace)) (27)

-i 3x (Billct(x) A ^Moved(x,Furnace)) (28)

(Moved Billets Furnace) (29)

(Not ((Not Moved) Billetl 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 ofX2200a300,2258,3J77506V

The values that are needed to describe the low level details of action oriented tasks are stored in relational tables. These tables arc accessed and manipulated with a relational algebra and the result of these manipulations is always another cable. 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 die database. The relational operators make up a majority of the database management system. Suppose the following clause was executed.

(Adjust Furnace Temperature Idle)

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.

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.

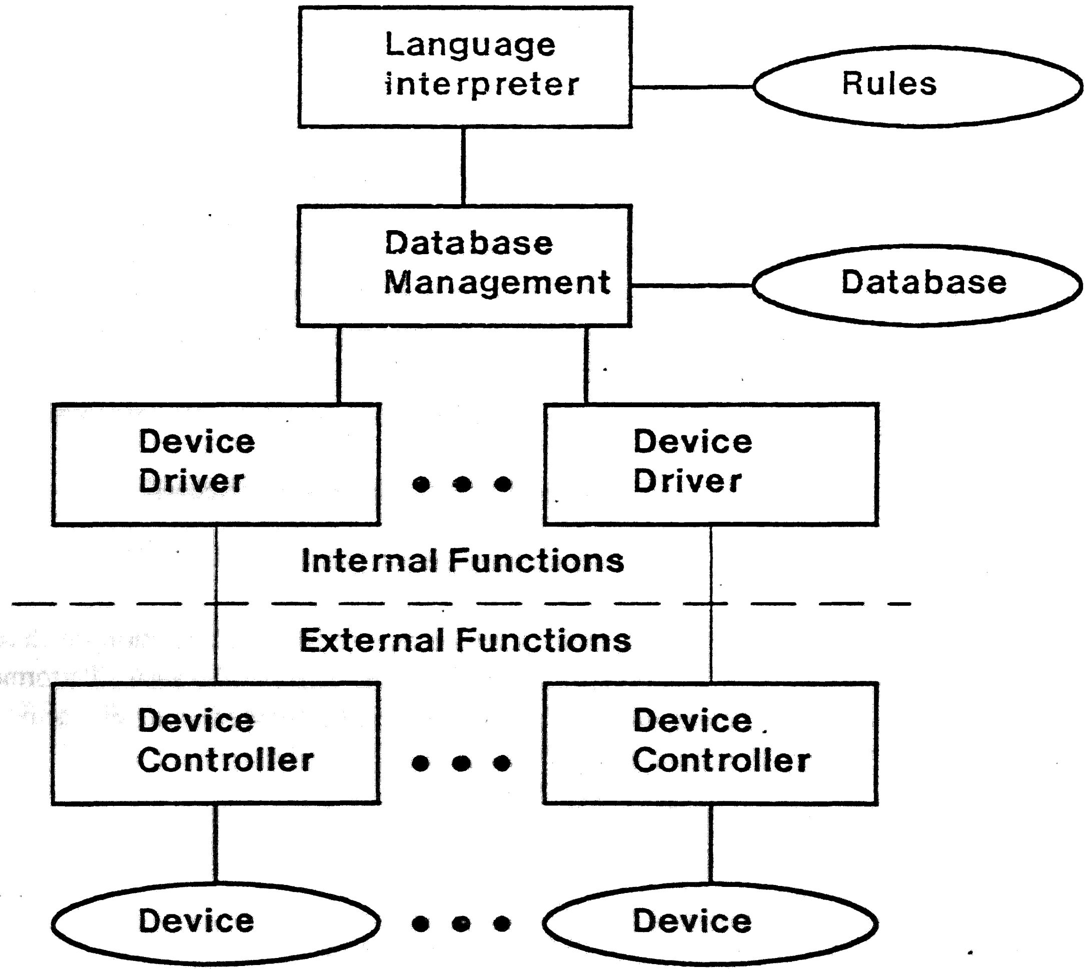

Figure 5-1: Three layers of control internal to the supervisor: top to bottom

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 die 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)          -»   {(Adjust Water Warm)}                          Rule-1
      (Moved Shampod ToShower))

      (Adjusted Water Warm)          -»   {(Get InShowcr)}                               Rule-2
```

A simple graphic's 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)                                 Rule-3
                                           ((Relatively Movc)-Plane Joystick)
                                           (Draw Plane)}
```

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 farther. At this point the entire context of Rule-5 is left and Rule-6 is ripe for execution.

```
      (Gripped Billetl)              -»   (Move Robot Todamp)                            Rule-4

      (Moving Robot Todamp)          -*   {(Strained Gauge)  -*  (Stop Robot)}           Rule-5

      (Moved Robot ToQamp)     ''    -•   (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 Senos)}                       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 most be known to the programmer at the time of writing. Therefore, this burden should be removed from die 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.

Shcm me all of the functions and arguments related to robots*

This request would result in a 1st of robot functions (eg., Move, Grip and Emergency Stop) and their parameters.

The day to day operation of a manu&ctaring 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 'iewtopncntlCADI and process development (CAM)* An expert system should be able to generate a family of designs dut saisftr a set of user design constraints. The resulting shapes and knowledge of machining ittbaclogiis gboukt produce a -series of part programs capable of producing the final product This process caa be viewed *M* a serin of language translations: product constraints to part geometries to machine tool operations to' Ac leal production of pans. Concise languages that help describe each phase of development will make the final trustation from design constraints to production a tractable problem. This research and others ike It are just the beginning.

10. Acknowledgment

I would like to ihaek my student Paul Fusscll for helping me sound out many of these ideas and Peter Angcline for programming support. In addition, I would like to thank Paul Wright and the members of Wcstinghousc 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 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.

Table of Contents

1. DP run-time environment **2**

1.1 Operating system 2
1.2 Installing the system on a Perq 2
1.2.1 Ethernet 2
1.2.2 Floppy Disk 3
1.3 Starting the program 3

2. DP: basic concepts **4**

2.1 The basic elements of a drawing 4
2.2 Status line and Prompt Area 5
2.3 Mouse buttons 5
2.4 Selecting and deleting 7
2.5 Editing strings 7
2.6 Windows 7
2.6.1 Windows and file-names 9
2.7 Layers 9
2.8 Check-pointing 10
2.9 Fonts 11

3. DP command set **12**

3.1 Basic items 12
3.2 Parameters and Fonts 13
3.3 Select and Delete 13
3.4 Copying and moving 14
3.5 Symbol-related commands 14
3.6 File I/O 14
3.7 Display commands 15
3.8 Mouse commands 16
3.9 Unusual Commands 16
3.10 Miscellaneous commands 17

4. Advanced topics **18**

4.1 The coordinate system 18
4.2 The size of a drawing 18
4.3 Editing items 18
4.4 Symbols and Instances 19
4.4.1 Creating a symbol 20
4.4.2 Symbol names 20
4.5 Memory allocation 21
4.6 Alternate input for commands 21

I. Command set table **22**

List of Figures

Figure 2-1: DP: primitive graphic elements 4
Figure 2-2: Moving one item at a time: **Yellow** button 5
Figure 2-3: Moving the items within a rectangle: **White** button 6
Figure 2-4: Moving the selected items: **Green** button 6
Figure 2-5: Changing the shape of a window 8
Figure 4-1: Editing a spline by moving a Control Point 19

Introduction

DP is a highly interactive circuit drawing program that runs on a PERQ computer. It allows a designer to create the description of an electronic circuit in a graphical form that corresponds to the way circuits are normally represented in logic diagrams.

DP is a purely graphic editor: it does not try to "understand^{11} what the user is drawing. All the semantic interpretation is performed by post-processors that are able to extract electrical information out of the drawings. This makes the program itself reasonably simple, giving the designer greater freedom. It also introduces a clear separation between the tool used to create a drawing and the "meaning" of the drawing itself, with the result of a much wider range of possible applications.
Although primarily intended as a tool for drawing electronic circuits, DP may be used as a general illustrating program, able to draw arbitrarily complex pictures.

Chapter 1 explains how to build the program on a new machine and how to start it

Chapters 2 and 3 contain the essential information; after reading those chapters one should be able to use DP for simple tasks.

Chapter 4 contains details that may be skipped at a first reading.

The Appendix contains a brief summary of the command set

Version 5.6 is documented in the present manual.

1. DP run-time environment

The following is a description of things one should know before trying to use the program; a short explanation about how to build the program on a Perq and how to start it is also given. A general understanding of the philosophy and operation of the Perq computer is assumed throughout this chapter. The reader should be able to perform the basic operations required to turn a Perq on and to reach the right directory.

Since the interface with the Perq operating system is still rapidly evolving, some of the information in this chapter is likely to become obsolete. Please consult the author for more detailed information.

1.1 Operating system

DP version 5.6 runs under POS, the Three Rivers Computer operating system for the Perq, version D.65. Previous versions of the operating system are no longer supported.

The program may run with both 256 Kbytes- and 1 Mbyte-memory. The smaller memory is not recommended, however, since many of the fast operations must be replaced by slower functions.

1.2 Installing the system on a Perq

Two standard procedures are available for installing the program on a Perq, depending on the physical medium used to transfer the files.

1.2.1 Ethernet

The standard procedure uses the Ethernet transfer program *CMUFTP*. The complete set of files needed to build the system is stored in the CMU-X, the Spice Vax/750. Two different command files allow to retrieve either the whole circuit-design system or DP alone. The two different procedures are listed below[1]:

- retrieve the whole circuit-design system (DP, WLIST, DWL, SL):

```
cmuftp retr /usr/dzg/ALL all
@all
```

- retrieve the graphics editor (DP) alone:

```
cmuftp retr /usr/dzg/DP dp
@dp
```

In order to use one of these procedures the Perq must of course have an Ethernet connection. The previous commands start Cmuftp and retrieve a command file that in turn retrieves all the modules in the system.

[1]These are POS commands; type them when you have the POS prompt

1.2.2 Floppy Disk

The alternative distribution medium is a Floppy disk, written in the standard FLOPPY format and available upon request. The disk contains the relocatable files, the Help and Command files for DP, some command files and the definition of the fonts used by the program.

The first file to be retrieved is called getdp.cmd; this command file will in turn retrieve all the others and generate the executable files. Typing

```
floppy get getdp.cmd
@getdp
```

will copy all the files from the floppy to the system disk and will generate the .RUN files.

1.3 Starting the program

Once the program is present on the disk of a Perq it can be started by typing

```
dp
```

This starts DP with default values for all the variables and switches. Such values are currently:

- Current Function = Line;
- Mouse Grid = 3;
- Display Grid = 6;
- Gravity Field = 5;
- Display Scale = 1;
- Current Thickness = 1;
- Current Font = 1 (Gacha7);
- Current Layer = 'STANDARD';
- Color = 1 (Black);
- Pin display OFF.

Typing a filename after "dp" on the command line starts reading commands from an alternate input file; see section 4.6 for further details.

To exit the program type **q**; answer **y** to the request of confirmation. Normal exit mechanisms, such as CTRL-C, are disabled by DP to prevent accidental termination.

2. DP: basic concepts

DP is an electronic-circuit drawing tool. It allows the user to interact with a circuit schematic, creating it from scratch or editing an existing one. The output generated by DP may be used as input to post-processors that perform error checking and generate a set of lists, such as a wire list, a stuffing list, a wrap list for wire-wrapping a board, etc..

DP runs on the Three Rivers PERQ computer and is entirely written in Pascal; it uses the high-quality graphic display and the pointing device (mouse) of the Perq. The mouse is the main input device; the keyboard is used for function selection and for typing strings.

DP is a general-purpose graphic editor that can be used to produce drawings other than electronic circuits. It therefore supports advanced graphics operations, such as instance transformations and generic curves, as well as the simpler graphics primitives.

2.1 The basic elements of a drawing

DP drawings are composed of a few primitive elements: lines, circles and arcs, splines, text strings, and pins. All the primitive elements may be combined to form larger elements, called symbols. Figure 2-1 shows examples of primitive elements in DP.

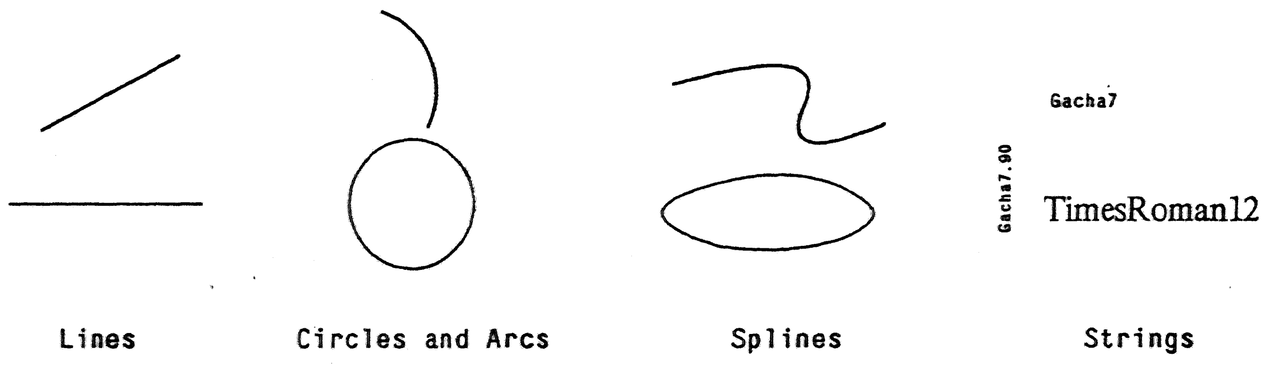

Figure 2-1: DP: primitive graphic elements

The following basic elements are used in DP:

1. Lines[2] with have any slope; the endpoints of a line have gravity.

2. Strings, composed of ASCII characters; all strings are truncated above a maximum length.

3. Circles, and arcs of a circle.

[2]finite-length segments, in reality

4. Splines, generic curves whose shape is controlled by a set of control points.

5. Pins, that is connection points; when used in a symbol pins have gravity.

6. Symbols: sets of basic items and possibly other symbols that may be used to represent electronic components. Symbols may be nested, thus allowing hierarchical drawings.

2.2 Status line and Prompt Area

The bottom part of the screen shows the status of the editor. The following items are displayed:

- command character of the function being executed;
- name of the function being executed;
- mouse buttons. Most functions perform different actions for different mouse buttons; the image on the right side of the status line describes the action of every button. The top square stands for the top (or Yellow) button, and so on.

Many commands that require the user to type text use the Prompt Area, located at the lower right-hand corner of the screen. When the program is waiting for keyboard input the area is highlighted; all the prompting commands may be aborted by typing a Ctrl-c (lower-case "c") while the area is highlighted. While the area is highlighted the string may be edited with the standard string edit mechanism; see section 2.5 for more details.

2.3 Mouse buttons

DP usually assigns different meanings to different mouse buttons. The current version of DP uses only three buttons of the mouse; in the case of 4-button mice, the Blue button performs the same function as the Yellow button.

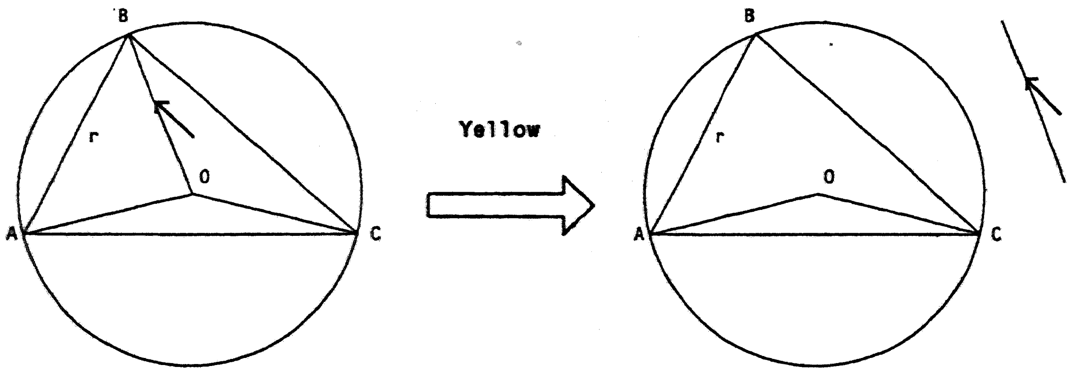

Figure 2-2: Moving one item at a time: Yellow button

All the commands that use mouse buttons to form a set of items use the following conventions:

1. The ***yellow*** button creates a set with the single element pointed to by the cursor; the Move

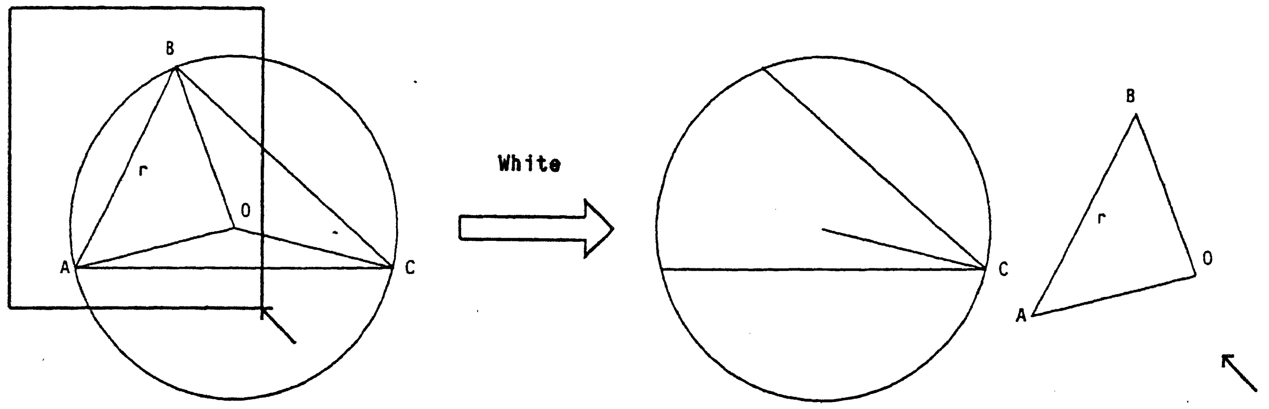

Figure 2-3: Moving the items within a rectangle: White button

command, for instance, moves a single item at a time when this button is used. If several candidates exist the smallest object is always chosen; this prevents large objects from "shading" smaller ones.

2. The ***white*** button creates a set with all the items entirely enclosed in a Rectangle[3].
3. The ***green*** button creates a set with all the selected items; for instance using the green button while in Move mode will move all the selected items.

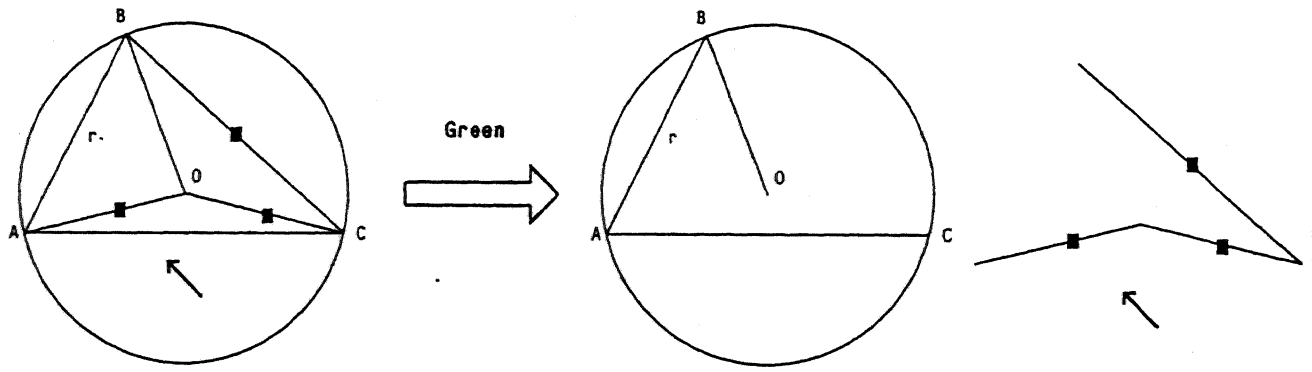

Figure 2-4: Moving the selected items: Green button

Figures 2-2, 2-3, 2-4 show the effect of a Move command on the same drawing when the three different buttons are used (the small arrow pointing NW represents the cursor).

Other functions use the mouse buttons differently; this is displayed in the Status line at the bottom of the screen, where each small black square represents a button.

[3]the rectangle is dynamically displayed while the button is held down

2.4 Selecting and deleting

Selection is used to form a set of items that should be handled together. Unlike the temporary sets created by the mouse buttons, the Selected set does not change unless explicitly specified by the user. Selected items are displayed with a small black square near the center of the visible portion of the object. In the case of strings, the whole area occupied by the string is inverted, so that selected strings are displayed as white text on a black background.

To provide some protection against mistakes during **delete** commands, deleted items are not immediately erased; they are kept in a special list that is neither displayed nor affected by normal operations. It is possible to undo the deletion, bringing back the objects.

Because of memory limitations, however, objects cannot be kept around forever. Every time a Delete command is issued all the items that had been PREVIOUSLY deleted are physically erased and their storage is released. In other words, only objects that have been marked as deleted since the last delete command may be undeleted.

2.5 Editing strings

A consistent string-editing mechanism is used throughout DP. This applies both to text strings and to all the commands that prompt the user for a string; in particular it can be used for strings in the Prompt Area. Any time a string is being edited, a special cursor[4] is displayed. Characters are always inserted and deleted at the position following the cursor.

The following characters have a special meaning during string editing:

- BS: delete the character immediately preceding the cursor.
- OOPS: delete all the characters preceding the cursor.
- RETURN: terminate the string editing, i.e. close the string and erase the cursor.

The Mouse may be used to position the cursor within the string and for other functions:

- The **white** button positions the cursor before the character pointed to.
- The **yellow** button positions the cursor before the character pointed to, if possible; the character following the cursor is then deleted. Holding the button down deletes several characters.
- The **green** button terminates the string editing, just as the RETURN key.

2.6 Windows

DP provides a powerful window facility that allows dealing with several drawings without improper interactions between their contents.

[4] a thin line between two characters

Each window has a *separate address space.* The following is a simple way to visualize that feature: a sheet of paper is dedicated to each window, and only the sheet corresponding to the current window[5] may be written on or changed. Operations on a window may affect only the items that have been drawn on that sheet of paper; for example a Select All operation selects all the objects associated with the window. Only some special functions (see section 3.4) may cause an object to "jump" onto another sheet of paper; these functions erase the object from the source window and write it into the destination window.

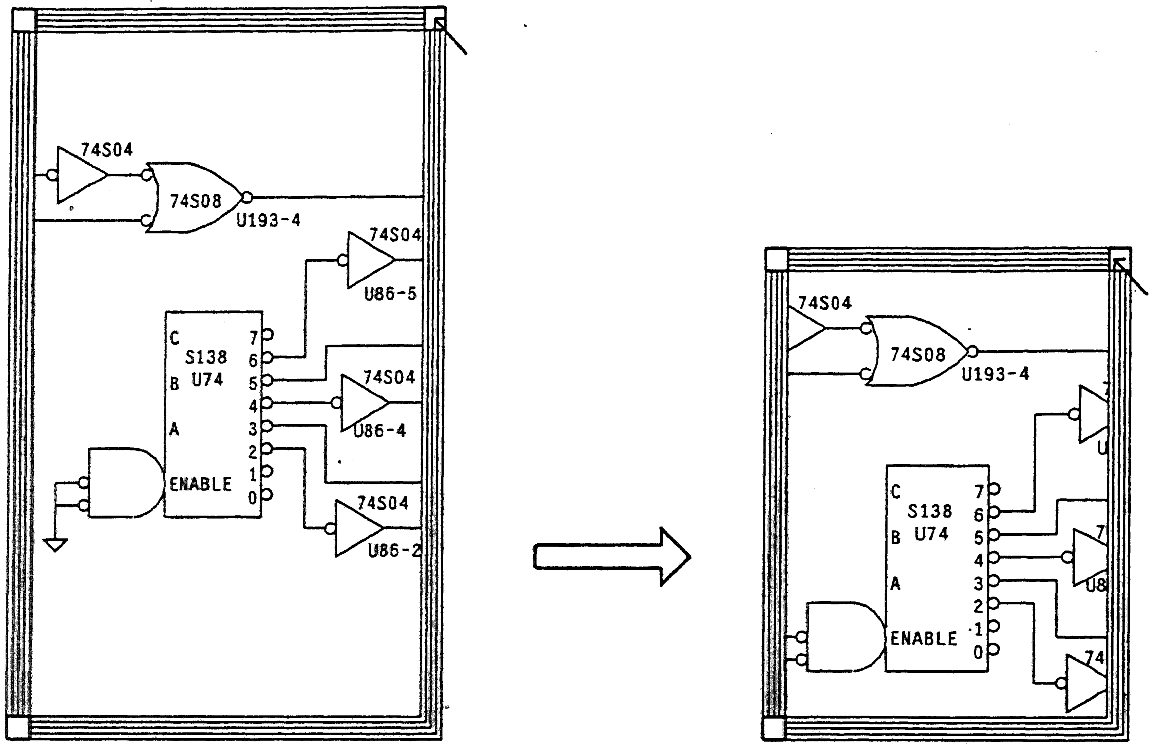

Figure 2-5: Changing the shape of a window

Windows may be manipulated through mouse movements only; the interaction takes place when a button is depressed ***over the border of the window.*** Different areas in the border have different functions:

- square at the **upper-right corner**: change the shape of the window. The corner follows the mouse until the button is released.

- square at the **lower-left corner**: create a new window, using one half of the area used by the current window. The new window has a different address space and is initially empty.

- square at the **upper-left corner**: delete the window. The window must be empty; the last window cannot be deleted.

- **gray border**: move the window in the screen, without changing its shape.

[5]the window the cursor is in

Figure 2-5 shows how to change the shape of a window by clicking with the mouse over the upper-right corner of the window.

Many commands are disabled when the cursor is not inside a window, since they would not be applicable. It is impossible, for instance, to perform a Select All operation outside a window. If this is the case, the Status Line will not show the new command and a 'beep' will be heard.

2.6.1 Windows and file-names

A region of the border of windows, near the lower left-hand corner; is used to display the file name associated to the window. This is the same mechanism used by Emacs[6]: if no file name is associated with a window when a file is input, the name of the file is displayed and is associated with the window itself. That file name is used as the default name when an output command for the window is issued (both Output and Hardcopy).

If a window has been modified since the last output command, a "*" is appended to the file name. The Quit command checks this flag for all the existing windows and issues a warning if some window is marked as modified. The algorithm used to decide whether a window has been modified is rather conservative (sometimes the window is flagged even if its contents have not actually been changed). The following rules determine the status of the window:

- the Output command clears the Modified flag;
- input commands do not alter the status of the flag;
- all the non-immediate functions set Modified to true;
- the immediate functions do not alter the flag, with the exceptions of Delete and Undelete.

2.7 Layers

DP supports a powerful layering mechanism: a drawing may be thought of as being composed of multiple layers[7] , independent of each other. This is intended to provide the same effect as multiple transparencies containing parts of a drawing: single transparencies may be added or removed, modifying the drawing itself.

The layer mechanism is especially useful for complex drawings that contain logically separated parts. An example is a Printed Circuit Board: only one layer at a time is usually being worked upon, but it is essential that all the layers be visible. Setting all the layers but one to read-only constitutes a protection against accidental changes to items on different layers.

The layer mechanism is also used by the circuit post-processors (see [4]) to quickly discard useless information. For instance, the post-processors totally ignore items in the "Comment" layer.

In the case of symbol instances, *layers act as a filter.* If an instance is on an invisible layer, for example, it will

[6]A screen-oriented, multiple-windows text editor

[7]The original idea of multiple layers was suggested by Joseph Newcomer

be totally invisible even if It contains items that belong 10 visible layers. If the instance is visible, on the other hand, the visibility of nested kerns depends oe each item's own layer.

Layers an be manipulated through the levers menu, accessible through the Unusual Commands menu. The ta>ef$ *menu* displays all the currently defined layers together with the sellings of the layer parameters.

The following parameters arc associated with each layer, and can be Individually set or reset A black rectangle in the Layers menu means thai the parameter is ON.

- Display; if ON, the layer is visible* If OFF, the items in the layer are Invisible and do not have gravity.
- Alter: if ON, the items in the layer are affected by DP operatiom; If OFF, the layer Is write-protected.
- Output; ;f ON, Ihe items in ihe layer may be written to- files, printed and sent to the plotter.

When a layer's set to invisible {Display OFF), it is automatically write~pn>toctod.

The La)ens nrenu also displays the ^^fflL^ffi* i,& ^e layer for acw items that will be created The Current Layer may *he* cfesged by citckmg the Default rectangle of a diffcreat layer.

The 1 a;ers ner/j hj$ x r e buirens at the betters T*!± ihe following functions:

- *C'IX?* :ne^ '!^er: ti?s iLiire 6 prompted fcn

tions.

2,8 Check-pointing

S*:^ r pi;w-i w::rg ^,i'c*s *ith DP tetc! :c- be &d; !ong, a check-pcindng mechariism has **been** [illegible]c^;ci. hacfi *ri^v* n-B i ;^r.t5r ^.cata i H:ji ;f; ±e :cunter is incremented e*ery time a new [illegible] *crt?:-:i* *[illegible]*: Ai?* n^-;A s xt*e \f'.cr ^|f^n numfeer of events* U12 whole conieois of the *r;&* ^r? ^rpur ^»i ^c d->^:^ ^-s^ci. T'h;s j^iure^ tiha: a recent copi of ihe drawing 3 ama>s available f^: r., *%nt* a^ ;1 :v^cr; r.^r.. n%c ;?t^i-p^^! :e r^nts 's fcrTrcd *h%* appending X K F to the file-same **associated with the window; thus the file "latch.dp" is checkpointed to "latch.dp.CKP".**

TV ^^;nf- ,?' r^ i^^ bc^et^ *ch^k^cixs TZ** ^^ ^tfdi^ec bv &e Unysual Ccntmands sienu (set **section 1.9).**

[8] default [illegible]

2.9 Fonts

DP supports multiple fonts. A "font" consists of the full specification of a font plus the name of a Perq font used to display it. The font specification is device-independent, i.e. it describes an "abstract" font such as Helvetica8 or TimesRoman12.

The description of an abstract font consists of four items:

Face:	one or two characters that describe the particular face used. Valid characters are ***{r b i}***: ***r*** means Roman (the standard face), ***b*** means **boldface** and ***i*** means *italics*. Some characters may be combined: for instance, ***bi*** means ***boldface italics.***
Size:	an integer indicating the size of the font. Although this number does not have a direct relation to some specific unit, small numbers mean small fonts. See [1] for more details on font lore.
Rotation:	an integer indicating the rotation of the font. The rotation angle is measured in minutes, starting from 0 for a standard-oriented font; a font that runs vertically up the page has a rotation of 5400 minutes (90 degrees).
Family:	the Ascii name of the family. A font family specifies a set of fonts with similar characteristics: Helvetica, Gacha, TimesRoman etc.

The previous entries are intended to specify the font contents of a drawing unambiguously, and for each output device the best possible approximation will be used.

The Perq font is a particular, device-dependent specification that tells DP what font to use when displaying a string on the Perq screen. It may thus happen that the same Perq font is used to display different fonts in a drawing, when there is no Perq font that matches exactly the different "abstract" fonts. The font information is however carefully preserved in all the drawings and used for different output devices.

3. DP command set

All DP commands are single keyboard letters. No Control key is required, in order to make it easier to type commands with one hand while holding the mouse in the other hand. Upper- and lower-case command letters are considered different Some of the Perq keys are labeled with a whole word; in the following they are listed in capital letters, e.g. "DEL".

Some commands are *Immediate,* they are immediately executed when the keyboard command is typed. Such immediate commands do not change the function being executed, and at the end of the operation the previous function will be displayed again in the Status Line of the display.

3.1 Basic items

1	*line* mode. Whenever one of the mouse buttons is depressed a new line is created; when the button is released the line is "frozen". Depending on the button, the line may or may not be constrained to be only horizontal or vertical. If a Gravity buttons is used, both endpoints of the line may be attracted by a gravity point[9]; the first endpoint is attracted when the button is *clicked,* the second endpoint is attracted when the button is *released.* The Current Thickness is used for the line,
a	*Asm̃ String mode.* Clicking a button causes a string to be prompted for and inserted at the current position. The Green button reads a sequence of strings and aligns them below the first one; the sequence is terminated by an empty line. Every character up to the end-of-liae is inserted in the string; strings longer than 80 characters are truncated. The Current Font is used for the string.
6	Grtfe mode. Qidang a button starts a circle with the center in the current position; releasing the button freezes the circle. The Green button creates an arc out of three points: the two endpoiBts, in counterclockwise order, and the center. The Current Thickness is used for the aide.
9	$nfijie mode. All the buttons but Green enter a new control point; the Green button creates a curve out of the set of control points. The endpoints of the curve will always lay on the first and last control points; at least two control points are required The Current Thickness is used for the spline.
p	*KM* mode. Every time a button is clicked a pin is inserted at the *cursor* position; the pin number is prompted for. If the Green button is used no prompting occurs, and the previous pin number incremented by 1 is used.
c	*Edt* items: change the shape of existing items. See 4J for the item-specific details of this command; is genera!, the new shape of the item is always redisplayed dynamically in order to provide an accurate visual feed-back.

cither Use *mipmint* of a segment or a *pia in* a symbol

3.2 Parameters and Fonts

- *(minus)*	choose the Current Thickness. The value of this parameter will be used for all the new lines, splines and circles[10],
n	use New Parameters for existing items. Alter some or all the parameters of already defined items; for instance, convert strings from one font to another or change the thickness of lines. A menu is used; the previous choice is always suggested as a default. The following parameters may be changed: • thickness (applies to lines, circles, splines); • font (applies to strings); • layer; • color (applies to all items except symbol instances).
f	choose the Current Font or enter a new font in the Font Table. A menu with all the active fonts is displayed; clicking over one of the black rectangles in the Menu selects the new Current FonL The last entry in the menu is labeled "New Font": clicking it will install a new font, reading a Perq font file if necessary, and use it as the Current FonL See section 2.9 for an explanation of the meaning of the font parameters.

3.3 Select and Delete

s	enter Select mode. New items are added to .the Selected list
z	enter Deselect mode. Items are taken out of the Selected list
S	select all the items in the current window. Immediate function.
Z	deselect all the items in the current window. Immediate function.
d	physically erase the previously deleted items; enter Delete mode, adding new items to the "deleted" list.
D	physically erase the previously deleted items; delete the selected items in the current window. Immediate function.
J	undelete the items in the "deleted" list (since the last D or d command), and select them. Immediate function.

[10] **The** valid range for the thickness is L.8 in the current **implementation**

3.4 Copying and moving

Move and Copy are the only commands that work across window boundaries, so that an item may b conveniently moved across windows.

m	*Move*: pick a set of items and move them until the button is raised. A new temporary set i created for each Move operation.
c	*Copy*: pick a set of items, make a copy, move the copy until the button is raised.
x	*Stretch*: pick a set of items and move them, stretching all the lines that are connected to the items. The current implementation stretches lines connected to lines, strings or symbols[11] This command tries to preserve all the existing connections and not to create new ones. Strings that represent signal names (on a line) or chip locations (on a symbol instance) are moved with the item they are attached to. *Not completely implemented.*

3.5 Symbol-related commands

These commands manipulate symbol definitions. See also section 4.4.

b	create a set of items and pack them in a new symbol definition. The symbol name is prompted for; entering an empty line generates an automatic name. If the name matches one of the symbols already defined, DP asks whether the old definition should be deleted. In this case references to symbols with the old name are changed to the new definition; this actually means ***redefining a symbol***, and the old definition is destroyed.
B	create a set of symbol instances and unpack them into their basic components; the instances themselves are erased[12]. This function leaves nested symbols and top-level items where the old instances used to be.
t	enter Transformations mode. Symbol instances may be arbitrarily transformed by means of rotate, scale and mirror operations; this command allows to apply a transformation to sets of symbol instances. A Menu is used to specify the desired transformation: a column of black buttons and text is displayed. Clicking over one of the buttons selects the transformation; some of the sub-functions prompt the user for the value of a parameter, e.g. the angle of rotation.

3.6 File I/O

These commands perform various I/O functions. The format of DP drawing files is described in [3]. The commands that deal with drawing files (I i O) automatically append the default extension to the user-typed name.

[11] A line is considered to be connected to a symbol if it ends exactly over a pin of the symbol

[12] Unreferenced symbol definitions are garbage-collected.

o	write all the items in the current window to a drawing file; the file name is prompted for, and if a file name is associated with the window it is used as a default. If a file with the same name already exists, it is first renamed with a dollar sign at the end; the file "test.dp" will thus be saved as "test.dp$". This is the standard mechanism used by the Perq editors; it always makes a backup copy of a file before overwriting it.
H	create a hard-copy of the drawing in the current window. A Press file is generated and left on the current directory; the filename is prompted for. The Press file consists of a single page, and may be shipped to the printer later on.
O	send the drawing in the current window to a plotter. The plotter must be connected to the Perq through the RS232 line. Only the HP-7221A plotter is currently supported.
I	read a drawing file and merge it to the items in the current window. To start from scratch, all the items should be cleared before typing this command[13]. The file name is prompted for; if the file does not exist an error message is generated. If the window is empty this command adds an offset to all the new items, so that the image is centered in the window.
i	read a single symbol definition from a drawing file. Both the symbol name and the file name are prompted for. If the symbol is not defined in the file an error message is displayed; if the search is successful an instance of the symbol is created and centered around the cursor position.
j	read a text file and display it as Ascii strings.[14] The Current Font is used for the new strings.
@	read commands from an alternate input file whose name is prompted for. The normal operation is resumed when the End Of File is reached; input is directed to the keyboard again.

3.7 Display commands

These commands affect the way the drawing is displayed; they do not change the internal representation of items.

R	redraw the whole screen; this command may be used if the display image has been damaged. Immediate function.
r	redraw the current window. Immediate function.
=	display the current window with a different scale. This command "zooms" the image, it

[13]This can be obtained by typing SDD

[14]This command is extremely useful to examine the output of a post-processor, for instance, together with the drawing from which the output was obtained.

	does not change the size of items; other windows are unaffected. The scale value is a number; the default scale is 1. A scale of 0.5, for instance, means that objects are displa half the real size. Immediate function.
' *(back-quote)*	Display Grid on/off; useful for aligning items. Immediate function.
#	display a black square ("diamond") at the connections of lines; two lines that form a con are not flagged. Immediate command; diamonds are not permanent items.
w	move the image in the current window; the image "follows" the cursor and is refreshed the end. The Green button uses a faster algorithm that requires less startup time.
~	enable or disable the displaying of pin numbers, and redraw the screen.
P	enable or disable the displaying of pins, and redraw the screen.
INS	insert a Mark at the center of the current window. Each window has a circular buffer (Marks[15]that are used to remember important positions in the drawing. Immediai function.
G	go to the next Mark in the circular buffer for the current window; that is, change th position of the window over the items. The next Mark is placed at the center of th window. This command allows one to "jump" between various places in a drawing Immediate function.
DEL	delete the previous Mark. Immediate function.

3.8 Mouse commands

g	change the mouse grid, that is the distance between the two nearest points the cursor can be in. High grid values make cursor movements gross; small values yield a smooth motion but less accurate positioning. The recommended grid value is 3; this is the best value to use when dealing with normal drawings. The smallest grid value is 1. The value of the Gravity Field is related to the current Mouse Grid: the recommended value is *gravity = (grid * 2) - 1.*

3.9 Unusual Commands

k	This command gives access to a set of rarely used functions; the previous value for the various parameters is always displayed as a default.

- **Checkpoint-value**: number of key-strokes between checkpoints. A checkpoint file is automatically written when more than the specified number of commands is typed in a window.

[15]The buffer is initially empty; there is no limit to the number of Marks in a window.

- **Display Grid**: alter the distance between the dots in the Display Grid. This simulates "graph paper" and is useful for aligning items.
- **Gravity Field**: size of the gravity field, in screen units. Smaller fields mean weaker gravity.
- **Clip on window when printing**: indicates whether the whole drawing or just the visible portion should be printed[16] . In the latter case the printed page will look exactly like the screen, without any centering or justification of the image.
- **Diamonds when printing**: indicates whether diamonds should be used at the connections of lines in the hard-copy.
- **Show invisible items**: redisplay the whole screen, showing all the invisible items (including items nested in symbols).
- **Toggle cursor shape**: toggle the direction of the cursor between North-West and North-East.
- **Layers**: invoke the special Layers menu (see section 2.7).

3.10 Miscellaneous commands

? print out internal information; primarily intended for debugging purposes. This command displays the following items:

- parameters of the current window;
- list of the Deleted items, if any;
- segment allocation table and number of free segments;
- Font Table, with the font numbers and the associated Perq font file;
- names of the Symbol Definitions, if any.

h, HELP type the help files for DP. There are currently three help files: introductory help, complete command set with a short explanation for each command, and changes in the latest version.

q quit DP. It asks for a confirmation (type 'y' if you really want to quit, everything else to remain in the program). If some window is marked as Modified and its contents are not null, a new request for confirmation is issued.

[16] applies only to Press files.

4. Advanced topics

This chapter contains some of the advanced techniques a designer needs to know in order to use DP as efficiently as possible.

4.1 The coordinate system

DP uses a standard-oriented, cartesian coordinate system to represent the objects in a drawing. 16-bit integer arithmetic is used throughout, and therefore the integers in the range [-32767..32767] are valid coordinates. No special meaning is assigned to the absolute value of the coordinates in a drawing; translating every item by a fixed amount does not change the drawing at all.

When DP is started the point (0,0) is placed at the center of the window. Both the window and the drawing may be moved, performing any arbitrary translation. As a quick "beam find" operation, specifying a scale of 0 (see 3.7) goes back to the initial position with the window centered around the origin and scale equal to 1.

4.2 The size of a drawing

DP does not restrict drawings to the size of a single screen. Using the window mechanisms one may create drawings that are about 64 x 96 times the size of the screen[17].

In practice, however, very large drawings are not recommended for output devices like the Dover printer. Large drawings must be either split into smaller ones or printed with small magnification. This does not apply to larger-paged output devices, such as plotters.

4.3 Editing items

The action performed by the Edit command depends on the particular kind of item being edited. Items are always opened for editing by pointing at the item with one of the mouse buttons depressed. Circles and Splines are opened by pointing at one of their endpoints; the whole item is "sensitive" in the case of lines, strings, pins, and symbols.

- ***Lines***: clicking over the line picks the nearest endpoint and moves it following the mouse. The line is dynamically adjusted and appears to follow the cursor in a rubber-band fashion. The same options are available as in Line mode: constrained/gravity, etc.

- ***Circles***: clicking over one endpoint with the Yellow button changes the radius; all the other buttons drag the endpoint, changing the subtended angle but keeping the radius constant.
  The circle is always scanned counter-clockwise; the First point marks the beginning of the visible portion of the arc and the Last point marks the end of the visible portion. When the order of the two points is swapped, the circle changes shape abruptly, e.g. from a short arc to an almost full circle.

[17] This means, theoretically, a drawing as big as 6000 sheets of paper.

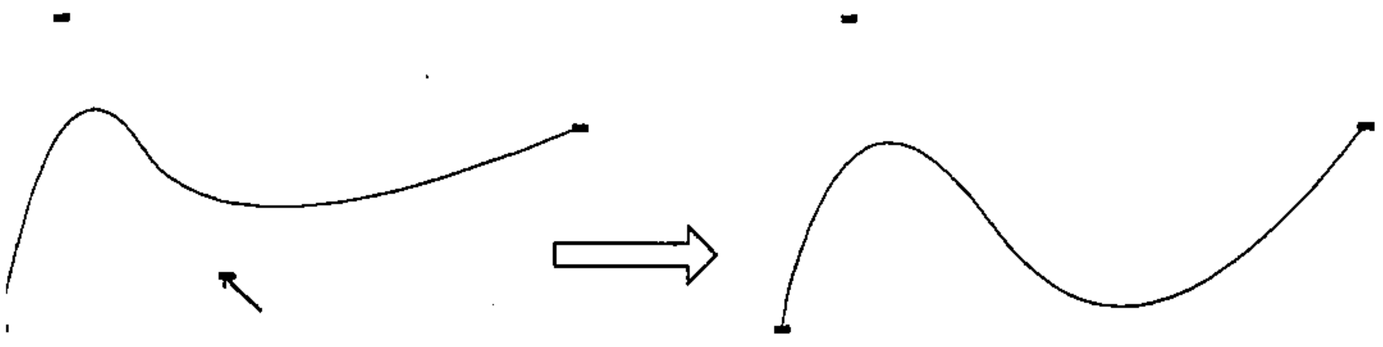

Figure 4-1: Editing a spline by moving a Control Point

If a circle is embedded in a symbol and the symbol is rotated of mirrored, the endpoints of the circle are also transformed. It may thus happen that when the symbol is unpacked the endpoints of a full circle are no longer at (Radius,0); they might be for instance at (0,Radius).

- *Splines:* after opening the curve for editing (button down over one of the endpoints) the Control Points are displayed. Moving the cursor over one of the control points locks the point to the cursor; the spline is recomputed when the cursor is moved. When the button is released the spline is frozen. Figure 4-1 shows how to edit a spline by moving one of its control points.

- *Pins:* change the position of the pin number. Clicking over the pin and moving the pointer moves the pin number in one of the four quadrants; Pin Numbers Display should be OR Notice that the display position for new pins is computed automatically when they are used in a symbol.

- *Strings:* alter the characters in the string. The string is displayed in the Prompt area and is opened for editing; see section 2.5 for more details. A string cannot be deleted this way: entering a null string simply puts the original string back.

- *Symbols:* display or alter the name of the symbol. The Green button "opens" the name and allows to modify it; this means that *all the instances* of the symbol will have their name changed, and the old name is deleted. The other buttons simply display the name of the symbol.

4.4 Symbols and Instances

A symbol is a set of items that are grouped together and represent a single object, such as a transistor or a NAND gate; the set is always identified by a unique name.

Although not clearly spelled out, the word "symbol" has been used in this document to mean "a symbol definition". The word "instance" has been used to mean "an instance of an already-defined symbol". To draw an analogy between symbols and programming-languagc concepts, a symbol (or symbol definition) is equivalent to the definition of a procedure; an instance is analogous to a procedure call

The symbol defines the structure of a graphical object: a set of basic items and possibly otlier nested instances. The definition is just a "template", since it specifies how to draw a graphical entity if requested; the definition itself does not add any item to a'drawing. It is only by creating instances of that symbol that items will be made visible on the page. See [2] for a discussion of symbols and instances.

An instance, like a procedure call, may specify the value of certain parameters, indicating where and how the graphical symbol is to be displayed. Each instance specifies the following parameters:

- X- and Y-offset: coordinates of the center of the symbol. This specifies the global translation that is to be applied to each item in the symbol.
- Rotation angle: the global rotation of all the items in the symbol This parameter is in minutes, a positive value meaning a counter-clockwise rotation. A rotation of 30 degrees clock-wise is thus specified as -1800 minutes.
- Scale: two values that specify the global scaling of the symbol and possibly mirroring transformations. A scale of 1 means do not alter the size of items.

4.4.1 Creating a symbol

The best thing to do when creating a new symbol (especially for circuit schematics) is to look closely at existing shapes, A grid value of 3, the standard value, is strongly recommended, A rather high scale, like 3 or 4, should be used when drawing the lines for a shape; when drawing very short lines Gravity should be offi

A good way to create a shape is to edit an old one. To do so, create an extra copy of the old symbol and unpack it; the basic components will be available for editing. When the shape is all right put down the strings** and the pins. It is usually better ta use the Rectangle button for the final Make Symbol command.

4.4.2 Symbol names

DP allows one to create a symbol without explicitly giving it a name; an internal name[19] is generated This happens when an empty line is typed as the symbol name.

Symbols with automatic names should not be used at the top level in a circuit drawing, since they cannot be retrieved by name from a drawing. They should be used inside other symbols; as an example consider an integrated circuit symbol. "Pins" in such a symbol are usually complex items: they have a pin, a string (the visible pin name), and possibly other items. Packing these items in a symbol is the basing naming convention for pins used by the circuit post-processors; automatic names are very handy for symbols like this.

is
Do this with scale I* in <wter to position the strings correctly

[19] Sudi as $$12:28:32; tills is called an *autmmk mm*

4.5 Memory allocation

The current operating system of the Perq restricts the total amount of storage that a program may allocate. DP tries to use as much storage as possible for drawings, but sometimes the available resources may be exhausted. Here are some suggestions about memory allocation:

1. Deleted items should always be physically erased when no longer needed. Simply deleting an item does not release the associated storage, since the item may still be "undeleted"; it is thus necessary to use Delete commands **twice**, since this physically destroys the deleted items. To get rid of a whole drawing, for instance, one should type "SDD": Select All, Delete All, Delete All (the second D releases the storage).

2. Working with many complex drawings at a time requires large amounts of storage; unneeded drawings should be promptly deleted.

3. Using many fonts requires several data segments to be permanently allocated; once a font has been installed it is not released until DP exits.

4.6 Alternate input for commands

There is a mechanism for reading commands from a transcript file instead of from the keyboard. A command file is read via the @ command and should contain the same commands that would be typed on the keyboard, plus cursor-positioning commands.[20] The alternate input file mechanism may conceivably be used as a "macro" facility to perform simple initializations of switches and parameters.

[20]Transcripts of DP sessions may be created with the $ command: this command alternatively opens and closes a transcript file, called "dpScript". The $ command is not yet officially supported.

I. Command set table

a	enter Ascii String mode.
b	create a symbol definition.
B	unpack symbol instances into their components.
c	copy items, move until the button is released.
d	delete items.
D	delete all the selected items.
e	enter Edit mode.
f	choose the Current Font or enter a new font
g	change the mouse grid.
G	go to the next Mark in the circular buffer.
h (or HELP):	type the Help file.
H	create a hard-copy of the current window.
i	read one symbol definition from a file, put it at the current position.
I	read a drawing from an input file, merging it with the current window.
j	read a text file, create strings.
k	unusual commands.
1	enter Line drawing mode.
m	move items.
n	force new parameters for existing items.
o	write the contents of the current window to an output file,
p	enter pin mode.
P	toggle displaying of Pin Positions.
q	quit DP.
r	redraw the current window.
R	redraw the whole screen.
s	enter select mode.
S	select all the items in the current window.
t	Transform symbol instances.
u	Undelete items deleted since the last delete command.
w	move the image inside the current window.
x	pick and move items, stretching connected lines.
z	enter deselect mode.
Z	deselect aO the items in the current window.
0	enter Circle drawing mode*
9	enter Spine drawing mode.
~	toggle the displaying of pins.
4	toggle the Display Grid.
=	enter a new scale for the current window.
#	display Diamonds at the intersections of lines.
-	enter the Current Thickness.
?	print out internal information.
INS	insert a Mark at the center of the curreRt window.
DEL	delete the last Mart

References

[1] Robert F. Sproull.
Font Representations and Formats.
Technical Report, XEROX - Palo Alto Research Center, October, 1980.

[2] William M. Newman, Robert F. Sproull.
Computer Science Series: Principles ofInteractive Computer Graphics.
McGraw-Hill, 1979.

[3] Dario Giuse.
DP - Format of the drawing files.
Technical Report, Carnegie-Mellon University, 1981.

[4] Dario Giuse.
SL: a hierarchical wire-lister for DP drawings.
Technical Report, Carnegie-Mellon University, March, 1982.