RSD-TR-17-82

A COMPARATIVE STUDY OF ROBOT LANGUAGES

Kang G. Shin[1]
Susan Bonner[2]

November 1982

CENTER FOR ROBOTICS AND INTEGRATED MANUFACTURING
Robot systems Division
College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109

---

[1]Department of Electrical and Computer engineering, The University of Michigan, Ann Arbor, Michigan 48109.

[2]Electrical, computer, and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, New York 12181.

TABLE OF CONTENTS

ABSTRACT

Most robot languages (RLs) have been developed as a need for them arises and are, therefore, geared toward specific robots and their applications. This article is intended to derive general desired features of an efficient RL of the future from an in-depth study of the existing ones.

1. INTRODUCTION

The availability of efficient means of communicating with industrial robots is a key factor to the success of contemporary, programmable automation. The communication means are commonly referred to as "Robot Languages" (RLs). Conventionally RLs have been developed in an ad hoc manner to meet the needs of a particular robot and application, thereby resulting in a situation where there are almost as many languages as there are robots. In this paper we intend to closely examine many (but not all) of the RLs now available, determine their desirable and/or undesirable features, and use this information to draw recommendations for an efficient RL of the future.

In all, fourteen RLs in use or under development in industrial or academic environments are reviewed, and principal characteristics are evaluated. Table 1 compares the industrial robots for which the languages were developed, and Table 2 contains background information on the languages.

The AL programming system at the Stanford Artificial Intelligence Laboratory (SAIL) is a high level RL with: ALGOL-like control and block structures; predefined data types for scalars, vectors, rotations, and positions; operations on those data types; local coordinate systems which may be affixed to one another, and the ability to specify motion in terms of objects grasped in the hands[1,2]. The Anomatic II Controller, commercially available from Anorad Corporation, provides a powerful numerical control language with programmable mathematical expressions, variables, jumps and subroutines, and the capability of self configuring[3] . AUTOPASS, a very high level programming system for computer controlled mechanical assembly under development at IBM, is oriented towards objects and assembly operations which enable the user to concentrate on the overall assembly sequence and to program with English-like statements using names and terminology that are familiar to him[4] . EMILY is an early attempt by IBM to develop a higher level workhorse RL with a reasonably simple pro-

cessor as an extension to their robot control language, ML[5]. FUNKY, yet another development of IBM, is an advanced guiding system which produces robot programs through the use of manual guiding and a function keyboard[6]. General Electric's RL, HELP, is a high level procedural language which: is relatively easy to learn to use; supports structured program design; supports simultaneous arm movement; is sufficiently comprehensive for robot operation, and has a special set of built-in functions/subroutines to support robot operation[7]. MAPLE, an RL developed at IBM, has a PL/1-like base language for computation and several extensions for directing a robot to carry out fairly complex tasks[8]. MCL, an extension of the popular APT numerical control language by McDonnell Douglas Corporation, is a high level language designed for the off-line programming of industrial robots and associated equipment under control of a robotic control system[9-12]. In the PAL programming system being developed at Purdue University, tasks are represented in terms of structured cartesian coordinates and every motion statement is a request to position and orient the robot such that a position equation is satisfied[13]. RCL is a command oriented motion control language under development at Rensselaer Polytechnic Institute (RPI) to program a sequence of steps needed to accomplish a robot task[14]. RPL, a FORTRAN -like user language developed by SRI International, is designed to facilitate the writing and debugging of application programs for material-handling, inspection and assembly tasks[15,16]. SIGLA, available from Olivetti with their Super-Sigma robot, runs using only 8K of memory and still provides features such as parallel task control and variable instruction sets for software tailoring[17,18]. T3, the language provided with the T3 industrial robot manufactured by Cincinnati Milacron, is a commercially available system which uses guided teaching and function buttons to program robot tasks[19-21]. VAL, a system and language for programming computer-controlled robots, has been developed over a period of several years at Unimation, Inc.[22,23].

In the following sections, the RLs are divided into five levels based on language features, the languages are discussed in terms of characteristics of a good programming language in general and additional characteristics important to RLs in particular, a programming example is included to provide a quantitative and comparative feeling among the fourteen languages, and then conclusion follows.

2. CLASSIFICATION OF ROBOT LANGUAGES

Robot languages are almost as varied as the robots they are designed to manipulate. Language emphasis varies from simplistic point-to-point motion to complex task oriented problem solving. The languages can be divided into five loosely formulated levels. Overlaps between levels occur but do not interfere with the basic features of each. A comparative summary of language features is included in Table 3. Figure 1 contains a breakdown of the fourteen languages into the five language levels.

2.1. Microcomputer/Hardware Level

This level consists of the lowest level of robot language control. The commands are highly dependent on the physical structures of the robot and, hence, there are no formal languages at this level. The emphasis here is on converting given joint coordinates (angles for revolute joints and distances for prismatic joints) to torques and forces on the motors and conveying sensory data to a higher level. This level performs analog-to-digital and digital-to-analog conversions between the control computer and the robot itself. The control of each joint is sometimes handled separately by a microprocessor or some hard-wired device[25].

2.2. Point-to-Point Level

Point-to-point languages are the most common type available on the market today. They provide programmed robot control by enabling the user to save a series of points obtained by guiding the robot through the motions. Usually the guiding is done by using a manual device to activate the joint motors in the robot and save desired locations. Sometimes, however, the guiding is done while moving the robot itself and saving points continuously (as with painting robots). Higher level guiding systems provide specialized function buttons which allow editing of programs and interaction with external signals.

Both the T3 language and IBM's FUNKY can be classified as higher level guiding systems. The T3 is operated manually through the use of a teach pendant. Motion can take place in either the cartesian, cylindrical or joint coordinate systems. Program revision is done by stepping forwards and backwards through the program steps and inserting and deleting steps where desired. Functions which act on external switches (e.g. a limit switch in the hand) can be associated with any step. The system can wait for external signals and signal events itself. Both of these features are very useful in a robot environment. FUNKY, which is similar to the T3 language, has a joystick to control the motion of the robot. Control of the system is similar to a cassette tape recorder. *play*, *erase*, *record*, *reverse* and *fast forward* modes are all comparable to their cassette player counterparts. These modes allow insertion and deletion of points and stepping forward and backward through the program. FUNKY goes a bit beyond the function capabilities of the T3, however, providing a command which uses touch sensors in the hand to center the gripper about an object and a command to operate an electric screw driver on an additional gripper.

Advantage of using point-to-point languages is that they are available and operational today. Once a task has been programmed, it can be repeated any number of times without operator intervention (unless something unexpected happens). Pro-

grams are also easy to debug because testing is constantly being done on the robot itself.

Disadvantages include little or no branching and subroutine capabilities, very little sensor interaction, emphasis on motion of the robot rather than the task to be performed, no software to handle emergencies, and no expandability to off-line programming.

2.3. Primitive Motion Level

The primitive motion level can be best described as "point-to-point motion in language form". It is to point-to-point languages as an assembler is to machine language. As with an assembler, the software helps shield the user from some of the more cumbersome aspects of the lower language. The following aspects are characteristic of primitive motion languages: simple branching has been added, subroutines (generally with parameter passing) are available, sensing capabilities are more powerful (sometimes much more), primitive parallel execution is introduced, and some attempts at frame definitions are made.

ANORAD, RCL, SIGLA, EMILY, VAL, and RPL are classified as primitive motion languages. All of the languages are based on interpreters or assemblers except RPL which has a compiler. They all provide simple conditional and unconditional branching. (EMILY, VAL, and RPL are also capable of do-looping). Motion can be specified using the joint angles directly or in cartesian coordinates. All of the languages provide absolute motion. (ANORAD, RCL, and VAL also provide relative and straight line motion). Subroutines can be called in all of the languages except RCL. Of these five only ANORAD and VAL have no parameter passing. ANORAD, VAL, SIGLA, and EMILY allow files to be included as executable code. ANORAD, VAL, and SIGLA achieve this by allowing all files to be run as subroutines. EMILY uses an *include* statement which causes immediate execution of the desired file.

Sensing features of the primitive motion languages vary widely. ANORAD, RCL, SIGLA, and VAL are provided only with simple, binary touch sensing commands and no vision commands. EMILY has several different kinds of tactile sensing, including touch sensors in the hand, a "whisker-like" wand sensor on one finger, a sonar proximity sensor, and an infra-red emitter and receiver which detect the presence of an object between the fingers. EMILY monitors these sensors in an on/off manner to aid in the assembly process, but it has no vision capabilities. RPL, on the other hand, has only the simplest binary touch sensing commands and a very complex vision system capable of taking a picture *(pictur)*, determining object features *(getfea)*, and recognizing an object as one of those defined in its data base *(recogn)*.

Simple parallel processing in the form of mutually exclusive operation of arms with limits and convergence points to ensure collision avoidance is used by both SIGLA and EMILY. SIGLA allows execution of several different files on several different arms all at the same time and provides an anti-collision command which sets up work boundaries for the different robots. EMILY uses the *synch* command to provide a convergence point for programs running simultaneously on different arms.

VAL and RPL have limited capabilities in defining and providing coordinate transformation capabilities. (See Ref. 24 for a detailed discussion on frames and transformations). They both provide commands to define frames, invert transformations, and multiply matrices. The use of frames and transformations in robot programming is not fully developed until the next level, structured programming level.

The primitive motion level provides advantages over point-to-point by adding branching and subroutine control. It introduces the concepts of parallel processing and the use of frames but does not develop them beyond a fairly primitive level. The use of sensor commands is also greatly increased.

Although some of the basic disadvantages of the previous level have been solved, the problems of emphasis on robot motion rather than the task to be performed, no

software to handle emergencies, and unsuitable expandability to off-line programming are still evident. More complex control structures and more effective parallel processing techniques are also desirable developments for the future.

2.4. Structured Programming Level

The structured programming level is a major improvement over the primitive motion level in that it incorporates structured control constructs into the RL and provides extensive use of coordinate transformations and frames. Other characteristics of these languages include complex data structures, improvements in sensors and parallel processing, and the use of pre-defined state variables.

HELP, PAL, MCL, MAPLE and AL are included in this level of languages. PAL is not truly a structured language, however it provides some structured control constructs and uses coordinate transformations to such a great extent that it should be included in this level. HELP does not make use of transforms at all but has structured programming constructs.

The languages within the structured programming level all have user definable subroutines with parameter passing (except PAL which has no subroutines and HELP which allows no parameters). They all, with the exception of HELP, are provided with complex data structures: PAL has transforms as its basic element, MAPLE allows the definition of points, lines, planes and frames. MCL provides for points, vectors, and frames and AL utilizes scalars, vectors, rotations, frames and transformations. AL and PAL allow explicit definition of transformations. MAPLE and MCL break transformations into a positional and a rotational part. MCL and AL also provide means of fixing frames together so that transforms applied to one part are automatically applied to another.

The term "state variable" is used to refer to a reserved word which represents some physical quantity associated with the robot. In PAL, the two major state vari-

ables are *arm* (the frame associated with the end of the robot arm with respect to the world) and *tol* (the frame associated with the tip of the tool with respect to the end of the arm). *arm* changes as the robot changes position and orientation. *tol* changes when the tool is changed. MAPLE has several state variables which it monitors and uses to perform tasks. Some of these are *leftfinger* (the position of the left finger), *wristroll* (the current roll angle of the wrist), *gap* (the size of the gap between the fingers), *hit* (a bit indicating that the fingers are contacting something), and *fgap* (the force being exerted on the fingers). The user must define state variables for HELP, MCL, and AL.

Sensor commands at the structured level are similar to those at the primitive motion level. There is a wide variation in the degree of sensing capabilities between the languages. PAL has no sensing capabilities. AL, HELP and MAPLE have touch sensing in the fingers. MAPLE has a proximity command. MCL has only simple binary touch sensing commands yet it is only one of the four languages with vision capabilities. The MCL system is capable of finding and identifying *(locate)* and inspecting *(inspec)* objects.

Parallel processing is expanded at the structured level. All four of the languages (excluding PAL) provide some sort of parallel execution. All have semaphore primitives that are activated only when a given event occurs. MAPLE has an *in parallel* construct for use when the order of execution of commands is irrelevant. AL has the high-level *cobegin* and *coend* constructs for synchronization between two robot arms. In all of these cases, the user is responsible for collision avoidance.

Motion on the structured level is defined in terms of transformations on the frame of the robot hand. In PAL and AL motion is specified directly in terms of the transform. PAL uses mathematical symbols of + and - and the *mov* stack to cause motion, whereas AL uses the more understandable *move to* statement. AL motion can be made more specific through the use of clauses which define intermediate

points, approach vectors,velocities and durations. MCL and MAPLE motion is separated into rotations and translations. The MCL *goto* statement will perform a translation and rotation but both must be specified (as a point and two vectors). MAPLE uses *move by* for translations and *rotate by* for rotations. HELP is an exception, providing motion in terms of the more primitive joint coordinates.

The structured level does much to aid program understandability and task-oriented programming. It provides more sophisticated parallel processing techniques and introduces the concept of state variables. Point-to-point programming has been replaced by manipulation of object frames. The structured languages themselves are more powerful than those at previous levels because of the complex data and control structures available. Off-line programming is more feasible at this level as long as relational transformations are accurate. Any discrepancy between the model and the robot environment can be expressed in terms of a transform.

The major asset with the languages at the structured level is also their major problem. Coordinate transforms allow many advantages to robot programming but they are difficult to understand and use. Even structured programming techniques require more education in the user than primitive branch commands. This level is more advanced than the primitive level but is also less feasible for robot applications today; (hence, the non-commercial origin of languages in this class). Questions which are still left unanswered at this level are: Can collision avoidance be provided with parallel processing?, Can we still maintain task-level operation and simplify the coordinate transformation problems? and Can simple decision making capabilities which allow the robot to recover after unexpected events be added successfully?

2.5. Task-Oriented Level

The task-oriented level of robot programming is a yet unachieved dream for a truly task-oriented language which conceals low-level aids like sensors and coordinate transformations from the user. AUTOPASS is proposed by IBM to meet these criteria. It is designed to resemble assembly instructions that might be given to a human.

AUTOPASS uses high level commands such as *place* object1 *on* object2. Execution of this command involves finding and identifying object1 and object2, determining a pick-up point and vector for object1, moving to pick up object1, deciding where on object 2 to place object1, placing object1 on object2 and remembering the new relationship between them. This is easier said than done. AUTOPASS absolutely requires a world modelling system to keep track of objects. Ideally this would involve vision location and identification and tactile sensors for help in locating and picking up objects. AUTOPASS must be capable of making assembly-oriented decisions, such as, how to pick up an object.

AUTOPASS commands are divided into four types: State change statements (such as *place)*, Tool statements (such as *operate)*, Fastener statements (such as *rivet)* and Miscellaneous statements (such as *verify)*. AUTOPASS statements mean precisely what you think they should mean and are, therefore, easy to understand and use. The high level of the statements, however, leads to ambiguities between the user's intended actions and how the robot interprets them.

In order to alleviate these ambiguities, the AUTOPASS system proposed is designed so that program debugging will proceed interactively with the user. Commands are interpreted into lower level code and the user must verify the validity. The user can alter any segment he wishes and the compiler can question the user about any ambiguities in the AUTOPASS code.

The AUTOPASS system is a highly task-oriented system with very English-like commands. The high level of the commands, however, necessitates the use of a complex world modelling system, artificial intelligence for decision making, and an interactive debugging system. AUTOPASS is unfortunately not implemented and still does not solve the problems of collision avoidance and emergency decision making.

3. COMPARISON OF LANGUAGE FEATURES

In the analysis and development of an RL, it is essential to consider several factors other than specific language features. These factors are more general in nature than the exact syntax and abilities of a language, but they are of equal importance to its success in an industrial environment.

The five language levels introduced in the previous section are compared and evaluated on the basis of (i) the features which apply to all programming languages in general, and (ii) some additional features which apply to robot languages in particular.

3.1. Characteristics of a Good Programming Language

Pratt[26] has cited six characteristics of a good programming language. These are features important to both programming languages in general and robot languages in particular.

A. *Clarity, Simplicity and Unity of Language Concept*

At the point-to-point level, the concept is extremely simple. Points are shown to and remembered by the robot. Both T3 and FUNKY have external function buttons and push-button debugging facilities. These languages are certainly clear and simple (hence, their relative success in industry today) but they do not have the programming power of the higher levels.

Primitive motion level languages have a tendency to have a great number of commands and few control constructs. Although the commands themselves are clear and simple, there are so many of them (and not all are exceedingly useful) that the user may get confused. The primitive motion level contains a hodge-podge of commands from the point-to-point and structured programming levels. This leads to a lack of unity.

Of the languages on this level, VAL is a good example of these inconsistencies. VAL users are provided with a manual teach mode characteristic of T3. They are also provided with the ability to define transforms and frames as with the structured level of languages. VAL has six different commands to perform motion from point A to point B with slight variations in the technique. The over-abundance of commands and this lack of unity in concept is a result of the fact that VAL was developed (as with many other primitive motion languages) on a dynamic basis. Commands were added as a need for them arose and the language structure was not able to handle the extensions in a reasonable manner. All four of the other languages in this group suffer from the same problem. RCL and EMILY provide several different commands to perform a branch, SIGLA is a language with so many different and specific commands that all users can only have a small subset. RPL has an entire package of user callable routines written to allow it to interface with a robot.

Languages at the structured programming and task-oriented levels were developed with consistent programming language characteristics in mind. The use of structured programming and data structures eases the demand for specific commands in the language. Coordinate transformation representation of points leads to a more general way of expressing motion and, therefore, cuts down on extraneous motion commands. The use of clauses in a single command to specify different aspects of motion greatly increases the generality of the basic move statement and eliminates the need for additional statements. All motion in AL is achieved by the *move* statement followed by optional clauses specifying velocity, acceleration, path

to follow, etc. MCL uses the *send* and *receiv* and a specification of which device to transfer data between devices. MAPLE provides both absolute and relative motion by allowing either *to* or *by* to be used in conjunction with the *move* and *rotate* commands. PAL specifies motion very simply by creating transformation matrices which solve equations. This allows PAL to cause motion using its mathematical notation. AUTOPASS provides all of the structured programming constructs but invokes certain restrictions in order to provide a higher level of commands. These languages are structured and consistent but their success in industry has yet to be proved.

B. *Clarity of Program Structure*

Structured programming techniques were developed to make programs easier to comprehend and debug. The simple conditional and unconditional jump statements were replaced by more "English-like" structured condition statements like "while-do" and "if-then-else". These allow the programmer to use more familiar logic to produce programs and debug them. The languages at the structured programming level all make use of these constructs. (AUTOPASS has them, too, but use is restricted). The primitive motion level languages have only simple *goto* and *if* statements. This decreases the clarity of the program structure itself. It should be noted, however, that structured programming techniques are more sophisticated and require more time to master than simple jump statements. For a trained computer expert structured programming techniques are essential, but they may not prove as cost-effective for application programmers in the field who probably do not have as appropriate a background.

C. *Naturalness for the Application*

Point-to-point languages are hardly languages at all. Points are remembered and function buttons take the place of programmed keywords. This method has proved quite successful for many robot applications.

Primitive motion level languages were developed because of the shortcomings of point-to-point methods. The language itself becomes more important than recording single points. Some languages at this level are more understandable than others, however. SIGLA and ANORAD have two letter commands which give little or no indication of what the command means. RPL, VAL, EMILY and RCL command names all make an attempt to indicate the function of the command but are limited to six characters. This is more effective than two, but still detracts from the understandability of individual commands. Control constructs at this level are only slightly more sophisticated than at the point-to-point level.

The structured programming level allows the use of any length keywords and variables. This not only makes the commands understandable but allows the user to call his parts by self-explanatory names. They introduce use of complex data structures like vectors and frames which are very useful if you can understand them but may not be easily comprehended by the average robot programmer. MCL still has the six character limitation of the previous level. It also is based on an already established machine tool programming language, APT. This causes the disadvantage that many MCL commands are useless additions for the robot programmer. However, MCL has the advantage that APT is currently widely used. Many related programs have already been developed in APT and machine tool operators are familiar with the language. It is believed that a natural extension of APT to include robot commands will be more easily assimilated into the market. PAL is based on matrix mathematics and only allows two character variables. It is, therefore, only suitable for use by those very familiar with robot control through the use of transforms.

AUTOPASS is a task-oriented language. It is very natural to use as the commands have "English-like" syntax. The use of transforms is hidden from the user. This makes programming in AUTOPASS easy for any type of user. The high level of the language however, creates ambiguities and therefore, the user must be able to understand a lower level language used in debugging.

D. *Ease of Extension*

It is extremely important that robot languages be easily extensible. They should have a modular and expandable structure so that they meet the needs of robotics today and can be easily extended to handle the needs of the future. As new applications for robots are discovered and sensing devices are improved, the language should be able to easily expand to meet these new needs and abilities. The most vital way to provide expansion easily is to provide user defined subroutines which allow the definition of further commands. Parameter passing is fairly essential to expandability to avoid the problem of re-use of global variables. Subroutines which provide expandability should be allowed to be nested to several levels in order to allow the language to grow upon itself.

The point-to-point level languages are difficult to expand because they have no subroutine features and use hardwired function buttons instead of software routines to perform tasks. Primitive motion languages vary in their extensibility. The VAL interpreter is contained in programmable read only memories (PROMs). Any changes to the language require re-programming of the PROMs. This does not lend itself to easy expandability. SIGLA has many commands from which the user is allowed a small subset. The language is constantly expanding to meet specific needs of different applications. SIGLA also allows user defined subroutines which become user-defined commands. Expansion is limited, however, by the amount of space available on the system. RCL has no subroutine capabilities and therefore, extension would involve rewriting part of the interpreter. RPL and EMILY have subroutine capabilities

and allow parameter passing. This leads to relatively easy extension from the user's and the system programmer's point of view. All of the structured programming level languages, except PAL and HELP, have powerful subroutine capabilities including parameter passing and nesting to several levels. They are, therefore, easily expandable. AUTOPASS has the same subroutine calling capabilities as the structured level languages.

E. *Debug and Support Facilities*

Efficient debugging and support facilities are extremely important in the development of robot programs. Facilities which allow quick and efficient alterations to produce working application programs are just as desirable as useful features of the language itself. Today, debugging time is especially crucial because programs are developed using the robots themselves. Off-line programming, which uses graphics simulations to help in program development, will make debugging features less critical because the robot may not be in use as program development takes place. Table 4 compares the debugging facilities which are available with the languages analyzed. The lower level languages have much more extensive facilities because they are in actual use today and debugging must be done on the robot. Structured programming level languages are designed with off-line programming in mind and, therefore, provide fewer hands-on debugging aids. AUTOPASS provides a complex off-line debugger which acts interactively with the user to produce executable code.

F. *Efficiency*

The efficiency of a programming language depends upon the ease with which programs can be developed *(programmability)*, and the ease with which the language can be adapted to a new environment *(transportability)*.

• *Programmability:* Programmability is a measure of the ease with which users of an RL can produce correct, executable code. The best basis for measuring programma-

bility is time. The amount of time that it takes to train an unfamiliar person to use the language and the time it takes a user to write a program to perform a task are especially important. It is very difficult to determine a quantitative measure of the programmability of one language versus another. Perhaps the most accurate method would be to take many programmers of different programming ability and measure the amount of time that it takes them to program the same task in different languages. In order to compare the languages most accurately, many different tasks would have to be programmed by each programmer. This method is only as accurate as the programmers involved (i.e., the same person may have different abilities on different occasions dependent upon his psychological and physical condition, etc.). The more programmers involved and the more tasks programmed, the more accurate the results would become. This method can develop a somewhat quantitative measure of programmability.

Other considerations for programmability include the format of statements (are they easy to understand and remember?), the length of a program designed to perform a specific task, and the types of data structures used. In an attempt to compare the programmability of the languages, an example of palletizing blocks from a conveyor belt is selected and studied in the next section.

• *Transportability:* A transportable RL can be easily adapted to be used by any robot and on any computing facility. A measure of the transportability of a language to a system is the amount of time it takes to complete the interface between the language and the system.

Because of the number of computing facilities and robot mechanisms available, an intermediate step between the compiler and the system is one way to ensure widespread transportability with minimal effort. This intermediate result (called a *p-code)* is a low-level pseudo-language which can be easily translated into the base machine language for the system. Another method would be to require the interfacer

to rewrite only the lowest level of routines to provide the interface. Using this method, no p-code interpreter must be written and with proper choice of routines, interface would be simple.

For an interface with a robot, a p-code language or low level routines would indicate, for example, the positions and orientations of the robot. This would require robot software to determine displacements for revolute and prismatic joints and the torques and forces required to produce the desired displacements. Any further specialization on the part of the p-code or routines would make it dependent upon the configuration of the robot. With a modular structure, however, lower level routines containing the transformations for different robot configurations can be included in the packages for different robots. This would allow the high level language to determine displacements for a specific robot configuration. The forces and torques must still be left to robot software because of the enormous variations in the physical robots themselves. RLs today are very specific to the robot and computer system they are being developed on. With the exception of an effort at Stanford University to make AL transportable, very little has been done in the development of highly transportable RLs.

3.2. Additional Characteristics of a Good Robot Programming Language

There are several properties that the developer of an RL must consider in addition to those presented by Pratt[26]. These are areas which apply specifically to RLs but not to programming languages in general.

A. *Decision Making Capabilities*

It is desirable to equip robots with capability of making intelligent decisions. Some ability to make decisions already exists in robotic vision systems capable of recognizing objects (like those provided with the languages RPL and MCL). This is only very primitive and limited to the set of objects within the data base for a particular assembly. The next higher level of vision recognition is to allow parts to obstruct each other in the camera's eye. The robot must be able to differentiate between objects and move the top one to reveal those underneath[27]. These are fairly simple capabilities which are very useful for assembly tasks, but perhaps the most important types of decisions involve the more complex problem of robot safety.

In order to function with the most efficiency and safety in real time, the robot system should be able to handle unexpected changes to the robot environment[28,29]. This requires some sort of dynamic model of that environment, constant monitoring of this model for unexpected changes, and the ability to make intelligent decisions based on any changes. The robot must execute its task concurrently with the monitoring system. When an unexpected change occurs, the monitor interrupts task execution and attempts to return the model to its expected state. For example, if a part needed in an assembly operation is not positioned correctly, the monitor will stop execution and perhaps use the robot to reposition the object. When the world model is back in its proper configuration processing can be resumed from where it left off or from some point in the future depending on how intelligent the monitor system is. (i.e., The monitor system can simply have the robot reposition the object and return to the interrupted position or the monitor can attempt to use the object for its intended purpose since the robot is holding it anyway and then return control to the robot execution program at some later point.) If the change in the model happened to be a person who had moved within the reach of the robot, the monitor would simply hold execution and wait for the expected world model (without person) to return

before allowing the robot to continue moving. This type of concurrent monitoring system requires complex sensing and decision systems which have not yet been fully developed.

Of the languages analyzed, only AUTOPASS has any simple decision making capabilities based on a dynamic world model of the robot environment.

B. *Interaction with External Devices and Sensors*

It is extremely important that an RL be able to handle interaction between the robot and external devices. It is very rare that a robot will operate as a stand-alone unit, independent of the actions of the devices around it. Usually robots are required to work in conjunction with other devices, such as conveyors, vision systems, machine tools, and even, sometimes, other robots. Desirable language features include messages which can be sent and received between devices and commands which allow concurrent activities of different devices.

The languages at the point-to-point level allow only very primitive interaction with external devices. FUNKY provides control of the gripper and a screw driver as its only "external" devices. It also allows monitoring of tactile sensors in the fingers. T3 allows input from and output to signal lines indicated by numbers. T3 provides a simple but fairly effective means of interacting with any external device.

At the primitive motion level, interaction ranges from practically none (as with FUNKY) to interaction with a fairly complex vision system. RCL and EMILY only have the ability to monitor inputs from the outside world. VAL, SIGLA and ANORAD have abilities similar to T3. RPL has vision interaction commands in addition to I/O signals which can be sent and received. Parallel processing at this level, if it does exist, is restricted to concurrency between more than one robot. It should be noted, however, a device that was signalled to begin by the robot program can be active as the program continues.

The structured level languages have abilities which range from no interaction at all (PAL) to allowing portions of code to apply to any device in the robot environment (MCL). MAPLE can only monitor inputs. HELP and AL allow signalling and waiting on external events. MCL can send and receive signals from any device. It also allows one program to control the actions of several different devices. Parallel processing, if provided, is in the form of waiting and signalling events. These can be used effectively for the control of external devices.

The task-oriented level provides commands which imply interaction with external devices and concurrent control of devices but they do not have explicit commands to send and receive signals. AUTOPASS has an *in parallel* statement which will execute commands at the same time and several statements, such as *operate*, which imply the use of external devices.

C. *Compilers versus Interpreters*

In robot applications, an interpreter provides many advantages over a compiler. An interpreter executes code as it is encountered, regardless of what happened before or after. A compiler passes through the code more than once before it generates executable code for the statements. It is easier to change a program which runs on an interpreter quickly because changing one statement does not require recompilation of all of the statements. Interpreters allow easy partial execution of sections of code (i.e., does not have to check the entire program) and hot-editing changes into the code as the program is being executed. Interpreters are generally slower, however, at run-time because parsing and interpretation must take place then. It is also much more difficult to implement structured control constructs with an interpreter because of the more complex branching requirements.

A popular solution to this dilemma is to provide a structured level language which compiles into a lower level primitive language. This allows the programmer to use the advantages of an interpreter during the debugging stages. The major

disadvantage to this compromise is that the original structured code is not always the true source code for the lower level program once it has been debugged. Little attempt has been made to recreate a structured level program from the lower level code. Languages which utilize this solution include FUNKY, EMILY, RPL, AL, and HELP. AUTOPASS and MAPLE use a variation on this theory involving a high level interpreter which creates lower level interpretable code. Most of the other languages are comprised of a single interpreter only; T3, VAL, RCL, SIGLA, PAL and ANORAD. Only MCL is based on a single compiler.

D. *Concurrent Operation of Devices*

In order to achieve maximum efficiency in use of the computing system and to allow the robot(s) and peripheral devices to perform in a synchronous manner, parallel task execution must be provided by the RL. Currently RL facilities to perform concurrent operations have two forms: signal/wait primitives and parallel block execution. Both are very useful in robot operations.

Concurrent operation of tasks using signal/wait primitives involves the use of events which can occur only when certain conditions are met. An event is "waiting" for a condition to become true before it can begin execution and it is "signalled" to begin when the condition becomes true. An example of this is a conveyor belt which has been halted and is waiting for a signal from the robot in order to resume movement. The code to control the motion of the conveyor is an event and it is waiting for a "go" condition from the robot. Of the languages which provide concurrent operations, most allow specification of signal/wait primitives: AL, VAL and HELP use *signal* and *wait*, MAPLE provides a *when* structure, and SIGLA uses *ez* (signal) and *ew* (wait).

Parallel block structure, a more sophisticated technique in parallel processing, involves the parallel execution of statements or blocks of statements by different devices. This type of concurrency can be used to control more than one robot arm and peripheral devices at the same time. Only those languages with more advanced

parallel processing abilities provide this type of concurrency: SIGLA allows parallel execution of any number of files, AL provides a cobegin/coend block structure, MCL provides an *inpar* statement for running more than one task, and AUTOPASS allows execution of many statements with the *in parallel do* construct.

E. *Interaction with World Modelling Systems*

The ability to interact effectively with a world modelling system is the key to the development of a truly intelligent robot programming system. Without a world model, the robot is essentially blind to its environment. The user must specify any changes that the robot's activity makes to the environment. A dynamic world modelling system would not only keep track of the initial state of the robot environment but would adapt changes made by the robot into the model. Some work has been done in the area of advanced world model development[30,32] but a functional dynamic world modelling system is not currently available.

AUTOPASS is the only one of the languages which requires a highly intelligent world model. In order to be successful, the AUTOPASS interpreter and the world modelling system would have to pass information about object placements and attachments between one another. This interaction should be hidden from the user. AL makes use of simple world modelling techniques through the use of a program called POINTY. POINTY is used to determine initial positions and orientations of objects in the robot environment and pass this information to the AL program. Any changes to the world model, however, must be specified by the user (the *affix* and *unfix* statements are provided for this purpose). POINTY positions and orientations are specified by using a manual pointing device. The use of such a device is time consuming but it is an early attempt at world model development. MCL uses vision to provide world model information to the robot program. Object position is returned implicitly and frames are calculated relative to the new data found. All picture taking and manipulation must be specified by the user. RPL uses "blobs" to represent

object features. They have not managed to attach blobs together, however, to specify a single object.

Although the development of complex world modelling systems is far in the future, some of the robot systems today have simple world modelling facilities under development. The most important fact to consider in the current development of a robot programming language is the ease of extension of the language to accept information from a world modelling system when a more powerful one becomes available.

4. A PROGRAMMING EXAMPLE FOR COMPARING LANGUAGES

In order to provide a quantitative and comparative feeling between the languages we selected the following programming example:

> "A robot is assigned to pick blocks off of a conveyor belt and deposit them in a pallet with a three by three array of positions for the blocks. It is assumed that the blocks are precisely positioned on the conveyor and the conveyor stops automatically when a block has arrived at the pick-up point. Re-activation of the conveyor is done either manually or via signals from the robot program." (See Figure 2)

This example is not only typical of robot operations today, but requires several important language features such as subroutine usage and interaction with external devices and sensors.

The example task is programmed in the fourteen different languages, and an attempt is made to derive a quantitative comparison of the programmability of each language as it relates to this specific example. The term "programmability" refers to how easy it is to program in the language. Measures such as program length, development time, readability, ease of extension, range of users, ability to program complex tasks, and necessary support facilities have a direct or indirect influence on programmability.

We have quantified these qualitative measures of programmability by ranking several of their aspects. By examining the rankings that each of these languages has

achieved in this example, it is possible to establish quantitative comparative relationships between the languages themselves. In general, aspects which are lower in number (close to 1) are more desirable features and those which are higher are less desirable.

With the quantification below Table 5 is constructed to show a quantitative comparison between the languages.

• *Number of Instructions:* Of the seven measures, this is the only absolutely quantitative one. It is the number of instructions in the program (excluding comments). In some cases, two numbers are provided. The first represents the actual number of statements. The one in parentheses is the number of statements that the program could be reduced to if extraneous assignments are removed. (The assignment statements are added to increase readability).

• *Development Time:* This measure is divided into seven aspects:

1. fast and simple
2. quick, but requires some thought and/or intelligence
3. quick with background in structured programming and knowledge of transforms
4. quick but instructions are not easy to read
5. encumbered by awkward control constructs
6. encumbered by complicated coordinate transform usage
7. requires extensive knowledge of coordinate transformation arithmetic.

• *Readability:* Since readability is also quite varied and complex, it has been divided into several sub-measures which effect the readability of the languages. These include understandability of instructions, use of structured format, and flexibility of user-defined variable names.

- *Understandability of Instructions:* Understandability is categorized into

seven aspects:

1. reads very much like "English"
2. uses function button control
3. instructions are words in English but does not read like english
4. readability is markedly improved through proper choice of variable names
5. instructions/variables limited to 6 characters
6. instructions/variables limited to 2-3 characters
7. instructions are not "English-like" at all.

• *Structured Format:* The degree of program structure in a language can aid in readability by enabling a known format for all programs and forcing variable declarations. There are three degrees of structure:

1. structure is an inherent language feature
2. can structure programs but are not forced to do so
3. language is inherently unstructured.

• *Flexibility of Choosing Variables:* The following categories were used:

1. essentially unlimited variable naming
2. variable names limited to six characters
3. variable names limited to 2-3 characters
4. variables available but only by number
5. variables not available.

• *Ease of Extension:* There are four degrees of expansion in the fourteen languages:

1. subroutines or other extension facilities available
2. expandable through subroutines with some loss of "English-like" syntax
3. Somewhat expandable through subroutines but nesting is prohibited
4. must rewrite code to add instructions.

• *Range of Users:* The users are divided into five levels:

1. inexperienced person
2. NC programmer or machine operator
3. person with some programming experience
4. person familiar with structured programming and/or transformations
5. person with extensive knowledge of transformations.

• *Programming Complex Tasks:* The languages vary in their ability to program different tasks. Some are limited to very simple pick and place operations while others are capable of complex tasks, such as fastening and compliant motion. This measure is not directly related to the example but it is included for comparative purposes. The task ability of the languages is divided into four levels:

1. capable of programming complex tasks using multiple arm operations such as fastening
2. capable of programming complex tasks using visual feedback, such as part recognition
3. capable of programming complex tasks using touch and force feedback sensing, such as compliant motion
4. capable of only simple tasks with minimal or no touch sensing in fingers.

• *Necessary Support Facilities:* Although these measures are not directly related to the above example, they re included in the table to portray a more realistic view of what languages are actually feasible for use today. The measures of support considered are:

• *Computing Power Required:*

1. micro-computer or small mini-computer
2. mini-computer
3. combination main-frame and small mini-computer.

• *Sensing Ability Needed:*

1. Only simple touch or no sensing required for

complete operation

2. proximity sensing specified in addition to touch sensing.

3. vision in the form of part recognition required

4. complete dynamic world modelling system needed for operation.

• *Availability:* Is the language as it is presented operational at present?

1. operational and available commercially

2. mostly operational but not commercially available

3. only a small subset is operational

The table presents AUTOPASS as the most programmable language, however it needs the most complicated and undeveloped support facilities; T3, FUNKY, AL and MAPLE indicate fairly good programmability; VAL, EMILY, and HELP fall into an average category; RCL, SIGLA, RPL, MCL and ANORAD are somewhat less programmable for various reasons; and PAL is perhaps the least desirable language, mostly because it is based on coordinate transforms and mathematical notation. These categorizations are quite general and can vary greatly. ANORAD, for example, is categorized as somewhat less programmable because of the coded nature of its instruction set. It does, however, require the least number of instructions (next to AUTOPASS) to code the program.

5. CONCLUSION AND DISCUSSION

From the above analysis and comparison, some general conclusions on the development of robot languages can be drawn. These are based on observations made within the report of the good and bad characteristics of the fourteen languages.

Because of the interactive nature of robot programming and the need for rapid debugging techniques, an interpreter-based language is favored over a compiler-based language. In order to alleviate the two major disadvantages associated with an interpreter (i.e., slow at run-time and less suitable for structured code) one can conceive

two compromises: (i) place some restrictions on program structure and commands provided in order to keep the interpretive nature of the language, and (ii) use an interpreter to produce correct programs and have a separate compiler which produces fast, efficient code from the same program to use at run-time.

RLs should be provided with as many of the debugging features shown in Table 4 as possible. Off-line debugging and development facilities, which use computer graphics techniques to simulate the robot and its environment, allow development to take place without the use of the robot[33,34]. An RL should be written in such a manner that it is easy to interface such a system.

In order to meet the needs of the growing robot industry, a robot language must be extensible. This implies a modular language structure with easy extension through user definable subroutines. Routines must be provided with parameter passing in order to allow re-use of the same variable name. The use of all global variables puts unrealistic restrictions on the user's choice of variables as the language grows. For a modular language structure which can build upon itself, subroutines must be allowed to be nested to as many levels as possible.

The RL should maintain structured programming techniques, but should present them in as simple a manner as possible. Although declaration of variables is important to readability, forced declaration of all variables may be cumbersome for simple program development. It may be desirable to have default values which indicate certain data types (as in FORTRAN) and also provide facilities to declare variables. The use of transforms in an RL is important because it allows definition of object locations instead of robot points. In order to make transforms as easy to use as possible, they should be broken into translations and rotations (as in MAPLE). A user with little training in transform arithmetic can still understand a change in position and/or rotation. Variable names should be unrestricted in length. There must be some cut-off, however, in the degree of recognition. In order to provide the user with maximum

flexibility, recognition should involve as many characters as feasible.

In order to make a language readable and understandable, "English-like" syntax should be used. Languages such as MAPLE, AL, and AUTOPASS which make attempts to supply commands which read very much like English are good examples of this. The major problem with providing "English-like" syntax is maintaining th same syntax through language extensions. Subroutines provide a means of extending the language, but the method used to invoke these routines will, in many cases, not conform to the language syntax. Perhaps some facility should be provided in the language which allows special extensions to be written by systems programmers which will provide a syntactically correct extension to the language.

A robot environment usually involves interaction of several devices. Some truly useful language features would be to allow commands to apply to any of the robots or peripheral devices in the environment and to allow concurrency of operation of the different devices. The simple ideas used in MCL, which allow the use of any device and provide facilities to transfer signals between devices, are general enough to apply to any situation. They would have to be expanded, however, to allow parallel operation of devices in the form of events with signal/wait primitives and parallel execution blocks.

The most important aspects in the development of a robot language are the use of a modular and expandable structure and the ability to interact with external devices and sensors. Without modularity and expandability, a language will not only be difficult to interface to different systems, but it will soon become outdated because of advances in technology. Without interaction with sensors and peripheral devices, the robot will be isolated from the world around it and will never be able to perform with any high degree of proficiency. These two aspects are the key to the success of any robot programming language.

ACKNOWLEDGMENT

This work was supported by Automation & Control Laboratory, GE Research and Development Center, Schenectady, New York. The authors wish to thank Joseph Gibbons, Lee Clark, and Stuart Miller at GE Research and Development Center for many useful, technical discussions.

6. REFERENCES

1. S. M. Mujtaba, "Current Status of the AL Manipulator Programming System," Proceedings of the Tenth International Symposium on Industrial Robots, Milan, Italy, pp. 118-127, 1980.
2. S.M. Mujtaba and R. Goldman, "The AL User's Manual," STAN-CS-79-718, Stanford University, January 1979.
3. Operation and Programming Manual, ANOMATIC CONTROLLER -- II, (CNC) Computer Numerical Controller, Anorad Corporation, Smithtown, N.Y., December 1979.
4. L.I. Lieberman and M. A. Wesley, "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly," IBM Journal of Research and Development, vol. 21, no. 4, July 1977, pp 321-333.
5. R. C. Evans, et al, "Software System for a Computer Controlled Manipulator," IBM Research Report RC 6210, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., September 1976.
6. D. D. Grossman, "Programming of a Computer Controlled Industrial Manipulator by Guiding through the Motions," IBM Research Report RC 6393. IBM T. J. Watson Research Center, Yorktown Heights, N.Y., March 1977.
7. "Automation Systems A12 Assembly Robot Operator's Manual," General Electric, P50VE025, Bridgeport, Conn., February 1982.
8. J.A. Darringer and M. W. Blasgen, "MAPLE: A High Level Language for Research in Mechanical Assembly," IBM Research Report RC 5606. IBM T. J. Watson Research Center, Yorktown Heights, N.Y., September 1975.
9. "Manufacturing Control Language User's Manual," McDonnell Douglas Corporation, January 1982.
10. A. Oldroyd, "MCL: An APT Approach to Robotic Manufacturing," SHARE 56, March 1981.
11. "MCL Language Definition-Version 1.0," McDonnel Douglas Corporation, November 1980.
12. E.E. Baumann, "Model Based Vision and the MCL Language ," Proceedings of IEEE SMC Conference, Atlanta, Ga., p. 433-438, October 1981.
13. K. Takasa, et.al., "A Structural Approach to Robot Programming and Teaching," IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-11, No. 4, pp. 274-289, April 1981.
14. K.G. Shin, G.P. Vukkadala, and N.D. McKay, "Development of an Explicit Robot Control Language, "Proceedings of COMPSAC 82, Chicago, IL., November 1982 (to appear).

15. C.Rosen. et. al., "Machine Intelligent Research applied to Industrial Automation, "Eighth Report, National Science Foundation, Washington, August 1978.

16. W.T. Park, "The SRI Robot Programming System (RPS): An Executive Summary, "SRI International, March 1981.

17. "Sigma Programming Handbook," Olivetti Sistemi per l'Automazione Industriale SpA, September 1977.

18. M. Salmon, "SIGLA-The Olivetti Sigma Robot Programming Language," Proceedings of the Eighth International Symposium on Industrial Robots, Struttgart, West Germany, pp. 358-363, 1978.

19. "Operation Manual for Cincinnati Milacron T3 Industrial Robot", Version 3.0, Robot Control with Restructured Software, Publication No. 1-IR-79149, Cincinnati, Ohio, 1980.

20. R.L. Tarvin, "Considerations for Off-Line Programming of a Heavy Duty Industrial Robot," Proceedings of the Tenth International Symposium on Industrial Robots, Milan, Italy, pp. 109-116, 1980.

21. C.S. Cunningham, "Robot Flexibility through Software," Proceedings of the Ninth International Symposium on Industrial Robots, Washington, D.C., pp. 297-307, 1979.

22. "User's Guide to VAL," Unimation Inc., Version 11, Second Edition, 1979.

23. "Unimate Puma Robot Manual," Unimation Inc., Text 398H1A, Vol. 1, April 1980.

24. R.. Paul, *Robot Manipulators: Mathematics, Programming and Control*, The MIT Press, Cambridge, Mass., 1981.

25. N.D. McKay and K.G. Shin, "A Microprocessor-Based Robot Control System with a Two-Level Hierarchy," Proceedings of the 20th International Symposium on Mini and Microcomputers and Their Applications, Cambridge, Mass., July 1982.

26. T.W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1975.

27. P.H. Winston, *Artificial Intelligence*, Addison-Wesley Publishing Company, Reading, Mass., 1977.

28. H.B. Kuntze, "Methods for Collision Avoidance in Computer Controlled Industrial Robots," Proceedings of the Twelfth International Symposium on Industrial Robots, Paris, France, pp. 519-530, 1982.

29. G. Gini, et. al., "Program Abstraction and Error Correction in Intelligent Robots," Proceedings of the Tenth International Symposium on Industrial Robots, Milan, Italy, pp. 101-108, 1980.

30. G. Gini and M. Gini, "Using a Task-description Language for Assembly. The Generation of World Models," Proceedings of the Eighth International Symposium on Industrial Robots, Struttgart, West Germany, pp. 364-372, 1978.

31. T. Hasegawa, "A New Approach to Teaching Object Description for a Manipulator Environments," Proceedings of the Twelfth International Symposium on Industrial Robots, Paris, France pp. 87-98, 1982.

32. J. Derby, "Kinematic, Elasto-dynamic, Analysis and Computer Graphics Simulation of General Purpose Robot Manipulators," Ph.D. Thesis, Rensselaer Polytechnic Institute, August 1981.

33. A.P. Ambler, "An Experiment in the Off-line Programming of Robots," Proceedings of the Twelfth International Symposium on Industrial Robots, Paris, France, pp. 491-504, 1982.

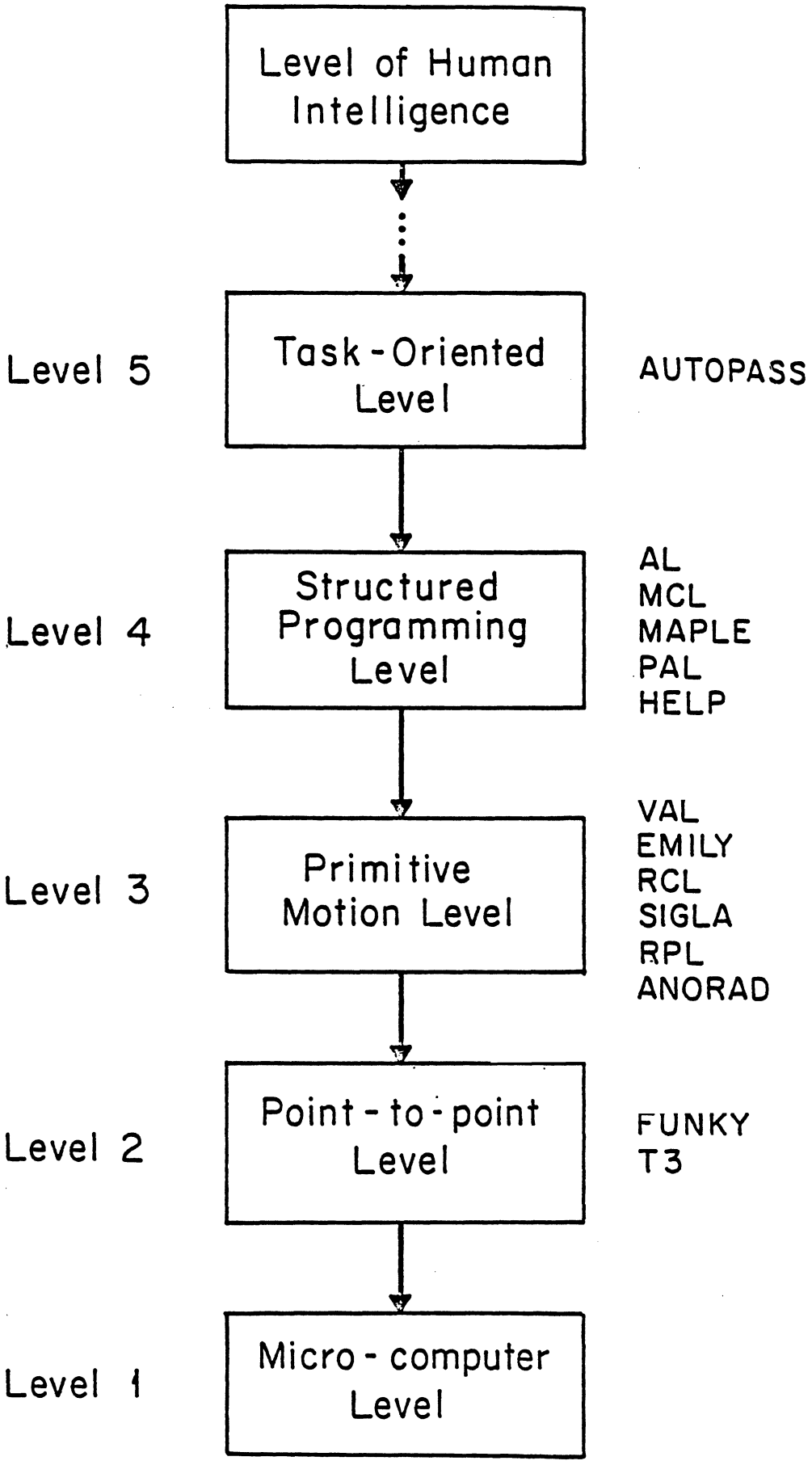

Figure 1. Classification of Robot Programming Languages

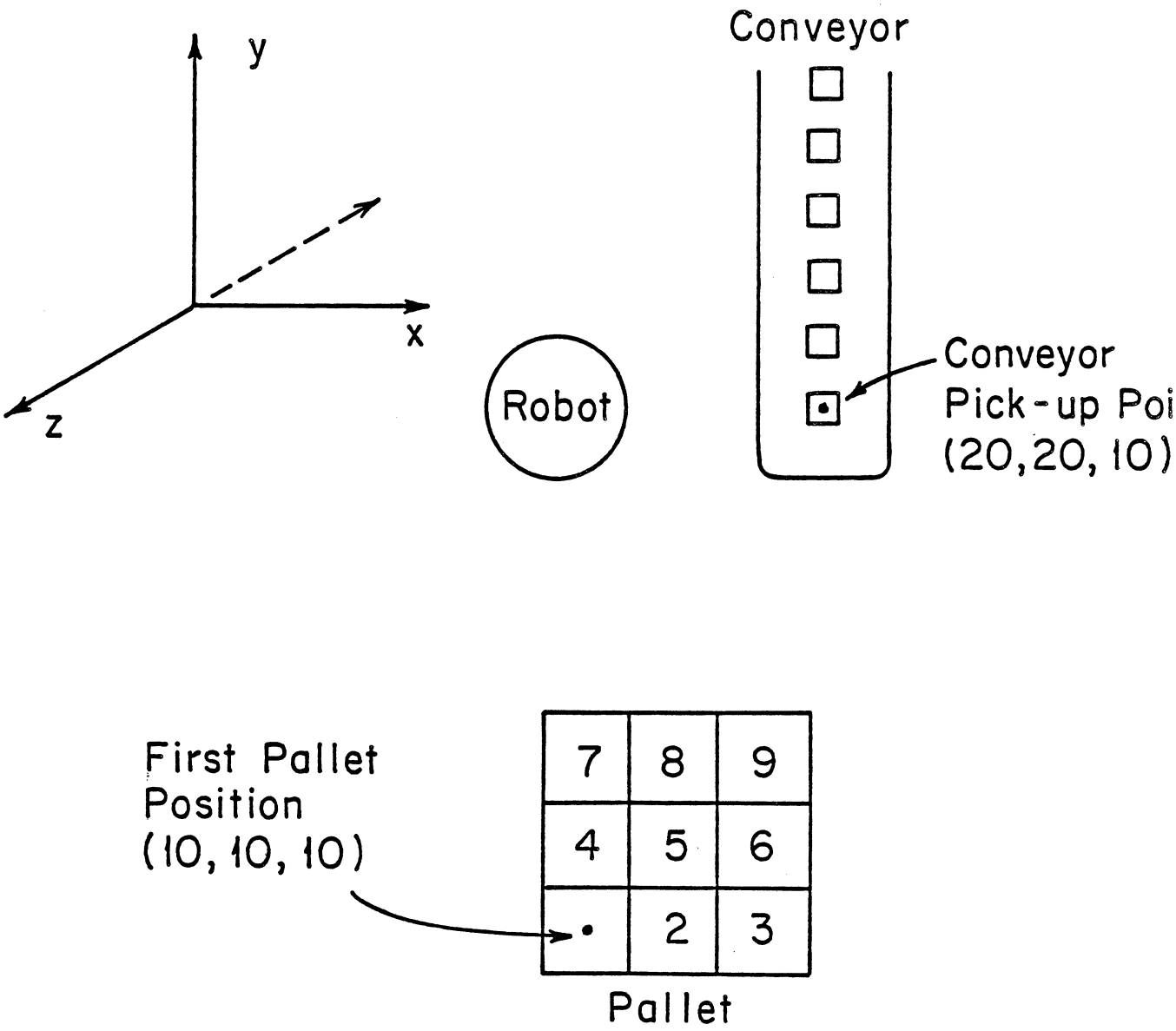

Figure 2. An Example of Palletizing Blocks

	T3	PUMA	Stanford	IBM arm	PACS arm	Allegro	Anomatic
Manufacturer	Cincinnati Milacron	Unimation	Sheinman	IBM	Bendix	General Electric	Anorad Corp.
Robot arm[1] Configuration	RRRRRR	RRRRRR	RRPRRR	RPPPRRR	RPPRRR	PPPRRR	PPPR
Degrees of Freedom	6	6	6	7	6	6	4
Maximum arm velocity	50 in/sec	3.3 fps	N/A[2]	30 in/sec	32 in/sec 120 deg/s	69 m/min	100 in/sec
Maximum weight load	100 lbs	5 lbs	2-3 lbs	5 lbs	25 lbs	14 lbs	45 lbs
Positional Accuracy	+/- .05"	+/- .004"	N/A	+/- .005"	+/- .010"	+/- .1mm	+/-.001 in
Maximum Reach	97 in	36.2 in	N/A	5 ft	vert. 84" hor. 49"	320x1000 x1050 mm	4 ft cube
Languages	T3	RPL VAL	AL PAL	MAPLE FUNKY EMILY AUTOPASS	RCL	HELP	ANORAD

1. R and P represent rotational and prismatic joints respectively.

2. N/A means "not available" to the authors at the time of this writing.

Table 2. Background Information on the Robot Programming Languages

	T3	FUNKY	VAL	EMILY	RCL
Robot Language Level	2	2	3 (4)	3 (4)	3
Origin	Cincinnati Milacron	IBM	Unimation	IBM	RPI
Computer Facilities	Controller based on AMD2900 bit-slice	IBM System/7	LSI 11/02	IBM 370/145 and System/7	PDP 11/03
Robot Arm	T3	IBM arm	PUMA	IBM arm	PACS arm
Number of Arms	1	1	1	2	1
Flexible to other arms?	no	no	no	yes	no
Changeable Tools	Can specify tool dimension	power screw-driver	none	none	none
Sensors Available	Limit switch in hand	Touch in Fingers	none	Touch in Fingers, Proximity, & presence	none
Vision Interaction	none	none	none	none	none

Table 2 (continued)

	SIGLA	RPL	AL	MCL	MAPLE
Robot Language Level	3	3 (4)	4	4	4
Origin	Olivetti	SRI International	Stanford	McDonnell Douglas	IBM
Computer Facilities	mini-computer	PDP-11/45 or LSI-11	PDP-11/45 PDP-KL10	Mainframe	IBM 370/145 and System/7
Robot Arm	SIGMA (1-4 arms) (3-8 DOF)	PUMA	Stanford	not specific	IBM
Number of Arms	1 to 4	1	2	more than one	1
Flexible to other arms?	yes	no	yes	yes	yes
Changeable Tools	yes	no	Electric screw-driver w/ tools	none	none
Sensors Available	Force and Torque feedback	tactile feedback & machine vision	force and torque feedback sensors	Simple and Complex (touch and vision)	force feedback, proximity & presence
Vision Interaction	none	orienting and placement	vision verification system	Modelling, Recognition, and Inspection	none

Table 2 (continued)

	PAL	AUTOPASS	HELP	ANORAD	
Robot Language Level	4	5	4 (3)	3	
Origin	Purdue	IBM	GE	Anorad	
Computer Facilities	PDP11/70	mainframe	PDP-11	Motorola 6800	
Robot Arm	Stanford	IBM	Allegro	Anomatic	
Number of Arms	1	more than one	1-4	1	
Flexible to other arms?	yes	yes	yes	no	
Changeable Tools	electric screw-driver	many tools indicated by instru-ction set	no	no	
Sensors Available	none	sensory feedback necessary	none	none	
Vision Interaction	none	Verifica-tion and vision modelling	none	none	

Table 3. Comparison of the Actual Features Available in the Robot Programming Languages

	T3	FUNKY	VAL	EMILY	RCL
Language Basis	Programmed Points	Programmed Points	Assembler	Assembler	Assembler
Language Type	Inter-preter	Compiler, Assembler, and Inter-preter	Inter-preter	Assembler and Inter-preter	Inter-preter
Control Structures	Conditional execution on signals	none	GOTO IF/THEN IFSIG/THEN	BRANCH LOOPFOR/ ENDLOOP BRCOND	GOTO BR** (** is condition)
Callable Routines	all sequences are routines	none	GOSUB RETURN (no P)	INCLUDE SUBR RETURN (P)	none
Nesting of routines	none	none	10 levels	4 levels	none
Data Types Available	real	integer, hidden vector and scalar	FRAME	INTEGER ARRAY	INTEGER POINT
Comments	none	none	REM	;	;
Simple Motion	Obtained by motion to points	Obtained by motion to points	GO, MOVE, MOVEI, APPRO	MOVE,MOTOR GOPOINT, joint motion	DRAW MOVE APPRO,DEPRT MOVEA
Straight Line Motion	Only when using Teach Pendant	Only when using Joystick	MOVES, APPROS	Cartesion robot provides st line	DRAWS MOVES APPROS DEPRTS
Continuous Path Motion	Yes	Yes, uses varying tolerances	Yes	none	none

Table 3 (continued)

	SIGLA	RPL	AL	MCL	MAPLE
Language Basis	Assembler	FORTRAN	ALGOL	APT	PL/1
Language Type	Interpreter	Compiler and Interpreter	Compiler and Interpreter	Compiler	Interpreter
Control Structures	IF JU	GOTO IF DO LOOPS	Begin/end, While/do, If/then/ else, etc	WHEN..ELSE ENDOF/WHEN WHILE..END OF/WHILE	if/then/else do/end,goto while/do/end begin/end
Callable Routines*	All files are callable routines (P)	CALL RTNSUB (P)	Procedures Functions Macros (P)	EXTENsion Invocation TASK,macro (P)	Procedures (P)
Nesting of routines	yes	yes	yes	9 levels	yes
Data Types Available	real	INTEGER REAL COMMON ARRAYS	SCALAR, VECTOR ROT,TRANS FRAME, etc	STRING REAL,ARRAY LOGIC FRAME	INTEGER REAL,ARRAY FRAME,POINT LINE,PLANE
Comments	none	COMMENT ;	{....}	$$	/*...*/
Simple Motion	MO	MOVETO	MOVE with constraint clauses	FROM, GOTO, WKPNT	MOVE TO/BY ROTATE TO ROTATE BY SWEEP TO/BY
Straight Line Motion	none	none	none	GOFWD GOLFT GORGT	Cartesian robot has straight line motion
Continuous Path Motion	none	"smooth path control"	Yes VIA gives intermediate points	GOFWD GOLFT GORGT	none

* (P) means parameter passing capability

	PAL	AUTOPASS	HELP	ANORAD	
Language Basis	Transform Basis	PL/1	PASCAL FORTRAN	NC programming	
Language Type	Inter-preter	Inter-preter	compiler and inter-preter	inter-preter	
Control Structures	for/to/do beg/end	If/then/ else,while /do,begin/ end, etc.	IF/THEN/ ELSE/END, FOR/TO/DO, WHILE/DO,etc	J(seq) J<cond>(s)	
Callable Routines	none	Procedures (P)	GOSUB RETURN	K(seq)	
Nesting of routines	none	yes	none	10 levels	
Data Types Available	Matrices Arrays Scalars Stacks	Integer Real Array world model	real, ASCII, arrays	real	
Comments	none	/*...*/	!	G69	
Simple Motion	mov	MOVE TO MOVE	MOVE	X,Y,Z,C	
Straight Line Motion	none	PUSH SLIDE	none	G1	
Continuous Path Motion	yes	implicit	SMOVE EOM	G9	

	T3	FUNKY	VAL	EMILY	RCL
Coordinate Transformation Commands	none	none	SHIFT, INVERSE, FRAME	PART defines part frame	none
Gripper Operation	use external signal commands	GRASP RELEASE	OPEN, OPENI, CLOSE, CLOSEI	HAND, DHAND	OPEN, CLOSE, FLIP
Parallel Processing	none	none	SIGNAL WAIT	SYNCH provides convergenc point	none
Interaction with external devices	WAIT OUTPUT EXTERNAL	none	IGNORE IFSIG REACT SIGNAL,etc	none	none
Force/Torque Feedback	none	A one dimensional SEARCH	CALIB	TOL, CALIB, GETPOS	STATSP
Touch Sensor Commands	SEARCH command for stacking	A command to CENTER the hand on an obj.	REACT, REACTI, SIGNAL, WAIT	WAND, SENSOR, GETSEN	SENSOR
Vision Commands	none	none	none	none	none
Tool Operation Commands	TOOL STATUS and exter-nal signal commans	TOOL operates screw driver	TOOL defines tool frame	none	none
Higher Level Commands	none	none	none	none	none

	SIGLA	RPL	AL	MCL	MAPLE
Coordinate Transformation Commands	none	TMULT TINVER TWRITE TREAD	*, -->, AFFIX UNFIX	USEFRM CONECT DISCON	TRANSLATED ROTATED FIXED RELATIVE
Gripper Operation	AX (external signals)	RELAYO RELAYC	OPEN CLOSE	SEND/HAND, OPEN SEND/HAND, CLOSE	OPEN TO OPEN BY
Parallel Processing	AU (// exe) EW,ES wait and signal events	none	Cobegin/ coend SIGNAL and WAIT	INPAR runs more than one TASK	IN PARALLEL WHEN event BEGIN..END
Interaction with external devices	AX,PP (I/O signals)	DTLRD,DTLWR SETOX,CLROX RDDXV,REL-AYØ,RELAYC	SIGNAL WAIT	SEND RECEIV DEVICE	STATUS
Force/Torque Feedback	RP (exert) MT (read)	INIARM	FORCE TORQUE	SEND and RECIEV	TORQUE, FLEFTFINGER FRIGHTFINGER FGAP
Touch Sensor Commands	TS and PP (test and set)	DTLRD DTLWR DTLINI	CENTER	SEND and RECEIVE	HIT RANGE
Vision Commands	none	INIVIS,PICTUR.GETFEABLINK,DELBLO,RECOGN	none	REGION, PROJEC, LOCATE, INSPEC	none
Tool Operation Commands	external signal commands	none	OPERATE with constraint clauses	TLAXIS defines tool frame	none
Higher Level Commands	IV,AV rel and abs motion; AC anti-col	RECOGN identifies an object	Compliant motion if FORCE set to zero	REGION defines part dimensions	LIMITS SETLIMITS

Table 3 (continued)

	PAL	AUTOPASS	HELP	ANORAD	
Coordinate Transformation Commands	+ (mult) - (invert and mult)	none	none	none	
Gripper Operation	gra rel	GRASP RELEASE	PULSE VALUE	M100-M195	
Parallel Processing	none	IN PARALLEL DO	SIGNAL WAIT ACTIVE TEST	none	
Interaction with external devices	none	SWITCH LOAD UNLOAD FETCH	SET RESET PULSE STROKE	M100-M195 M200-M223	
Force/Torque Feedback	compliance by equating axis align-ments	GRASP, INSERT, PUSH	A<axis> FORCE	none	
Touch Sensor Commands	none	INSERT, EXTRACT, LOWER/ONTO LOWER	IF TESTB TEST HIGH	M200-M223	
Vision Commands	none	VERIFY	none	none	
Tool Operation Commands	scr	OPERATE, FETCH, REPLACE, SWITCH	none	none	
Higher Level Commands	none	PLACE/ON. DRIVE/IN. NAME/ASS-EMBLY, etc	none	G2, G3 circular interpola-tion	

Table 4. Debugging Facilities Available for Each of the Robot Programming Languages

	T3	FUNKY	VAL	EMILY	RCL
Language Type	Interpreter	Compiler, Assembler, and Interpreter	Interpreter	Assembler and Interpreter	Interpreter
Manual Teach Mode	Yes with external functions	Yes with external functions	Yes using HERE or T command	Yes using JOY command	Yes using MOV and ! or HERE
Trace Feature	Gives current robot location	Yes in PLAY and LIST mode	none	Yes using TRACE command	none
Single step execution	Yes using function buttons	Yes	Yes using NEXT command	Yes using TRY command	none
Hot Editing	Yes can insert and delete steps	Yes by using ERASE and RECORD	none	ML steps may be inserted or deleted	Yes, in interactiv mode and error mcde
Halt Execution	Yes using function buttons	Yes using HALT command	Yes using ABORT command	Yes by pushing panic button	Yes using ABORT command
Back Stepping	Yes using function buttons	Yes using REVERSE mode	none	none	none
Breakpoints	Yes using WAIT command	none	Yes using WAIT command	Yes using TRY or partial execution	Yes using BREAK command
Immediate Execution	Yes using Teach Pendant	Yes using Joystick	Yes using DO command	ML commands can be issued	Yes, use IMM command in monitor
Partial Execution	Yes using function buttons	Yes Modes work like tape recorder	Yes can give starting step	Yes can give start address	none
High Level Facilities	none	none	none	none	none

Table 4 (continued)

	SIGLA	RPL	AL	MCL	MAPLE
Language Type	Interpreter	Compiler and Interpreter	Compiler and Interpreter	Compiler	Interpreter
Manual Teach Mode	Yes using TE/file command	Yes using JOYON an JOYOFF	joystick	Proposed off-line graphics teaching	joystick
Trace Feature	none	Yes, trace occurs automatically	none	none	PTRACE
Single step execution	Yes using SC/file command	yes	none	none	none
Hot Editing	none	Can modify octal contents of memory	Can modify p-code	none	none
Halt Execution	Yes using HL command	Yes from keyboard	Yes using STOP or ABORT	Yes using ABORT command	QUIT
Back Stepping	none	none	none	none	none
Breakpoints	none	Yes programmed and user settable	Yes using STOP in program	none	none
Immediate Execution	yes	none	none	none	none
Partial Execution	none	none	none	none	none
High Level Facilities	none	none	examine and set var's, signal and wait events	none	memory DUMP

Table 4 (continued)

	PAL	AUTOPASS	HELP	ANORAD	
Language Type	Interpreter	Interpreter	Compiler Interpreter	Interpreter	
Manual Teach Mode	read current position planned	none	yes using Teach button	none	
Trace Feature	Automatic trace in teach program	Interactive verification of int. code	ECHO	done automatically	
Single step execution	none	Automatic step thru interpretable code	STEP	STEP button	
Hot Editing	Interactive teaching program	Can modify code in interpreting phase	none	none	
Halt Execution	none	halt available	hold and EMERGENCY buttons	RSTRT button	
Back Stepping	none	Can Backup at any time to change int. code	none	none	
Breakpoints	none	none	ASK commands	none	
Immediate Execution	none	none	none	MDI mode	
Partial Execution	none	none	none	none	
High Level Facilities	teach program	compiler can ask user for information	memory map	none	

Table 5. Quantitative Comparison between the Languages for the Programming Example of Palletizing Blocks

	T3	FUNKY	VAL	EMILY	RCL
Number of Instructions	94	94	35 (22)	34 (27)	76
Development Time	1,5	1,5	2	2	2,5
Understandability of Instructions	2,6	2	3,4,5	3,4	3,4,5
Structured Format	3	3	3	1	2
Flexibility of variables	4	5	2	2	2
Ease of Extension	4	4	4	1	4
Range of Users	1-2	1-2	2-3	2-3	3
Programming Complex tasks	4	4	4	4	4
Computing Power	1	1	1	2	1
Sensing Ability	1	2	1	2	1
Availability	1	2	1	2	2

Table 5 (continued)

	SIGLA	RPL	AL	MCL	MAPLE
Number of Instructions	35	58	50	69	44 (35)
Development Time	4	5,6	3	6	3
Understandability of Instructions	6,7	3,4,5	1,4	3,4,5	1,4
Structured Format	3	3	1	2	1
Flexibility of variables	4	2	1	2	1
Ease of Extension	4	1	2	3	2
Range of Users	2	4	4	4	3
Programming Complex tasks	4	2	3 (1)	2	3
Computing Power	1	2	2	3	3
Sensing Ability	1	3	1	3	2
Availability	1	2	2	1	3

Table 5 (continued)

	PAL	AUTOPASS	HELP	ANORAD	
Number of Instructions	25	8	30	20	
Development Time	7	1	2	4	
Understandability of Instructions	3,6	1,4	3,4	6,7	
Structured Format	3	2	2	3	
Flexibility of variables	3	1	1	2	
Ease of Extension	4	2	3	4	
Range of Users	5	1	2-3-4	2	
Programming Complex tasks	4	1	4	4	
Computing Power	2	3	2	1	
Sensing Ability	1	4	1	1	
Availability	2	3	1	1	