N90-29810

Thread: A Programming Environment For Interactive Planning-level Robotics Applications

John J. Beahan, Jr.[1]

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109

Abstract

This paper discusses the Thread programming language, which was developed to meet the needs of researchers in developing robotics applications that perform such tasks as grasp and trajectory design and analyzing sensor data, and which interface with external subsystems in order to perform servo-level control of manipulators and real-time sensing. This paper discusses the philosophy behind Thread, the issues which entered into its design, and covers the features of the language from the viewwpoint of both researchers who want to develop algorithms in a simulation environment, and those who want to implement physical robotics systems. This paper does not attempt to explain the detailed functions of the many complex robotics algorithms and tools which are part of the language, but only to give an impression of their overall capability.

I. Introduction

This paper mixes many concepts from robotics, programming language design, large-scale software engineering and systems theory. Because of the unusual set of issues addressed, and the fact that some of them may be rather esoteric to some members of the robotics community, this paper mixes abstractions and "nuts and bolts" examples rather freely, with little middle ground, which will hopefully not prove too disjointed.

The paper will first explain the philosophy behind this work, then cover a bit of the background, rationale and goals it tries to address. The Thread language is then described, first from the viewpoint of software engineering, then from the viewpoint of the specific robotics capabilities it provides. The software features are covered first because they contribute greatly to the utility of the language in developing algorithms and performing experimental robotics, and have a significant impact in how the customized robotics facilities can be used. The language's individual features will first be discussed in an abstract manner, then with specific examples, from the viewpoint of both robotics and software engineering. A concluding section will discuss the lessons, both positive and negative, which were learned during the development and use of the language.

Thread is an interpreted (threaded) language, implemented in Ada, which provides an environment for interactive program development using a wide variety of language features specialized for robotics. The language was developed to meet the needs of the Run Time Control (RTC) subsystem of the JPL Telerobot Demonstrator, which is aimed at performing flexible satellite servicing in space using a cooperative man-machine system partitioned into a base site (ground or space station/shuttle) and a remote site. The Telerobot system is organized hierarchically by bandwidth of contact with the physical manipulator hardware, with an interactive Operator Control Station connected to a Task Planning and Reasoning subsystem, which then sends the RTC goals and constraints expressed as English-like commands such as "`move the right arm to a location 10.0 cm above the Door_handle top using elbow_up poses please.`". The RTC performs the detailed numerical planning steps and sensor monitoring operations necessary to execute the command and recover from simple anomalous events, calling on a manipulation subsystem and a machine

[1]This work was performed with the generous support of Guillermo Rodriguez, Supervisor for the JPL Tele-Autonomous System Research Group.

vision subsystem for functions such as joint and cartesian motion, compliant hinge rotation, intelligent "macros" such as bolt threading and 3-d real-time object position determination. The Telerobot system is very large, consisting of a network of five primary computers and roughly 200,000 lines of embedded software, which use a highly diverse mix of operating systems, programming languages and implementation philosophies.

Philosophy

The philosophical starting points for the development of Thread were ideas of object-oriented design from computer science, integration and robustness issues from software engineering, coordination and dependency concepts from systems theory, and abstraction principles from programming language theory.

The number of additional issues which computer implementation adds to the problems of robotics is such that it is not unusual for the implementation details of hardware/software development to dominate all other concerns in time and effort. The magnitude of the software overheads can be significantly lowered, however, by using advanced software engineering methods, and techniques from some of the less well-known branches of computer science. Thread is an attempt to reduce some of the many overheads associated with implementing robotics concepts in software, and this paper tries to convey the philosophy and methods which went into the design of Thread. The intention of this paper is not merely to recommend Thread as an environment in which to build general robotic systems, although it is a very powerful tool for the class of problems it was designed to address, but rather to explain the ideas behind Thread so that they may be applied to implementing other robotic systems in whatever particular context is of interest.

The system which Thread was developed to serve, the RTC, was a large project involving many developers, and this led to the fact that the large-scale system issues and software engineering concerns were likely to dominate development unless significant investments were made to manage them. A strong commitment was made from the outset to accept the additional initial investment necessary to insure proper flexibility and generality in the future. Also, the needs of the RTC necessitated that the language be extensible at the compiled level, extremely interactive, and be able to be made extremely robust to runtime errors. For these reasons, Ada was chosen as the implementation language and the decision was made to use Thread as the RTC development environment. In addition, the observations made during many large system development efforts [3] to the effect that peripheral concerns with implementation details can make up the majority of the development effort, sometimes over 90%, led to the need to reduce from the outset the number of minor software implementation details the RTC developers must deal with, and allow them to work in close proximity to the actual issues of robotics, rather than being distracted by software issues.

There were also several other issues associated with the development of large systems which were prominent in the original conception of the Thread language. A well-known result from software engineering studies is that productivity, in terms of quantity of software per unit time, is relatively independent of the level of abstraction in the language used, i.e., writing in a low-level language with simple instructions is neither significantly faster nor slower than writing in a highly abstract one with very powerful individual instructions. Therefore, unless computational efficiency prevents it, one should attempt to use as abstract a language as possible, to achieve maximum functionality with as little software as possible.

Another well-known aphorism of software engineering is that "the production of a large system that works is practical only by extension from a small system that works", by building in layers upon a firm foundation. It is utterly impractical to build a deep multilayered system if the lower levels are not initially built to be very general and robust so that the later work can be productive, rather than being largely devoted to revisiting and redressing the design of the lower levels [3]. When an implementation is built to solve a problem, it should solve that problem once and for all, and be able to be reused *with no significant change or effort* in whatever context that problem arises. Implementations which do not generalize are a complete waste of time for large systems, which are long-term and diverse enough that specialized implementations will end up being rebuilt several times in slightly different ways, at roughly the same cost each time.

For similar reasons, it is also unacceptable that the products of several separate research and development efforts should require more than moderate effort to be integrated. Much of the implementation of robotics research systems is carried out in a manner which isolates the results of different researchers from each other, so that even though one research group may have a very powerful manipulator control system, capable of sophisticated force control, and another may have a very flexible kinematics/dynamics simulation environment and graphics interface, integrating these two at anything more than a superficial level usually requires that the two software (and sometimes the hardware) systems be extensively modified, if not virtually rebuilt from scratch. Although this situation is thought to be a fact of life by many people, methods do in fact exist to guide the implementation of systems so that they are very robust to shifting requirements and much more able to accommodate changes. This not only allows separately developed software to be more easily integrated, but more importantly, it leads to the development of software which is far more generally applicable, and thus allows the development effort to easily and safely build progressively higher on previous work.

Related Work

There have been a number of languages specialized for robotics [9,18,10], but virtually all of them have been intended for use as standalone systems, not as components in a very large integrated system. For that reason, most of the requirements placed on Thread were not met, and were often not desirable for previously existing robot languages. Many were overlayed on existing low-level languages [10,18] and thus, although they do provide specialized facilities for robotics, do not relieve the user of any of the minutiae of software development. Also, systems have traditionally been much smaller, and it has been only recently that a robot system could afford to rely on separate subsystems to perform servo control and real-time sensing and concentrate on the planning-level issues which arise when flexibility becomes the primary goal; previous robot environments have been devoted primarily to the performance of very rigid tasks in an extremely structured environment.

The characteristics of the Thread language actually place it closer to one of the modern very-high-level specialized languages such as *Mathematica*TM, than to any traditional robot control language, as it has a much stronger interactive orientation than do traditional automated robotics systems.

Rationale and Goals

The rationale behind the design of Thread involved a number of factors, the first of which is the incremental cycle of design and implementation inherent in a research environment, which requires from the outset that the system be able to cope easily with significant design changes which occur as experience and understanding is gained from initial prototypes. This leads to a heavy emphasis on flexibility rather than performance, reinforced by the fact that the system is intended as a rapid-prototyping research tool rather than an application system. Also, there are typically many more developers and users of software at the planning level than at the servo level of robotics, and this also leads to a decreased emphasis on optimally efficient implementation and more concern with issues of flexibility, reliability and abstraction than has historically been present in most robotics implementations. The large size of the Telerobot led to the RTC development team numbering five people, which created much stronger requirements for formal software development discipline than is typical in a research project. The tight integration between the various subsystems in the Telerobot meant that faults and liabilities in one subsystem could quickly collapse the system as a whole if they were not carefully controlled, putting heavy burdens of reliability on the software. The fact that software behavior was strongly dependent on physical interactions in the environment led to the need for a completely interactive development environment which could be easily and safely used to perform software development during laboratory experimentation with actual manipulators. The strongest contribution to the overall design requirements of the language, however, was the desire to support an extensible "tool-oriented" development environment, which would give the developer *and the user* the freedom to use the system either as a single program, providing complete robotic tasks as inputs, or in a very piecemeal way, using the various functional modules of the system individually, to control the precise details of a portion of a task.

Tool-oriented Philosophy

The desire is for the Telerobot to be usable at either a very high level, or to have the operator intervene and aid in some operations, at whatever level of detail is desired or appropriate. The implementation structure chosen for the Run Time Controller is a layered set of mutually supportive tools, with the highest level made up of a few large, separate modules, used as utilities with conventional menu/graphics interfaces, for planning obstacle-free paths, generating trajectories, simulating grasp effectiveness, modifying the world model, executing actuation primitives which physically move arms, accessing the vision system, etc. These large modules are built up from toolkits designed for building path planners, trajectory generators, etc., *each with its own appropriate graphical/ textual interfaces.* These tools are built out of smaller, simpler tools, corresponding roughly to conventional software utility libraries, but *usable in an interactive manner.* The lowest level is simple numeric operations, control structures, etc. plus customized robotics features, *all interactively available at any time.*

The benefits of this method are myriad, including: (i) as mentioned above, the operator naturally has visibility and access into all points in the structure of the system at all times; (ii) the design of a tool-oriented implementation will probably be far superior to a monolithic design in generality and robustness, since far more thought and effort will have gone into organizing its subtleties and interrelationships into a coherent structure; (iii) because tools are far more easily reusable than specific programs, it allows construction of later work by relying heavily upon what has already been built, in a much more efficient way than more specialized development styles, which often repeatedly replicate effort.

The drawbacks of this implementation style lie mainly in the complexity of the resulting interfaces, which typically are inefficient at hiding detail from the user; overlaying the textual interface with a graphical/ iconic one is quite feasible, and may go a long way toward alleviating this problem.

II. The Thread Language

Overview

Thread's set of syntactical and semantic features are somewhat unusual, but most of its characteristics are found spread among several other existing languages. Thread allows the use of several different formalisms of programming, including applicative and object-oriented programming along with the conventional style, and the language supports the standard complement of basic data types, operators, control structures, file access facilities, access to the operating system, and various utilities such as a help system and a dictionary of user-defined objects and procedures. The language's syntax superficially resembles Forth [4], but the semantics are closer to that of very abstract languages such as Smalltalk [7,8] or LISP [19]. Thread's design principles are an attempt to apply very abstract programming language concepts to make *building tools* as convenient as possible, rather than focusing only on simplifying the task of writing applications programs.

The most unusual aspect of the language is that it has no input primitives of any kind: the only method of input into a program is the interpreter itself, used recursively. Because of the simple syntax of the language and the interpreted nature of its implementation, any code fragment in any form behaves as an entire program, able to be compiled to the internal form used by the interpreter and executed. There is thus no distinction made betweeen data and code, which naturally gives the ability to treat user input, or the contents of a file or text string, as if it were an executable function. This is a very powerful feature, which can be used in a variety of useful, if slightly unorthodox ways, as will be discussed later.

The fundamental design concept of the language was extensibility, to allow the language to be extended with new routines by entering interpreted code, and also to allow the modular addition of new primitive functions, in the form of compiled code, into the language interpreter kernel itself. An extremely simple interface exists between the **Thread** language and several compiled languages (Ada, C, Fortran, Pascal) which allows the user to use existing high-speed software developed in conventional environments to add new primitive operators to the language, usually at a cost of only a few lines of interfacing "wrapper". The dual compiled/interpreted nature of the language allows the use of compiled code for high-speed or well-established functions, and interpreted code for rapid prototyping and fluctuating components which are used directly by the human operator. Also, because of the extreme modularity of the implementation of the language kernel, functions are easily extensible and shiftable between compiled and interpreted levels with little overhead.

Syntax and Semantics

The syntax is a simple postfix notation, read from left to right, with each data item implicitly pushed to a global stack and each procedure executed by taking its arguments from and leaving its results on the stack. This has the effect of making every single function and subroutine automatically into a facility which can be directly accessed from the keyboard, which is a strong encouragement to construct programs so that they can more conveniently and modularly be tested and debugged, thereby encouraging generalized tool building rather than specialization. No distinction is made by the language between language primitives and user-defined procedures; this encourages the development of small, modular, carefully tested units of software which can be shared among many developers with a minimum of overhead. (Typically, the only documentation needed for the new user of the language is a short, hands-on tutorial and a list of the names and argument formats of the available primitives and user-defined procedures.) This stack-oriented notation is prevented from becoming cumbersome because procedures possess local stack contexts, use named arguments and named local variables to store data. This removes the heavy burden of stack manipulation which is common to other stack-oriented languages.

The following figure gives a very quick overview of the basic syntax and semantics of **Thread**, which is provided as an introduction to the many short examples given later. (The convention for naming objects in **Thread** is that data items begin with an uppercase letter, and procedures are lowercase.)

```
"Hello, world." print.                        A program which prints a string.

2 3 plus 4 times print.                       Calculates the number 20.

3.2 make A_variable.                          Create a variable.

"Something else." store A_variable.           Store something else into it.

1 define factorial                            Defining a simple function.
        N
    if N 1 equal then
        1 return
    else
        N 1 minus factorial N times return
    endif.

11 factorial print.                           Use the function.
```

The postfix syntax also allows the use of applicative-style programming techniques [17,2], which allow the construction of entire programs by successively applying operators to data items, and other operators, without the usual overhead of declaring specific intermediate data structures to transfer results between routines.

Features

The language also supports access to the interpreter itself as a function, so that code can be constructed using string manipulation functions and then compiled or executed as desired. This supports both static (at compile time) and dynamic (at execution time) binding of procedures and data objects, as the user desires, which supports implementation of higher-order and abstract functions in a very natural way. Using these facilities with Thread's string manipulation capabilities, it is very easy to build very simple utilities which will manipulate fragments of programs to perform very powerful operations, such as searching sets of previously defined source code for the presence of a certain variable. The capability to define abstract functions also enables the use of object-oriented programming techniques [7], a method of maintaining extremely rigid boundaries between software components so that internal modifications and implementation details do not propagate throughout the program, but are localized for detection and correction.

Here are trivial examples of using the interpreter as a function:

`"2 2 plus print." listen_to_string.`	Call Thread to execute as string.

`"define double 2 times print." listen_to_string.`
`2 double.`
	Compile a definition in a string, then use it.

`Door 30.0 dg Param_value setvalue.`
`Door Param_value getvalue print.`
	Store and retrieve the value of the door hinge angle from the world model.

Here are examples of the use of abstract procedures which take code fragments as input parameters and thus can be implemented to function independently of any particular data type or specific application:

`List_of_drivers_licences "has_fewer_tickets" shell_sort.`
 	
`List_of_names "alphabetically_earlier" shell_sort.`
	This shell sort algorithm takes as inputs a list of arbitrary data objects and a code fragment containing an inequality relation to use during comparisons.

This same facility allows the creation of "generic" algorithms, which are only partially determined and can be instantiated to perform many different functions:

`Task_board_frame Subtree_list getvalue`	Fetch the list of primitive objects making up the task board assembly.
`"Name_string getvalue print cr"`	Create a piece of code which will display the name of a database object.
`apply_to_each_element.`	Apply the code to each element of the list.

`List_of_subtrees "length_of"`
`apply_to_each_element`
	Use a similar method to find the number of leaves in each subtree.

`"plus" reduce_list store Number_of_leaves.`	Add all the individual sums together into a single count.

By applying this technique to individual data objects, they can contain their own local procedures to control how they need to perform their own operations. This allows (i) the construction of tools which are very easily reconfigurable to suit new contexts, because their functions are largely determined by input data rather than built-in coding, and also allows (ii) modular segregation of *portions of algorithms* which are specific to certain data types (or situations) to be stored with those objects (or associated with that situation). An example is the use of an object-oriented database to store algorithmic information about how to compute grasp points individually for each object:

```
Door_handle1 Hinged_grasp Howto_grasp setvalue.
Panel_handle Hinged_grasp Howto_grasp setvalue.
Crank_handle1 Rotary_grasp Howto_grasp setvalue.
```

Store the appropriate hinged or rotary grasp *algorithm* (or any other code desired) with each object, as is appropriate.

```
define grasp_object
    ...
    Grasped_object Howto_grasp getvalue thread
    ...
```

To use the algorithm, retrieve it from the object model and execute it.

The language is not strongly typed, in the conventional sense, because although it performs rigid type checking, it does so entirely at runtime, which allows all variables and arguments to be generic, able to hold data of any form. This removes most of the overhead of declarations, type conversion, etc. from the user, at the cost of some computational inefficiency. The prime benefit of runtime typing, however, is that all the details of memory management and dynamic allocation are handled transparently by the language, so that objects are automatically created, copied, and destroyed whenever they should be, removing a great deal of simple, repetitive detail from the developer, at the cost of an efficiency loss. Runtime typing is also an important feature of the language because of the lack of any pointers, or reference types, which are one of the most common sources of software errors.

Another property which was designed into the language is that all data structures have direct syntactic representations, and can thus be entered at the keyboard through the interpreter at any time. This means that users never need to write special formatting routines to read or write their own data (which are built using the Thread **aggregate** type), because the language itself supports direct entry of arbitrary data structures. The language also has the capability to "decompile" either code or data back into source text which can be modified and recompiled, which will evaluate to the value of the data structure or code fragment.

An interface between the language and the host operating system is supported, which allows the user either to "shell out" to the operating system interactively and then return to the language, or to call the operating system as a subroutine to perform complex file- and program-manipulation operations. The interactive help system and many other utilities for common use were built using these facilities.

One of the prime features of the language is the extremely extensive error diagnostics, which were necessary to complement the runtime type checking, since it would otherwise be very difficult to find the location and cause of errors. This feature, combined with the decompiler and operating system interface facilities, makes it easy to obtain very rapid turnaround time during debugging.

Thread is implemented in Ada, C, Fortran and Thread itself, with its utilities and development tools implemented partially in various other interpreted languages (VMS command language, RUNOFF text processor scripts, KERMIT file transfer protocol, etc.). The capability to build this rich set of utilities comes from the fact that the other software products involved are driven by interpreted languages, which can be generated in a structured way by Thread programs and executed to perform useful work. This philosophy, of interfacing two functional modules by use of a language, rather than data structures, represents a very powerful design and implementation technique.

Using recursive calls of the interpreter itself as an input method allows the use of any available facilities, either built-in or extensions, whenever user input is being accepted or requested. There are no mode restrictions, in that any and all elements of the language are immediately available at all times, rather than being segregated into different artificial partitions which are mutually exclusive. At any time the interpreter is active, either in normal command mode or when the user has been queried to enter some information, the user can: (i) enter code at the keyboard as response to queries; (ii) access the operating system, editors, etc.; (iii) use existing utilities to generate requested inputs; (iv) override current activities; (v) access facilities at high or very low level. Typically, all geometric computations are done using the world model facilities in a convenient, interactive manner, rather than by building any specialized user-friendly interfaces.

Because of the fact that Thread is an interpreted, extensible language with a rich variety of primitive capabilities, it is very easy to use it to construct highly heterogeneous utilities which use preexisting software written in other languages. Many of the development utilities of **Thread** are written in other languages, and the utilities operate by calling the other languages as "subroutines", through the operating system, to perform their operations. In addition, whenever a product becomes available which is useful for robotics and which accepts input in some form of command language, it is usually quite straightforward to build a simple interface wrapper to it using Thread's abilities to call itself and the operating system as functions. A hidden-surface graphics display primitive was developed using this technique to transfer Thread graphics model structures into *Mathematica*™. This ability to so quickly and easily interface with completely separate pieces of software is probably among the most powerful features of **Thread**, derived in this instance from the fact that both **Thread** and *Mathematica*™ are fully functional languages, instead of just being specialized interactive programs to perform specific functions.

III. Robotics-Specific Features

Contributions

Because the **Thread** language is extensible, it has been worked on in many different ways by nearly a dozen people, many of whom contributed algorithms and suggestions about how to integrate different facilities into the language. This author developed the language concept, implemented the interpreter kernel, the development utilities, the core of the database/world model and, later, several of the high-level features such as the pseudo-English command language and the obstacle-avoiding trajectory planner. Many other individuals in the Tele-Autonomous Systems Research Group were responsible for the majority of the robotics-specific algorithms and features of the language[2].

Overview

The standard objects and operations of robotics are supported as elemental data types and primitive operations within the language. The following data types exist: homogeneous coordinate transformations [12], rotation matrices, translation vectors, N-vectors, M*N matrices, kinematics poses, sets of joint angles, solid modelling primitives (both CSG and convex polyhedra) and volume/surface/edge decompositions, graphics models, and the **Thread aggregate** data type, which can hold any collection of data objects. The basic operators available include: the standard linear algebra operators for vectors and matrices, the Singular Value Decomposition, forward/inverse kinematics based on analytical solution for the PUMA 560 manipulator (currently being generalized to recursive methods for redundant and multiple arms [14]), checking of joint stops/singularities/degeneracies, interpolation of manipulator trajectories in joint space and task space [16] (including a modification of the Taylor algorithm, developed by H. Stone, which allows pose flips to take place smoothly during joint-approximated task space motions).

Features

The language provides an active world model/database which can model both the robots and their environment using object-oriented methods for algorithmic and data abstraction, and in which commonly used functions are handled transparently by the database itself: coordinate transformations in 4 different frames associated with each object, 3-D perspective graphics (wireframe without hidden line), collision detection, using unions of both CSG [13] and of convex polyhedra, nearest-distance metrics of computational geometry [6,5], parameterized geometric relationships (hinges, etc.), general point-to-point jacobian algorithm based on Recursive Spatial Algebra [14], (in progress: consistent update using *a priori* and sensor data fusion heuristics).

Using the world model and the basic kinematics facilities, a set of motion simulators has been built to allow the user to predict the effect of singularities, jacobian manipulability measures, etc. on the outcome

[2](in alphabetical order) J. Balaram, A. Jain, K. Kreutz, B. Lau, A. Lokshin, B. Mueller, R. Sidney, H. Stone

of a motion:

```
Bolt2 Abslocal_trnsf getvalue

10.0 cm up

RUN Task_interpolation Move_Absolute.
```

Get the coordinates of the bolt from the world model.
Offset them by an appropriate amount to hover above the bolt.
Use Right-arm elbow-Up Negative-wrist kinematics pose and cartesian interpolation mode.
Simulate a motion to that point.

A basic graphics capability is available for animation, using 2-D plots and 3-D wireframe models without hidden line removal, but with a very user-friendly interface for controlling viewing conditions of the world model. (More sophisticated hidden-line graphics, both animated and hardcopy, are available through interfaces to other products, such as *Mathematica*™ and the IRIS 4D-70 GT graphics engine.)

For purposes of high level planning, simulators which predict the kinematical behavior of the robots for several specific actions have been developed, including grasping of an object, swinging a door, or contacting a surface while performing guarded motion.

```
Task_board 10.0 cm Z_direction move_contact.
```

Move 10 cm in the Z direction, expecting to contact the task board.

```
Crank_bar 30.0 dg 3.0 dg swing_object.
```

Rotate the crank bar by 30 degrees at a rate of 3 degrees per second.

One of the high-level utilities built into the language is a simple heuristic search path planner, which will find obstacle-free paths which also satisfy all other kinematic restrictions. The planner usually finds a path within one minute for an uncluttered environment which requires only one or two straight line (joint or cartesian) segments to negotiate. The path planner functions using a heuristic combinatoric search through choice-sets of positions, kinematics poses, and joint and cartesian interpolation modes [16], calling on whatever objects are in its copy of the world model to perform kinematic constraint checking. This very naturally allows the path planner to be used for a variety of purposes in addition to free-motion planning, such as choosing a grasp point on a handle which will permit the successful opening of a door, automatically taking into account the additional motion of the door for the simple reason that the world model registers it as being attached to the hand.

```
Crank_handle1 Abslocal_trnsf getvalue
15.0 cm up
Left_elbow_up_poses
build_choice_set
plan_a_path
move_the_arm.
```

Fetch the coordinates of the crank handle, offset them to the desired position near the handle, choose which kinematics poses of the arm to use, build the combinatoric set for the path planner to search, then use the planner's result to move the arm.

The Run Time Controller's highest level of functionality is a process planner and pseudo-English command parser which provides a front end to all existing RTC autonomous capability. It is, of course, available at all times to perform convenient services for the researcher experimenting in the laboratory. One common use is to employ the command language to avoid the details of controlling each individual motion of the arm:

```
grasp the Door_handle with the right arm
    at a location 2.5 cm below its top
    with an orientation of 90.0 dg left_tilt
    using compiliance
    please.
```

Self-explanatory.

One of the most important class of primitive functions in the language is the set of facilities which permits communication with the external subsystems devoted to manipulation and sensing. These primitives are so important because they permit the user to actually perform physical actions, either via software or manually through the interactive capabilities of **Thread**. These interfaces were implemented in the JPL Telerobot using a customized Network Interface Package (NIP) which supports definition of transaction protocols in real time by the various subsystems. This is obviously a very specialized communications interface, but it has been very simply and modularly integrated into the **Thread** interpreter, using facilities which permit the user to build communication packets on a byte-by-byte basis, which may then be transmitted and received using either the NIP or whatever other method is desired. Alternative communications protocols have in fact been implemented in **Thread** for various purposes, the simplest being a KERMIT-based protocol which allowed **Thread** software to interface through a modem to a personal computer, to create a **Thread** primitive called `prolog` which accepts Prolog language source code as a text string and remotely compiles and executes it on the personal computer, returning the result as another text string.

IV. Conclusion

Over the last two years, **Thread** has had roughly a dozen users who have produced a total of over thirty thousand lines of **Thread** code, for many different applications, and **Thread** has proven to be quite useful both for off-line algorithm development and especially for experimental work with actual manipulators.

Unfortunately, another lesson learned during the development of the Run Time Controller is that the magnitude and severity of the software development task for large systems such as the Telerobot is still not taken seriously by some of the robotics community [15].

The original decision to use **Thread** as an interactive development operational environment led us to a much more cooperative man-machine viewpoint on the Telerobot than is typical for either traditional robotics or teleoperations, and this has had a strong impact on our understanding of both autonomous and teleoperated systems. One observation made is that the benefits in capabiliity which accrue from sharing functions between the human and the robot are very signficant: a dual-arm single-joystick shared control facility was demonstrated to be able to allow the human to guide two arms simultaneously under master-slave compliant control, to perform motions which could not realistically be planned by an autonomous system, with a degree of delicacy (forces applied to the carried object) which would be impossible to a human two-handed teleoperation system.

The increase in capability which occurs through sharing becomes even more drastic at higher levels than that of servo control, however, and also has the added advantage for space applications [1] that it is robust under time delay, which force-reflecting teleoperation is not. The interactive nature of the **Thread** development environment led to many instances in the laboratory when it was relatively easy for a member of our software development team to directly use the individual subsystem components (trajectory planner, world model, sensing/manipulation subsystem interfaces, etc.) to directly control the robot in an interactive manner to perform tasks which were beyond the ability of both the planning and reasoning capability of the existing autonomous robot, and beyond the dexterity of the teleoperation system.

Also, the interactive, multilayered architectural structure which **Thread** encourages is not at all restricted to non-real-time planning software, as a prototype extension of **Thread** to an interactive interpreter which allows complete control of servo-loop behavior at the lowest level has been built and successfully tested.

This work indicates that the proper direction for practical telerobotics [11] is to pursue a much more cooperative man-machine style of interface than either traditional supervised robotics or pure teleoperation.

References

[1] J. S. Albus, H. G. McCain, R. Lumia, "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)", *Technical Report, National Bureau of Standards, Robot Systems Division*, December 1986.

[2] J. Backus, "Function Level Programs as Mathematical Objects", *ACM Transactions on Programming Languages*, October 1981.

[3] F. P. Brooks Jr., "The Mythical Man-Month (Essays on Software Engineering)", *Addison-Wesley*, New York, 1982.

[4] M. Ewing, "The Caltech FORTH Manual", *Owens Valley Radio Observatory Technical Publication*, 1980.

[5] B. Faverjon, P. Tournassoud, "A Local Based Approach for Path Planning of Manipulators With a High Number of Degrees of Freedom", *Proceedings of 1987 IEEE International Conference on Robotics and Automation*, pages 1152-1159.

[6] E. G. Gilbert, D. W. Johnson, "Distance Functions and Their Application to Robot Path Planning in the Presence of Obstacles", *IEEE Journal of Robotics and Automation*, Vol. RA-1, No. 1, March 1985.

[7] A. Goldberg, "Smalltalk-80: The Language and Its Implementation", *Addison-Wesley Series In Computer Science*, 1983.

[8] A. Goldberg, "Smalltalk-80: The Interactive Programming Environment", *Addison-Wesley Series In Computer Science*, 1983.

[9] W. A. Gruver, B. I. Soroka, J. J. Craig, T. L. Turner, "Industrial Robot Programming Languages: A Comparative Evaluation", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-14, No. 4, July/August 1984.

[10] V. Hayward, R. Paul, "Robot Manipulator Control Under UNIX RCCL: A Robot Control 'C' Library", *International Journal of Robotics Research*, Vol. 5, No. 4., pages 94-111, Winter 1987.

[11] M. D. Montemerlo, "The Space Perspective: Man-Machine Redundancy In Remote Manipulator Systems", *Keynote speech, NATO Advanced Research Workshop on "Robots with Redundancy: Design, Sensing and Control"*, June 27-July 1, 1988, Salo, Lago di Garda, Italy.

[12] R. Paul, "Robot Manipulators: Mathematics, Programming, and Control", *MIT Press*, 1981.

[13] A. A. G. Requicha, R. B. Tilov, "Constructive Solid Geometry", *Technical Memorandum 25, Production Automation Project*, University of Rochester, 1977.

[14] G. Rodriguez, K. Kreutz, A. Jain, "A Spatial Operator Algebra for Manipulator Modeling and Control", *Proceedings of the 1989 NASA Conference on Space Telerobotics*, JPL Publication 89-7 (this proceedings).

[15] H. W. Stone, J. Balaram, J. Beahan, "Experiences with the JPL Telerobot Testbed - Issues and Insights", *Proceedings of the 1989 NASA Conference on Space Telerobotics*, JPL Publication 89-7 (this proceedings).

[16] R. Taylor, "Planning and Execution of Straight-line Manipulator Trajectories", *IBM Journal of Research and Development*, 23, 1979.

[17] D. Turner, "The Semantic Elegance of Applicative Languages", *ACM Transactions on Programming Languages*, October 1981.

[18] M. I. Vuskovic, "R-SHELL: A UNIX-Based Development Environment for Robotics", *San Diego State University Technical Report SDSU-CS-RL-03-87*, October 1987.

[19] P. H. Winston, B. K. P. Horn, "LISP", *Addison-Wesley*, Philippines, 1981.

MULTI-ARM CONTROL