Automated Design of Modular Field Robots

by

Nathaniel Rutman

BSME Stanford University (1991)

Submitted to the Department of Mechanical Engineering in Partial Fulfillment of the Requirements for the Degree of

Master of Science

at the

Massachusetts Institute of Technology

June 1995

Signature of Author ______________________________
Department of Mechanical Engineering
May 1995

Certified by [illegible]
Steven Dubowsky
Thesis Supervisor

Accepted by ______________________________
Ain A. Sonin
Chairman, Departmental Graduate Committee

Automated Design of Modular Field Robots

by

Nathaniel Rutman

Abstract

Robots are needed for important missions in field environments. A major limitation to the practical use of such systems is their high cost and long development time. It would be highly desirable to have systems that can be rapidly designed and fabricated, on the order of days or weeks instead of years. One approach to achieving this goal is the development of an automatic design methodology based on the use of standardized modular physical components.

The objective of this research has been the development of a framework for the automatic design of modular field robotic systems. Under this methodology, robots are assembled from sets of modular components. The assembly method as well as the component designs are based on fundamental solid engineering principles.

This thesis includes the generation of computer-based design search algorithms. The framework utilizes a hierarchical search over the possible robot assemblies. Physics-based rules are used to limit the search to "reasonable" assemblies. Assemblies are analyzed and ranked according to their ability to perform a given task. The methodology was used to suggest a robot for an inspection task aboard the USS Constitution.

Supervisor: Dr. Steven Dubowsky

Title: Professor of Mechanical Engineering

Table of Contents

Table of Figures

Acknowledgments

This work is the result of a research program sponsored by the NSF under its Small Grants for Exploratory Research (SGER) Program #IRI9320138 and by NASA grant #NAG11632.

I would like to thank my advisor, Dr. Steven Dubowsky, for providing the broad vision and direction for this thesis, and the opportunity to work on several interesting projects at MIT. I would also like to thank my fellow graduate students for their advice, humor, and fascinating discussions.

Chapter I: Introduction

This thesis studies the fundamental technical issues and problems of applying mobile robotic systems in complex, and possibly remote unstructured field environments, such as might be found in systems performing civil infrastructure inspection and maintenance tasks. The major focus of the research has been to develop a design framework so that robots can be assembled automatically from sets of modular components. The research has been performed in connection with a demonstration project called *The Project Constitution*.

Motivation

Robots are needed for field environments

Robots are needed for important missions in field environments [1, 2]. These systems could perform such important tasks as maintenance and disaster mitigation in nuclear power facilities, cleanup of toxic and hazardous waste sites and chemical accident cleanup, terrorist bomb disposal, infrastructure inspection, and commercial tanker hull maintenance [3, 4, 5, 6]. For many of these missions, robotic systems could remove humans from dangerous tasks or enter locations that are not readily accessible. In some applications, such as the inspection of the undersides of highway and rail bridges, robotic systems could also be very cost effective.

A great deal of research has been done to develop robotic manipulators (usually a fixed based single arm) for work in manufacturing cells structured specifically for them. Some research is now being done to develop field robotic systems that are able to perform missions where the task and environment are not well known and the system must be capable of mobility as well as being able to manipulate and investigate the environment [7, 8, 9], see Figure 1 [10].

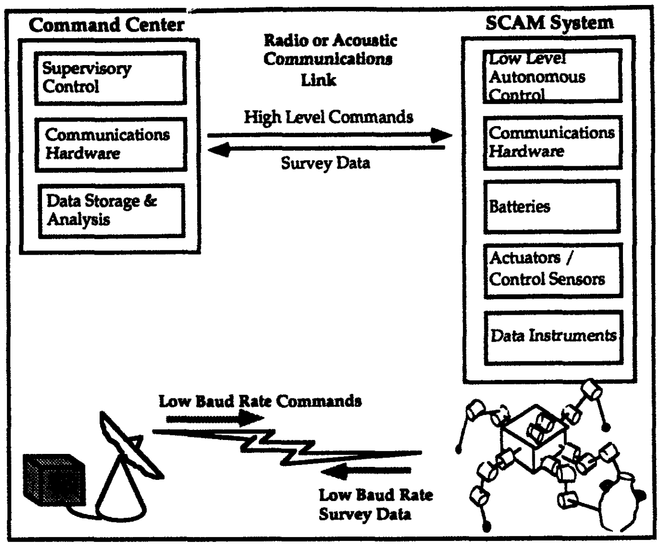

Figure 1. A concept for a field robotic explorer

Challenges facing these systems are that they must be robust, self-contained, power efficient, dexterous, agile, and have a high degree of autonomy. Besides these technical challenges, a major future limitation to the practical use of such systems is their cost and their development time. The design and development cost of such systems using current approaches would be prohibitive for many applications. This is largely because these systems will not be mass produced; each system would need to be designed for a specific mission or task. Not only will their costs be very high but the systems development time for a given mission could take years, when deployment in weeks or months may be required. For this reason it would be highly desirable to have systems that can be rapidly designed and fabricated.

One very promising approach to achieve this goal for field robotic systems is the development of a design methodology based on the use of standardized modular physical components, and control and planning algorithms and software that are compatible with a modular structure. To date, no other quantitative methods to rapidly design a field robotic system using modular components for a given mission have been developed.

Rapid deployment

Clearly, methods that would permit the rapid assembly and deployment of cost effective field robotic systems using standard modular components would make these systems practical for many important missions. As discussed below, in this research the *USS Constitution* (Old Ironsides) serves as a testbed for evaluating many of the research results. This permits the experimental testing of the design methods developed in this fundamental research program for a practical mission. The modular design techniques are used to suggest several candidate robotic designs for the Constitution.

Other advantages of modular systems

Using modular parts and designs leads to higher production volumes and lower part costs. Robots with modular components can be easily and quickly repaired by replacing a defective module. Additionally, the modular part inventory can be improved off-line, allowing aging modular robots to be updated to state-of-the-art technology with little redesign or down-time.

Objectives

The objective of this research program has been the development of a rational design framework for automatically designing robots based on task requirements using a set of modular components. With this modular design approach, robotic systems for missions in unstructured field environments can be designed and deployed rapidly and cost effectively. Computer based search algorithms are implemented to quickly and automatically find appropriate candidate designs.

It will be shown that simple tests can screen large numbers of alternative robot designs to quickly yield a few candidate designs which are believed to have the potential to perform a given task well.

This research does not attempt to plan the actions of the robot for the performance of the task. Instead, for a fixed task and solution domain, certain characteristics and capabilities are deemed necessary based on fundamental solid engineering principles. These characteristics are tested in a computationally efficient manner. Related research covered in other theses includes the development of modular planning [11] and control algorithms.

Background and Literature

To date a number of field robotic systems have been developed and proposed for specific applications. Systems have been designed to work on construction sites, to crawl through small-diameter gas lines, and climb up walls [4, 12, 13]. Systems are being developed for specific tasks in the service industry, to act as a nurse's aide, to perform sentry duty, to clean and to care for the elderly, or to prune grape vines [14, 15, 16, 17, 18, 19, 20, 21]. While some of these systems may prove to be effective, some have demonstrated the limits of current technology. For example, Dante, a system designed to explore the insides of volcanoes, did not perform as hoped during its initial field tests because of difficulties with its umbilical cable [22]. Clearly, if the technical problems with designing self-contained systems could be solved these systems would be more robust.

In the past, most design studies in robotics dealt with the problems of "classical" fixed-base industrial manipulators. Many of the results of this work have entered engineering practice [23]. More recent research has focused on such issues as the development of new and innovative components, sensors or computer architectures for robotic systems [24, 25, 26, 27]. Recently, significant research has been devoted to developing micro-mechanical precision and new exotic components for robotic applications [28, 29, 30, 31, 32]. However, robotic systems for field missions need to be quite different from conventional fixed base manipulators commonly used in industry, see Figure 1.

Research into designs with potential use in field applications has studied mobility methods based on walking, climbing and crawling [6, 9, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42]. As a result of this research a number of very innovative specific designs have been developed and demonstrated for field systems and for simple laboratory systems [8, 43, 44, 45, 46, 47]. Also some interesting lessons on the design of mobile robotic systems have been obtained from studying biological systems, such as animals and insects [48, 49, 50]. Some studies of mobile systems have attempted to develop design methods based on fundamental mechanics, such as developing design rules for motor selection to avoid actuator saturation and to minimize systems power consumption [7, 51]. It is interesting that most of the studies on the design of field robotics referred above have largely focused on achieving mobility without manipulation. However, there have been some important exceptions [52].

What is clear from an examination of the past research in field robotic systems is that is has either dealt with development of specific technology, or with a specific "one of a kind" system. Little or no work has been done to develop general methods to aid in the rapid design of field systems. It would be of great benefit to have general methods for designing field systems that consider the important attributes for a given task, and then quickly yield the configuration and design parameters of robotic system that is capable of meeting the mission's functional requirements.

In recent years there has been some important work done exploring some of the issues of industrial manipulators constructed with modular components. These studies include research dealing with the mechanical design [53, 54] the kinematic modeling [55, 56, 57], the enumeration of assembly configurations [58], the configuration selection based on computer aided design techniques [59] or on expert systems [60], the design based on task requirements [61, 62], and fault tolerant design [63]. This research has focused on industrial-type manipulator systems; systems consisting of relatively simple open loop chains of links operating from fixed bases. Such systems are very different from field systems that must be capable of mobility as well as manipulation, such as shown schematically in Figure 1.

The performance required of these field systems is quite different, arguably more difficult, than those for industrial systems [64]. For example the precision of an industrial system is often paramount. In field systems the need for the system not to turn over while moving, or for a battery powered system, not running out of power [51, 65] can be the critical issues. Such issues can make any design method very difficult, in part because in a field environment, unlike a factory, the task and the environment are not well controlled, or possibly even well known. This makes the modular design a challenging problem. While we are aware of one specific field system designed with a modular character this was done on an ad-hoc basis [4]. The development of general techniques to design modular field robotic systems remain a virtually unexplored problem. Such design methodologies would need to consider the fundamental limitations of the physical hardware, such as actuator saturation characteristics and battery life profiles that are available for field robotic systems. They should be able to quickly assemble system designs from modular subsystems and components. These methods should be based on fundamental engineering principles to insure they have the flexibility to be used for a wide range of systems and missions.

Modular Design

Modular robots

The key to this approach for achieving cost-effective and rapidly deployable multi-limbed mobile field robots is the use of modular components. Figure 2 shows, schematically, a relatively small set of modular components. Figure 4 and Figure 4 contain just three of the many possible systems that can be obtained from this inventory of modules. A real inventory might contain more, but possible not a great deal more, component types. Components of various sizes might also be included.

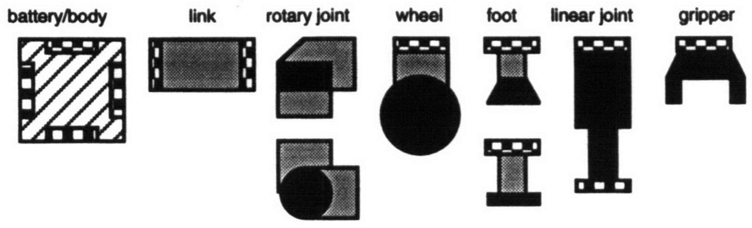

Figure 2. A sample set (inventory) of modular components

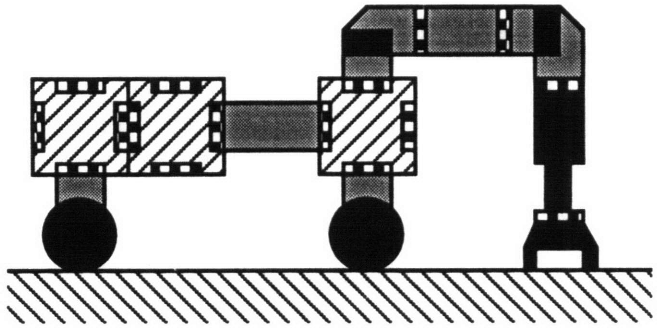

Figure 3. A system assembled from a limited inventory of modular components

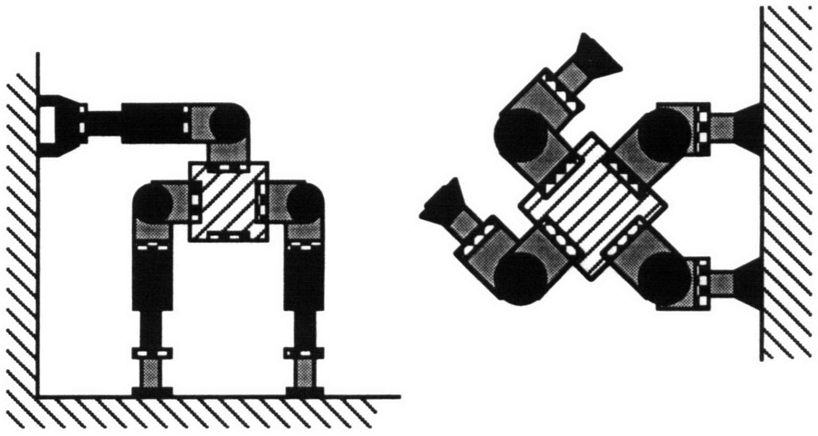

Figure 4. More assembled robotic systems

Small inventory of parts yields wide selection of robots

Modular systems make sense in the context of automated robot design and assembly. As shown in this thesis, using a simple set of fixed rules, modular systems can be automatically generated and tested. Combining a small set of modules in different ways permits an assembly space of many topologically diverse robots. A modular system can automatically span this space.

Most of the possible robots are not useful

Of the many possible robots, only a small fraction are useful. The vast majority will contain one or more features that precludes the possibility of a useful robot. If these features can be recognized, then such robots might be quickly discarded, or avoided altogether.

There are too many to consider in a conventional way

Even with modern high-speed computers, the combinatorial explosion of the number of robots precludes analysis of every possible assembly.

The number of robots (actually, the number of kits, explained below in Kits, page 26 - basically, a kit is an unassembled group of modules) that can be produced by taking exactly *r* modules of *n* unique types with replacement is $\frac{(r+n-1)!}{n!(r-1)!}$[66]. Thus, using up to 30 modules of 12 unique types yields 11 billion possible robots (note the expression given is for exactly *r* modules, not up to *r* modules. The sum of these expressions with *r* running from 1 to 30 yields the 11 billion.) If the entire algorithm took only 1 millisecond to conduct per robot, a computer would still require 127 days to execute.

Hierarchical selection process to avoid combinatorial explosion

The key to the development of a practical method for automatically generating robot designs is a hierarchical filtering process that reduces the design space at every stage, see Figure 5. At each added level of complexity (topological or structural) broad regions in the design space are eliminated, minimizing the total number of assemblies to be analyzed. This method is hierarchical because it eliminates entire sub-trees from consideration, multiplying its effects in every subsequent level.

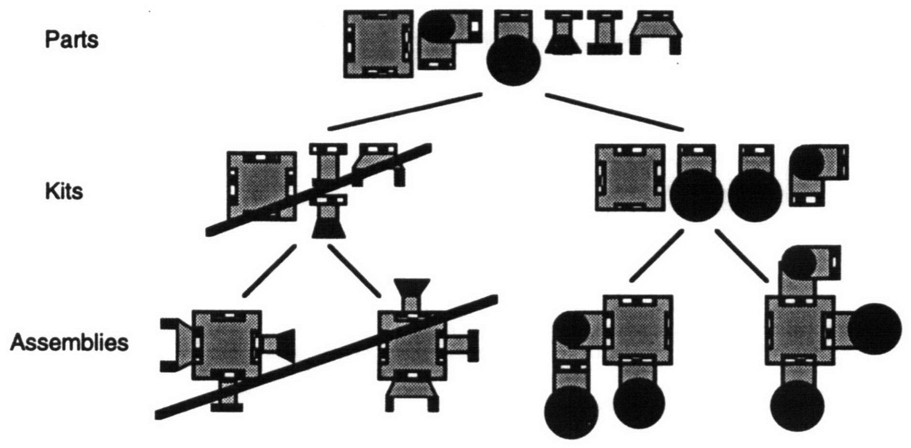

Figure 5. The multiplying effects of hierarchical filtering. Removal of a bad kit eliminates the subsequent analysis of all its child assemblies

Simple tests can quickly distinguish between remaining assemblies

Assemblies that make it through the filtering process must be evaluated in terms of how well they can perform the required tasks. A numerical objective function based on diverse features of the assemblies is used in a search for the best candidate robot for the task. This objective function is the sole element determining the outcome of the search. An important aspect of this research has been the development of a computationally cheap objective function that can reasonably predict robot performance, and therefore deliver useful robots.

Evaluation

This thesis seeks to address the advantages and limitations of the modular design approach. The research uses simulations and analytical studies to evaluate the results obtained. This approach has been undertaken within the context of multi-limbed field robots, using *The USS Constitution* as a demonstration project. The modular design techniques are developed and implemented in a software system, a modular inventory is designed, and the system is used to suggest several inspection and treatment robotic systems for the Constitution.

Chapter 2: Continuous vs. Modular Design Space

In some important ways, the design of a modular system can be simpler than the design of a conventional system. The design space of a modular system, given a set of available modules, can be represented by a finite set of possible assemblies, while the non-modular design space is, in theory, infinite.

Figure 6 conceptually shows a two dimensional non-modular (continuous) design space. The vertical axis symbolizes the robot performance, while the two independent design variables (α and β) can assume any value in their range. Figure 7 shows the corresponding modular design (a_i, b_j, $i = 1, 2, \ldots n$, $j = 1, 2, \ldots m$). There is now a finite series of systems that can be composed of two components that are allowed only discrete values. The values are constrained by the available inventory, made up of *n* a-types and *m* b-types. For systems with more components the dimensionality of the space would grow. While the modular space has a finite set of components, the number of assemblies that can be made, as mentioned above, grows very rapidly. For any real problem an exhaustive evaluation of all possible designs is out of the question. The key to a practical search lies in reducing the search space to a computationally feasible size. Such a space is indicated in Figure 8 where a number of the discrete design values have been eliminated using a method such as the hierarchical physics based method described by this thesis. In Figure 8 only systems with estimated good performance remain. At this point it becomes feasible to evaluate the performance of these "good candidates" in detail.

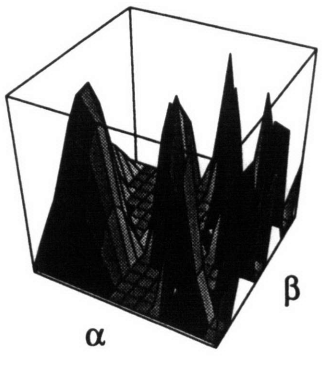

Figure 6. A continuous design space of a conventional system

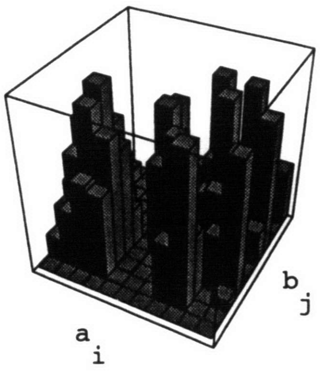

Figure 7. A discrete design space of a modular system

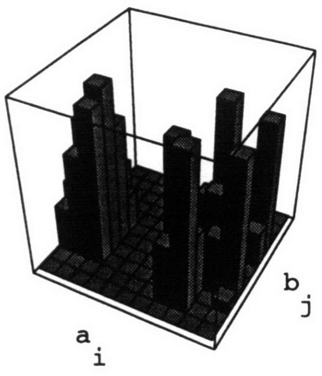

Figure 8. A reduced discrete design spacc of a modular system

Chapter 3: Modular Design Method

In this research structured hierarchical procedures are formulated to search and evaluate the performance of possible assemblies of modular components. The methods are framed to exploit the physical nature of behavior of these systems and their tasks and environments. The methods also attempt to achieve effectiveness by recognizing that some of the performance characteristics of these systems are much easier, and computationally efficient, to predict than others [67]. For example, it is much easier to evaluate a candidate design configuration's static stability - if it will fall over - than the communication data rates required to control it in a given mission. The modular search algorithm applies the simplest tests first to prune the set of all possible designs and quickly converge on a small set of candidate assemblies of modular components. Only the successful candidates need to be considered by the later, more computationally intensive tests [7].

Figure 9 shows the search method schematically. This figure is referenced and explained throughout the remainder of this thesis.

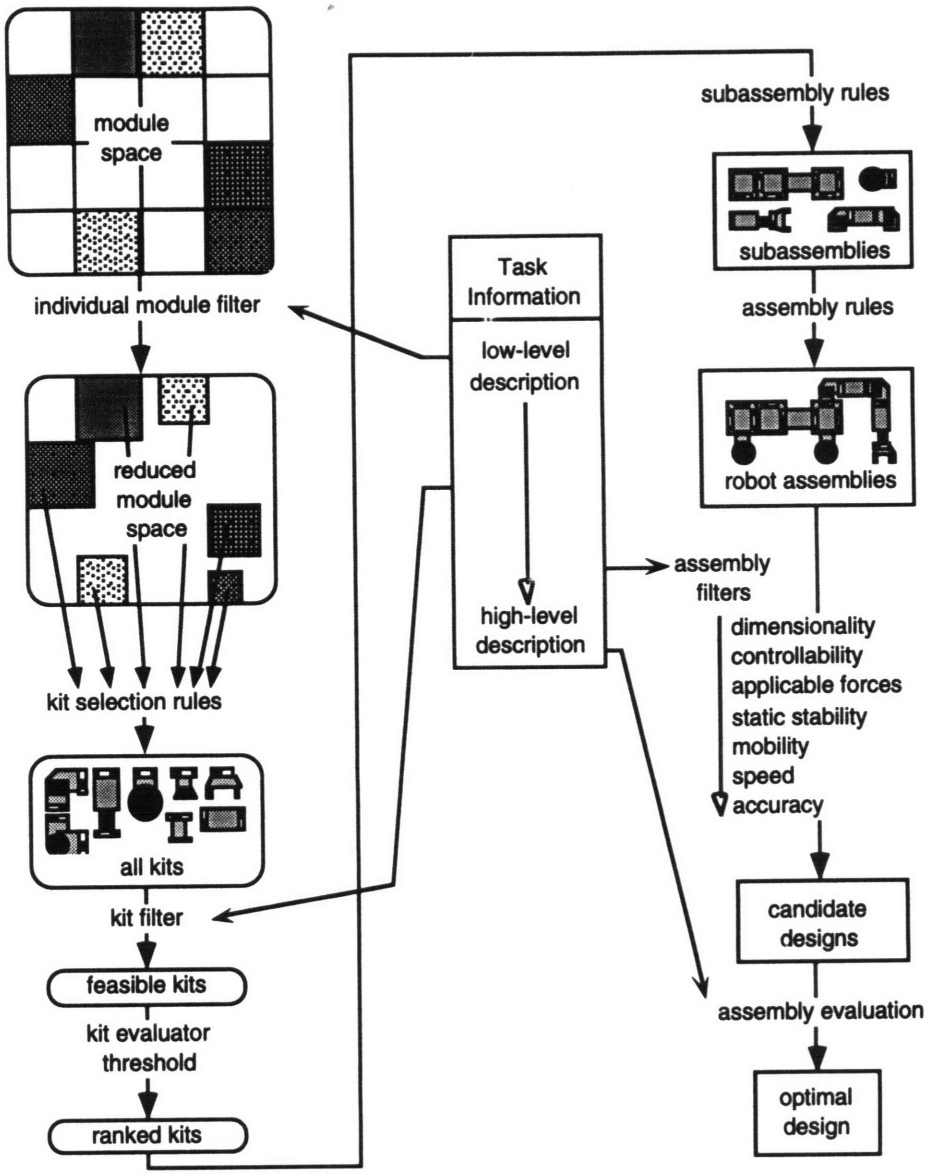

Figure 9. A physically structured search process

Domain

Task domain

The modular design approach has been applied to the domain of field robotic applications in partially known, complex environments.

Solution domain

The solution domain is restricted to the class of multi-limbed, static walking robots. These robots should be self-contained in order to function in extreme field environments. In general, the robots are capable of force manipulation of the environment to some degree.

Task Structure

Sample task

For purposes of illustration, the modular design method will be described with a sample task in mind. The task is taken from the USS Constitution, where a robot is needed to inspect wood surfaces in the bilge of the ship for rot. This area is shown schematically in Figure 10.

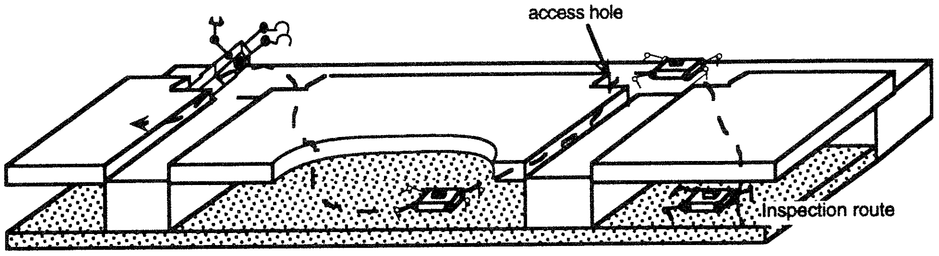

Figure 10. A schematic of the USS Constitution bilge area

Further details of this task can be found below.

Representation

If a detailed model of the task environment was known beforehand, highly specialized robots might be valid solutions. In Figure 11, the task is to reach the target with a gripper. Knowledge of the pipe diameter and material, exact location of the target object, size of the fitting, location of the bend, etc., as well as access to one end of the pipe would be required to implement the pipe hanging robotic solution. Knowledge of the water extent, depth, composition, and target location might make a floating robot possible.

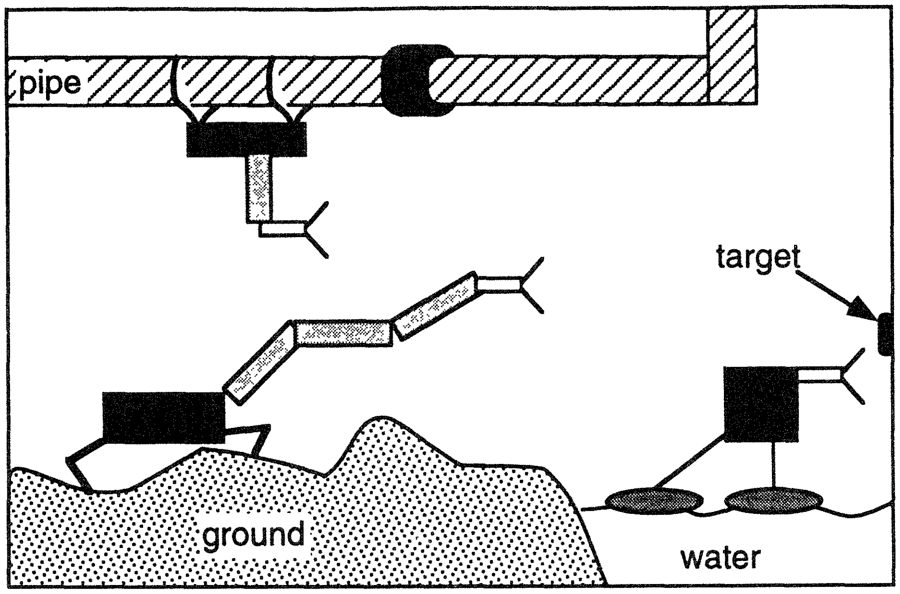

Figure 11. Analytical solutions depend on task representation

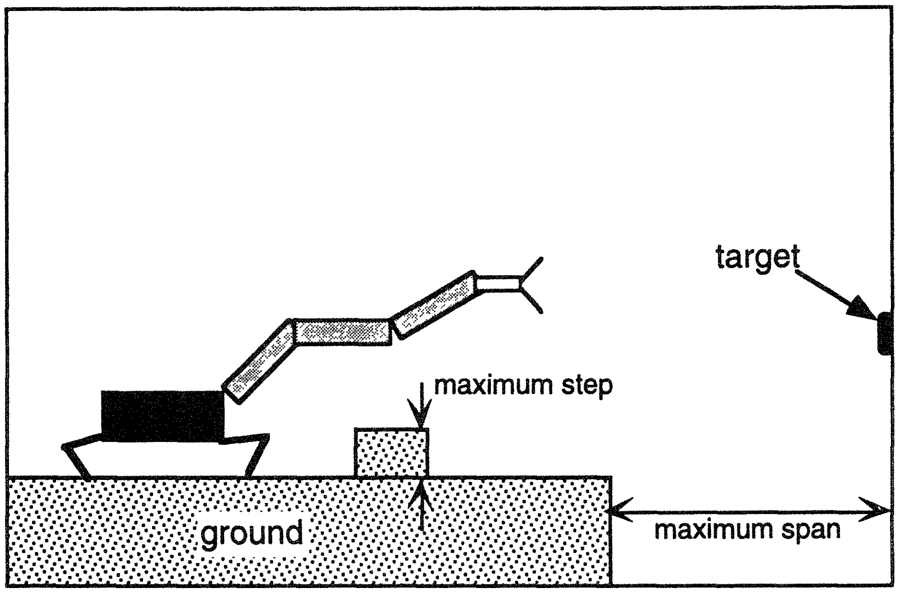

Figure 12. Simple task representation

A simpler task representation is shown in Figure 12. The task is described in simple terms with a limited number of concise constraints, such as the maximum span and maximum step required. Since the solution domain is limited to walking, limbed robots, the constraints were developed for this domain. If the solution domain included pipe-crawling or floating robots, then constraints appropriate for these domains would be included as well.

The task is described via these simple constraints for two reasons. First, because the task domain is in field environments, exact knowledge of the environment is unavailable. The constraint method eliminates the need for complete detailed environmental knowledge. Second, simple constraints lend themselves naturally to simple tests, which are the basis for the hierarchical search method.

If all the constraints are met, then it is assumed that the robot can perform the task adequately. This thesis does not involve the analysis of more complex plans [11] to determine task adequacy.

Weighting factors are also included in the task representation. These are used to achieve the desired mix of cost, running time, and reliability appropriate to the task. Other factors, such as robot mass, are not given weighting factors because they are used only as constraints; as long as the robot mass is under 9 kg, the robot is acceptable.

Constraints are geometric (maximum step size required, maximum span to be crossed, minimum hole size the robot must crawl through), static (maximum force to be applied against the environment), kinematic (maximum payload to be lifted), and environmental (all modules must work in the dark and wet for the sample task.) Additionally, a list of required end-effector capabilities (the sample task requires 4 feet that can operate on a horizontal wooden surface, and 1 gripper.) Other constraints include maximum mass allowed, minimum required operating time. Cost is also used as a constraint, as well as a weighting factor.

Assembly Structures

In the design methodology presented in this thesis, five levels of structure are used to develop robots, see Figure 13. First, individual modules are designed in full detail. These modules are then grouped together by function. Modules are then chosen from these functional groups and collected into kits.

From the kits, subassemblies are constructed. Finally, the subassemblies are combined into full robot assemblies.

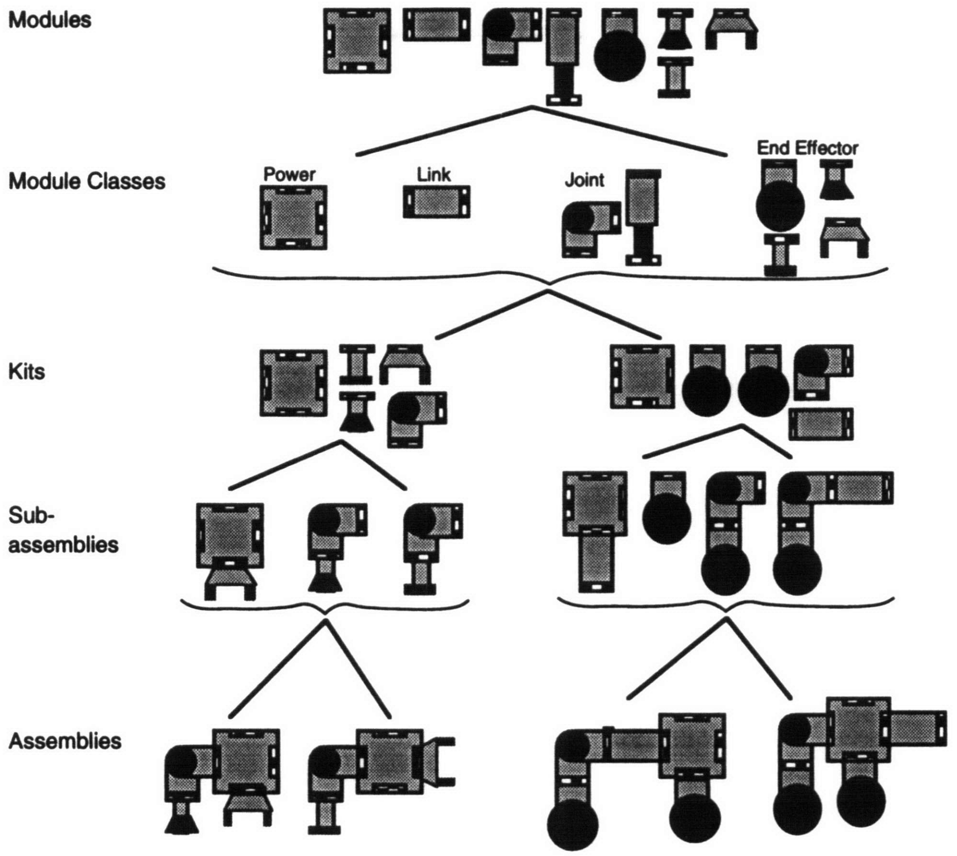

Figure 13. Levels of robot structures

These five levels of structure were chosen because general physical rules can be applied at each level. These rules can determine whether subsequent assemblies will be reasonable robotic solutions. Every structural level provides additional rules, helping reduce the solution search space. The organizational overhead of maintaining multiple structural levels is insignificant compared to the computational savings realized through the use of these rules.

Modules

The individual modules used in the automated design algorithm are designed in detail before the search can begin. Evaluations used during the search depend not only on general characteristics of the modules, such as classification type, but on actual physical characteristics as well, such as weight, maximum torque, power consumption, and material properties. Different evaluations and tests used throughout the search use models of varying complexity extracted from the full module model. For example, a test for static strength may depend on a lumped mass model of the modules, while a dynamic stability test may require a moment of inertia matrix.

By knowing all the details of the modules in advance, the refining process of traditional design is eliminated. A potential robot can be described completely from its modules and their organization. The robot's capabilities are accordingly completely calculable as well. This feature is necessary for ranking robots' task performances.

Table 1. A sample modular parts inventory

Name	Icon	Energy Type	Total Energy (w-hr)	Energy Usage (w)	Environments	Size (cm)	Mass (kg)	Max. Force (N), Torque (Nm)
Electric Power Supply	E	Electric	400	0.2	dry, damp, metal, ground	4.5 x 5.0 x 5.0	9.4	
Hydraulic Power Supply	H	Hydraulic	20000	50.0	dry, metal, ground, radiation	8.4 x 8.7 x 12.0	40.0	
Links		None	0	0.0	dry, damp, wet, metal, ground	4.0 x 1.0 x 8.0	1.5	
Hinge Joint/Actuator		Electric	0	5.0	dry, damp, metal, ground	2.0 x 2.0 x 3.4	3.4	1.0 Nm
Roll Joint/Actuator		Electric	0	2.0	dry, damp, metal, ground	1.2 x 1.2 x 1.5	1.7	0.5 Nm
Linear Joint/Actuator		Hydraulic	0	50.0	dry, damp, metal, ground	6.4 x 6.7 x 8.0	25.0	900 N
Video Camera		Electric	0	6.0	dry, damp, lighted	1.0 x 1.0 x 1.4	.2	
Laser Range Sensor		Electric	0	6.0	dry, damp, lighted	1.0 x 1.0 x 1.4	.2	
End-effector Foot		None	0	0.0	dry, damp, wet, metal, ground	0.4 x 0.4 x 1.0	0.4	
End-effector Gripper		Electric	0	3.0	dry, damp, metal, ground	1.4 x 1.4 x 3.0	1.3	

Module Classes

The modular parts inventory is divided into subclasses based on functionality. The four subclasses used are power supply modules, actuated limb segments, passive structural link segments, and end-effectors, see Figure 13.

- Power modules supply power to the other connected modules. Most commonly these would be battery based, but a gasoline-powered air pump might also be used as a pneumatic power source, or a module with a self-winding power cord attached to a fixed power station if self-containment was not necessary. Power modules would come in a variety of capacities. Also, because they are used in the center-body of all robots (see Subassemblies, below) power modules have multiple connection locations, or *ports*, where other modules can attach to them.

- A passive structural link segment, or *link*, passively connects two other modules together, rigidly, a fixed distance apart. Key features of links are material properties such as strength and stiffness.

- An actuated limb segment also connects two other modules. These modules consist of an active actuated joint of at least one degree of freedom, combined with a rigid beam. Because the major difference between these modules and links is the addition of the actuated joint, these modules will be called *joints* for the remainder of this thesis. Joints use power provided by an appropriate power module, and can be characterized by applicable torques and speeds.

- The final category is end-effectors, such as grippers or feet, or wheels in a wheeled system. End-effectors come into direct contact with the environment on one end, while the other attaches to another module.

All modules have some common features, such as size, weight, reliability, sustainable loads, and acceptable working environments. Some module classes have additional specific parameters - actuated limb segments and end-effectors have a required power type (i.e. electrical or hydraulic) and maximum torques. Power modules have an energy storage capacity.

These module classes were organized by functionality so that structural analyses could be simplified. This decision has worked well in practice, but it has proved beneficial to further subdivide the end-effector class into feet and grippers for certain analyses. Other organizations, such as by power type or operable environments, have not been found to be as widely useful in this search hierarchy, because they do not lend themselves to structural analysis as easily. These properties are therefore accessed as attributes of individual modules, and not as module classes.

Kits

A kit is an unassembled group of modules, which might later be assembled into a robot, see Figure 13. Typically, a kit can be assembled in many different ways, producing many topologically diverse robots. All modules within a kit must be used in the assemblies made from that kit. (If a module was not used, the remaining modules would simply constitute a smaller kit.)

However, there are some kits that can never produce useful robots, no matter how they are assembled. The physically based kit selection rules shown in Figure 9 eliminate these fundamentally flawed kits. For instance, a robot with a hydraulic power supply and electric actuators cannot function in any assembly; similarly, a robot made up of only "feet" will never be useful, see Figure 14. These kits are simply not produced by the modular design algorithm, effectively pruning large areas of the design space as in Figure 5 and Figure 8.

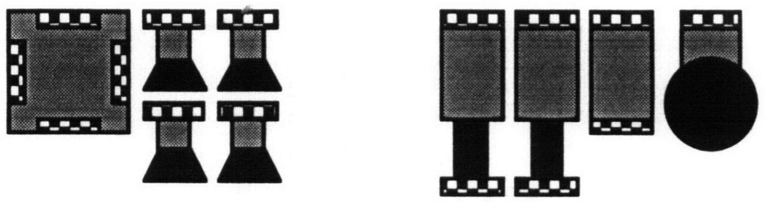

Figure 14. These kits can never produce useful robots

Subassemblies

Kits are then organized into subassemblies of a center *body* and serial-link *limbs*, see Figure 13. Subassemblies are created according to certain physical rules that eliminate "nonsense" structures. Every limb must end in an end-effector and contain at least one joint. The center body must contain all the power supplies (by definition) and can also contain links and joints, but not end-effectors.

The subdivision into center body and limbs is a consequence of the stated solution domain, multi-limbed walking robots. In other solution domains, different subassembly types might be required.

It can be argued that limbs do not necessarily need to be serial. This then blurs the distinction between center-body and limbs, and eliminates most of the consequential subassembly rules. If subassemblies are eliminated, as indeed they can be, more full assemblies must be tested.

The tradeoff between more structured and more open classifications, as in most areas of the modular design algorithm, is in the incremental performance benefit provided by allowing more assemblies versus the increased design space to be searched at later (and more complex) tests.

Assemblies

As shown in Figure 9, the subassemblies are combined in various arrangements into completed robot assemblies. This is facilitated by a minimal set of assembly rules, which simply insure that legs are connected to the center body, and that there are at least four legs (to allow static walking. This rule could be relaxed if dynamic walking was allowed in the solution domain.) Assemblies are fully defined representations of actual robots, and can be used in simulations or evaluations, or constructed out of parts from the modular inventory and inserted into the task field environment. Good assemblies will be able to perform the task well.

Assembly Representation

Modules are described in full detail. Assemblies can be described in full detail if the component modules are known, and the order in which they are connected is known. The assembly is therefore represented internally as a connection diagram. Different tests can extract the features they need from the assembly description and the module descriptions, as in Figure 15.

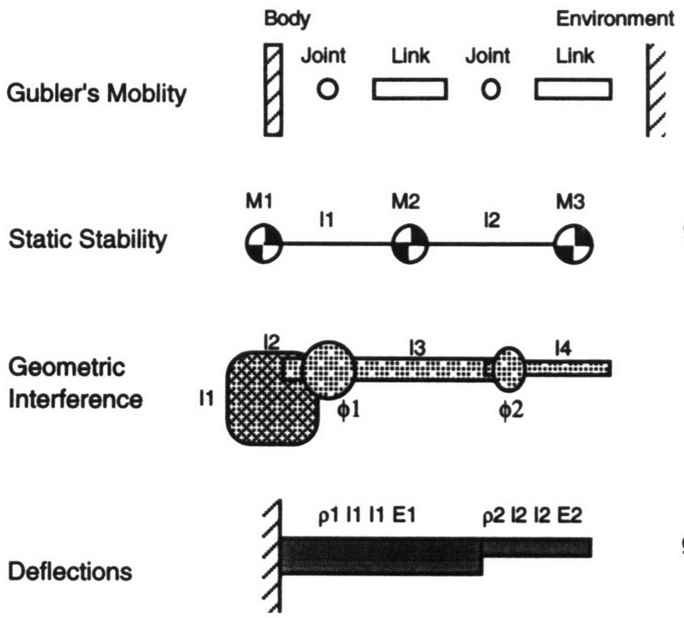

Figure 15. Assembly representations. The robot body and one limb are shown.

Figure 15 lists some assembly tests and the relevant information from the assembly that is needed by the test. These tests are described in detail in Chapter 6: Application to the USS Constitution. Grubler's mobility test requires only the type of joints or links and their order. A static stability test requires the relative positions of mass concentrations. A geometric interference check requires full dimensional knowledge and the location and ranges of joints. A deflection under loading test requires the lengths and material properties of the limb segments. Not shown in the figure, a full dynamic simulation would require mass moment of inertia matrices, relative positions, actuator torques and speeds, etc. These representations are derived as needed during the tests. See Table 2 for other test information requirements.

Table 2. Required information for tests.

Test	Required knowledge
interference	dimensions, motion ranges
static forces	dimensions, torques
mobility	topology, static forces
static forces	dimensions, torques
static stability	center of mass, foot positions
deflections	material properties, moments of inertia
accuracy	dimensions, backlash, deflections, sensor accuracy
dynamics	dimensions, joint velocities, moments of inertia, torques
kinetostatics	dimensions, joint velocities, joint accelerations, moments of inertia
power consumption	torques, actuator efficiencies
speed	dimensions, joint velocities, joint accelerations

These tests are designed to be independent of the task and the plan. They only use information available in the module descriptions and the assembly representation.

Search Hierarchy

Physical-based *rules* determine how assembly structures are constructed. Additionally, task-based *filters* are used at several points throughout the search to further prune the solution space. The distinction between rules and filters is that rules always hold true, regardless of the nature of the task, and filters are task-dependent. A rule might be: the robot must stand statically; and a filter might be: the robot must weigh less than 20 lb. Finally, numerical *evaluations* of the structures are used as ranking functions. An evaluation might give robots with low weights better scores. Evaluations can be task-dependent or not. Rules, filters, and evaluations are collectively referred to here as *tests*.

Filters

Filters remove from consideration any modules, kits, or assemblies that cannot meet certain task requirements. Note that filters do not insure that a passing candidate will be able to meet that task requirement - passing a filter is generally a necessary but not a sufficient condition to insure task compliance. For this reason, filters can be kept simple, saving computational effort for later in the search, when there will be fewer candidates to consider.

For instance, one assembly filter may be a test to see if a robot can walk. A full test of walking ability might require a full dynamic simulation of the robot, which, although possible, is a computationally expensive procedure. However, a quick-and-dirty walking test might be: check that each foot can be lifted individually without the robot falling over. This test will still eliminate many, but not all, candidates that cannot walk; foot-lifting ability is one of a number of requirements for walking.

Filters are arranged at each structural level according to their complexity and ability to eliminate assemblies. Computationally simple filters that eliminate a wide selection of robots are the most useful in terms of search speed; later filters that involve more computation will have to be applied on fewer robots. The effects of filter arrangements are discussed in "Effectiveness of Tests on Search Size," page 68.

Module Filters

The search process begins by considering all available modules in the inventory, see Figure 9. Modules are removed from the working inventory of parts if they do not meet some low-level task-based criteria - for example, since the bilge task involves exposure to sea water, only sealed sensor modules can be used. Any unsealed modules are removed from further consideration. Additionally, the task involves a spatially complex path, so a fiber-optic communications module with a tethered fiber optic line would be eliminated. Reducing the number of modules in the inventory reduces the design space as shown in Figure 8.

Using the expression on page 14, if just 3 modules are eliminated by module filters (for example, two hydraulic modules are too big to fit in the confined space of the task, and a videocamera module could not work in wet environments), the number of kits is reduced by a factor of 50 to 212 million.

Kit Filters

The kit filters, see Figure 9, insure that all feasible kits meet some aspects of the task requirements. The kits do not have to be assembled for kit filters to be applied. For example, the weight of the robot in the Constitution task is required to be under 20 pounds, so kits whose components sum total weight are greater than this are eliminated. Again, eliminating kits before they are assembled greatly reduces the design space to be searched, see Figure 5, so developing broadly applicable kit filters is an important aspect of this research.

Assembly Filters

Assembly filters eliminate assemblies based on aspects of the assembled structure. Many of the more accurate tests of the task requirements (walking ability, range of motion, or endpoint force) can only be addressed with a full assembly representation.

In the sample task, a robot is required to climb a step. A kit filter checks step height as a function of average limb length and rejects robots not capable of this action. A related assembly filter might check the step height based on the actual, assembled limb lengths. The kit version represents a necessary condition to climb the step. The assembly version is more complex, but represents a sufficient condition.

Extra computation time is involved in executing both filters for robots that can walk, given that the second (assembly) filter alone is sufficient. However, all of the kits rejected by the first (kit) filter have saved the computations involved in executing any remaining kit filters, setting up all possible assemblies to be made from that kit, and executing any assembly filters over all of these assemblies. The actual computational tradeoffs between test are described in Effectiveness of Tests on Search Size, page 68.

Robot Evaluations

Robots are evaluated for their potential task performance at two levels in the search structure; first for kits, then for assemblies. These evaluations are numerical and represent the quality of the solution. Robots with the higher evaluations are more appropriate for the given task. The evaluation at the kit level, including factors such as weight and cost, is used as an additional kit filter - low scoring kits are removed - and it is for this reason that the evaluation occurs at this level. The kit evaluation is included as a component of the second, assembly, evaluation. The assembly with the highest assembly evaluation score is considered the optimal robot for the given task.

Kit Evaluation

Kits that pass the kit filters are evaluated and compared to a threshold, see Figure 9. This evaluation is simple compared to the final assembly evaluation. The evaluation is the weighted sum of a series of tests, such as estimated reliability based on parts count, total cost, component inherent accuracy, power efficiency, weight, actuator speed, and predicted span. The bilge example robot needs to slowly probe closely spaced points for rot over an area of thousands of square feet for many hours without an umbilical. Therefore, for this task, kits with estimated low static actuator power consumption would be given a high evaluation.

Kits that have scores below a given threshold are filtered out at this point, under the assumption that below some level a kit is not worth considering, even though it may have passed all the filters up to this point. This threshold can be changed to influence search speed and search thoroughness. Reducing the threshold to zero will allow all kits at this point to be assembled, and raising it above some maximum will eliminate all kits.

Assembly Evaluation

The final assembly evaluation computes the effectiveness of a robot for the given task. Again, the evaluation function is the weighted sum of a series of tests that include bandwidth, range of motion, mobility, walking speed, endpoint force, accuracy, and redundancy. Fast, accurate, low-cost, high-force robots would generally score well. These evaluation function in general will be more complex than previous tests, but since the number of assemblies has already been reduced significantly, the total time required for the assembly evaluations is minimized.

In the most complete incarnation of the automated modular design method, complete kinematic and dynamic models of the modules and their arrangement could be passed to external dynamic modeling programs for evaluation, such as [68,69,70], for complete dynamic simulation. Results from this simulation would be reported back to the modular design algorithm to be incorporated in the assembly evaluation.

Chapter 4: Designing Modular Components and Inventories

Good Design Characteristics for Modular Components

In designing the inventory of modular parts used in this implementation, characteristics for good modular designs were investigated. Different and new technologies and materials for actuators, sensors, structures and energy storage elements were evaluated within this context. The characteristics below are true in general for modular systems, but of course there will always be exceptions. Some of these characteristics are well-known, while others remain relatively undeveloped.

Some of these characteristics were incorporated into the modular inventory used in the Constitution task.

Features

Structural Features

- The scale of most mechanisms (85%) is between 3 cm and 40 cm [71]. This might be an appropriate scale in which to develop modular mechanical parts as well.
- Circular cross sections of structural components should be used because their final loading orientations are unknown [54]. Circular cross sections bear loads equally well in all directions.

Gaits

- A wheel diameter must be twice the size of a leg in order to traverse the same step height [72], see Figure 16. However, wheels are more energy-efficient than limbs because they have no dead-time in their cycles [72]. Also, static stability on wheels requires no power. So, for size-dominated tasks, limbs are better; for energy-dominated tasks, wheels are better.

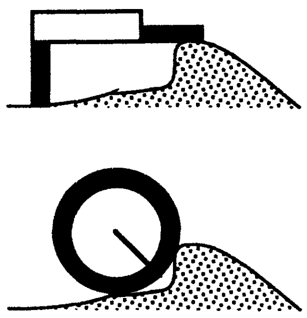

Figure 16. Leg step height versus wheel diameter

- Continuous (wheeled) gaits also lead to smoother motion over flat terrain, resulting in higher speeds, more efficient energy usage, and additional stability due to rotational inertia [72].

- Statically stable gaits are safer than dynamic gaits [72]. Because the environment is partially unknown, safety concerns dictate slower, more robust gaits.

Interfaces

- Symmetric couplings at modular connection interfaces allow for multiple orientations and redundant connectors [73].

- Hydraulic and other fluid-based modular connections are more difficult to make and maintain than electrical connections. On the other hand, they have higher power to weight ratios [54]. They should thus be avoided unless the task is power-dominated.

Transmission Features

- Most applications require high torques and low speeds [73]. Because the domain is field environments, and not manufacturing, robot actions are generally unique (non-repetitive), and because the environment is partially unknown, safety concerns dictate slower motions.

- Compared to transmissions, direct drive actuators are stiff, have no backlash, and low friction. However, they are backdrivable, have lower torque to weight ratios, and are optimized for a particular speed and load [54].

- Harmonic transmissions have high single-stage reductions, and are accurate, repeatable, and efficient [54].
- Non-backdrivable actuators require power only when moving. For slow moving, statically stable robots (typical of field applications) these will be more energy efficient.

Technology

The following Table 3 [74,75,76,77,78] lists some well known and some unusual actuators with quantitative and qualitative ranges in which they are effective. This list is not comprehensive but is included to show the wide breadth of the field typically ignored during the design process. Modules can be refined and improved as new technologies become available, and modular robots can be improved in the field, with minimal down time.

Table 3. Some actuator types and characteristics.

type	dimension	frequency	force	power output / weight	work energy density	comments
electro-static	50 μm	high	5 gF	high	.4 J/cm^3	difficult output coupling
piezo-electric	1 mm	100 kHz	4 kgF/mm^2	high	5e-4 J/cm^3	
ultrasonic	1 cm	5 Hz	high	low		no trans-mission necessary
shaped memory alloys	.1 mm	70 Hz	4 kgF/mm^2	high	10 J/cm^3	heat dissipation necessary
electro-magnetic induction	5 cm	100 Hz	high	low	1 J/cm^3	
rubber-tuators	1 mm	low	2 N	high		pressure differentia l to motion
magneto-strictive	2 mm	low	high	N/A	high	B field changes viscosity
electro-rheo-logical	2 mm	1 kHz	3 kPa	N/A	high	E field changes viscosity
chemo-mechanica l gels	1 cm	25 cm/min.	low	low		chemical availabili ty to motion

Design Via Frequency in Solutions

The automatic modular design algorithm can be used to help determine its own best inventory of parts. Varying the modular components available for a set of tasks and using the modular design algorithm to assemble the best systems, it is simple to note how often each module appears in the solutions. Modules that are utilized infrequently can be discarded. This in turn helps keep the modular inventory size to a minimum, allowing more rapid searches, and reducing the costs of manufacturing and maintaining the parts inventory, all without significantly degrading the performance of the systems produced.

The inventory can be constantly improved over time by adding new modules, and then trimming the size back with the above "survival of the fittest" approach.

Since statistical data on the best robot candidates can be recorded automatically by the algorithm, the algorithm could itself quite easily modify its own choice of inventory with this technique. Given a set time limit, it would choose the top *n* modules such that the search finished on time. A longer time limit would allow a greater selection of modules, reaching further into the more unique reserves.

Design Via Usage in Solutions

Modules can be refined and optimized individually in a more sophisticated manner as well. If usage data are collected instead of frequency for a range of tasks, then each module can be refined according to how it is most often used. For each successful robot design (with an assembly evaluation above some threshold), the average and maximum loads (for example) are recorded that an individual module sees, as determined through evaluation. If the loads are generally greater along a particular module axis, the module structural design can be modified to reflect the typical loading. The amount of material along subcritical axes can be reduced until all axes has equal safety margins, reducing the weight and cost of the module.

Similarly, if an actuator is generally driven in the upper end of its speed range, a higher speed actuator may be in order. If it is generally driven in its lower end speeds, then a cheaper, lower-speed actuator might be substituted, resulting in a cost savings with minimal performance impact.

This type of design refinement does not lend itself to automated improvement as easily as the frequency method. One can imagine that the usage data would be analyzed by an engineer, who would then suggest design improvements. However, it is possible that in controlled situations, module refinement could be completely automated. For example, the dimensions of a link I-beam might be adjusted automatically based on the typical loads it sees.

Design Via Modular Design Algorithm

Another approach to the design of the modular inventory might be this modular algorithm again, applied at a lower level. Now joints or end-effectors might be the desired assemblies, the task would be suitably redefined, and the modular design algorithm would be applied using an inventory of lower level modules such as motors, shaped memory alloys, and hinges. Although this approach was not used to design the modules in this research, the level of modularity is significant and is discussed below.

Chapter 5: Other Issues

Growth of Design Space

Increasing the number of unique module types available exponentially increases the number of kits to be searched, which in turn increases the search time. Because of this, significant effort has been spent trying to keep the inventory size as small as possible. Although it may be theorized that in general, the larger the inventory size the better the quality of the solution will be, the actual relationship of inventory size to solution quality has not been examined in this thesis.

Level of Modularity

It has become clear at this point that a different level of modularity could be built into the algorithm, see Table 4. The modular parts inventory could be limbs or bodies (similar to subassemblies, above) instead of joints and links. Well-designed limbs and bodies would be developed before the search began. These then would be combined into robots, and everything before this step in Figure 9 would be eliminated. This would greatly reduce the assembly search space, and explicit rules could begin to be written for every combination, approaching an expert system robot designer. However, such and "expert system" will never produce an original robot. In this case, it was decided that predesigning subassemblies would unacceptably limit the variety of solutions.

Modules could also be designed at the very low level of motors, gears, or even screws, but then the combinations rapidly increase, and the combination rules become exceedingly complex. The design space approaches the continuous space, and the advantages of modularity are lost.

Table 4. Module Levels. These are characteristic points on a continuous scale.

Module Level	Example	Pros	Cons
low	gear, motor	greater design freedom	long search time, complex connection rules
middle	rotary joint + link + interface	reasonable design freedom and search speed	
high	6-DOF arm	fast search, can guarantee good subsystems	small design freedom, maybe no solution

In general, the lower the level of modules, the larger the search space, and the better the solution, but the longer the search will take. Solutions are better for lower level modules in general because they can be more finely tuned to a specific task. The level chosen for this design algorithm is therefore in "the middle," low enough to produce a wide variety of robots, but high enough to hide the very complex layer of fine details and allow simple combination rules, and execute in a reasonable amount of time.

Search Method Comparison

Gradient

Traditional gradient-based search techniques [79] cannot be used for this search problem. The space of modular assemblies can be considered by definition non-continuous, and therefore no gradient exists. In a physical sense, this means that two robot assemblies that are extremely topologically "close," say only a single module is different, can have widely varying performance evaluations. No information about which robot to try next can be predicted via gradient methods.

Branch and bound

The quality of a branch and bound search is dependent on the accuracy of a guaranteed underestimator of robot performance [80]. For this application, because the performance is dependent on a hierarchical series of tests, a single simple and accurate performance underestimator is impossible. Branch and bound searches might be performed on various subcomponents of the main search. This method might produce a better robot testing order. However, because of the hierarchical nature of the of the search, the "bounding" portion of the branch-and-bound technique would have to be designed to take this in to account.

Simulated annealing

Simulated annealing is a probabilistic process that traverses the solution space in a somewhat random manner [80]. This technique does not span the solution space, and does not guarantee optimality. Additionally, simulated annealing in general requires lengthy parameter tuning for convergence, and so is not appropriate for an automatic algorithm meant to cover a wide variety of circumstances (tasks, modules, rules, and evaluations.)

Genetic algorithms

Genetic algorithms by their nature identify good design components in complex systems [81]. This leads to the concept of related robots - a highly ranked robot is encouraged to have similar children, with the hope that some of them may be better than the parent.

Like simulated annealing, genetic algorithms do not guarantee an optimal robot. However, they do not require tuning and so might be used for the modular design problem. The filters, rules, and evaluations developed in this research can be applied to a genetic algorithm search as well; if the optimal robot is not required, then genetic algorithms might be a good search method to use for a faster solution.

Genetic algorithms might be incorporated into the hierarchical modular design algorithm in the following way. The search is conducted as before, except that when good robot assemblies are produced, some representation of them is added to a gene pool, see Figure 17. Good is defined as a high scoring assembly evaluation. There may be two gene pools, one for kits and one for assemblies. Subsequent kits and assemblies are chosen via the genetic algorithm procedure, instead of in a regular order.

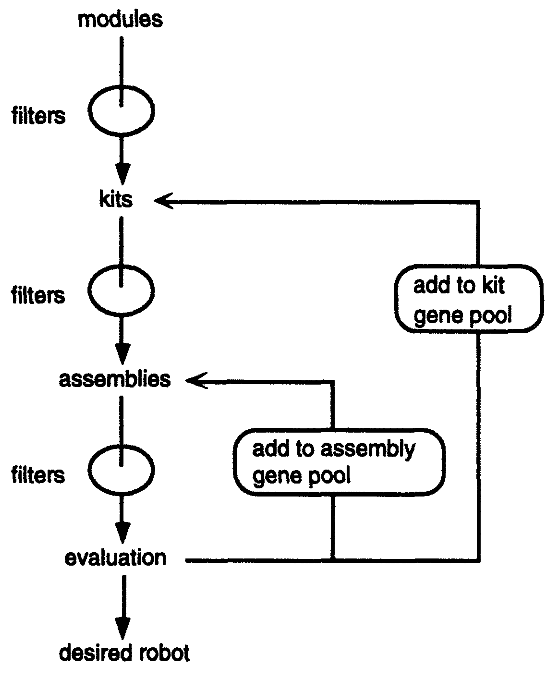

Figure 17. Adding a genetic algorithm search

Learning

The statistical learning process for designing the modular inventory described above could also be applied to more complex structures, such as subassemblies. Subassemblies that are used frequently could be remembered and tried first in future design searches. This would improve the average designs in a time-limited search.

Sensing, Control, and Communications

Robot parts such as sensors, communication devices, and control units are not presently included in this algorithm. These types of parts are generally decoupled from the structural design of the robot in a modular system. Because these modules are non-structural, and do not effect the topology of the robot, they can be added after the rest of the robot is built. For instance, if a robot needed to detect rot, a rot-detector sensor would be needed, independent how the robot was designed. The robot could be designed using the search algorithm without the detector, and the detector would be added after the solution was found. If the detector did affect the robot structure - if it weighed a significant amount, for instance, then those features should be factored into the robot design by including them as task requirements - a fixed payload with a certain weight and power consumption.

Chapter 6: Application to the USS Constitution

Constitution Task Description

To insure practicality of the modular design, the system was developed with a demonstration task in mind. The example task is taken from a project to restore and maintain the USS Constitution, a historic naval warship. The task is to inspect an area of the Constitution in the hold beneath the ballast supports, see Figure 18, locating any areas of rot in the wooden beams of the ship, and reporting this information, see Figure 10. This sub-ballast support area is from 5 to 8 inches high, 4 feet wide and 150 feet long. The area is entirely wood, and the environment is usually damp and dark. There may be puddles of water 1-2" deep in the inspection areas, and water may drip from above. There may be spaces between the floor boards of 1″. Additionally, the area is compartmentalized by diagonal riders spaced every 102″ along the ship, necessitating the ability to climb 8.5″ out of one compartment and into the next. The narrowest path is 4″x5.5″, see Figure 19 and Figure 10. Inspection includes video transmission of rotten areas, and a probe to be inserted with 2 lb. force perpendicularly into rotten areas. The inspection must be self-contained because of the complexity of the path (no cables), although not necessarily autonomous, and moving speed should be at least 10 feet per minute.

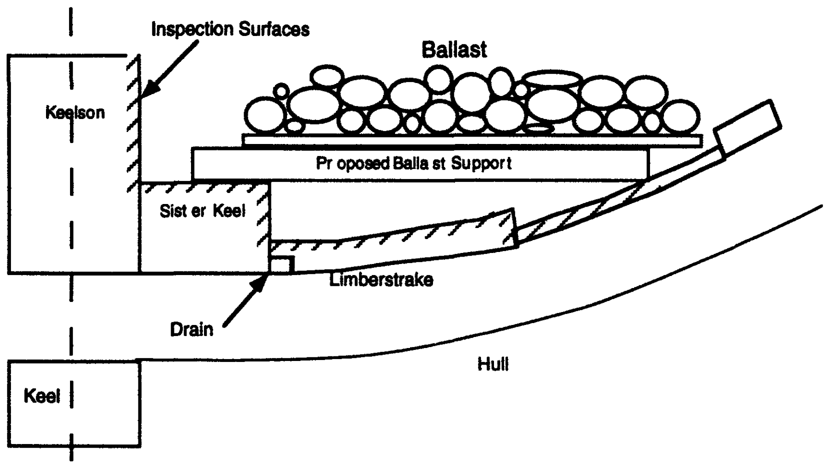

Figure 18. Constitution hold cross-section

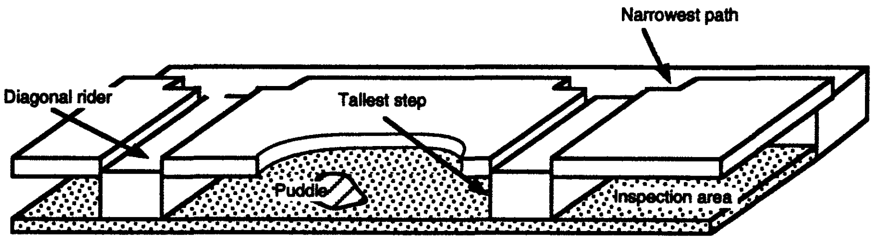

Figure 19. View looking inboard. Diagonal riders compartmentalize the space.

Furthermore, there may be loose obstacles up to 4" x 4" x 4", 1/4 lb. in the inspection areas. The robot might have to move these objects to complete its inspection. The robot should weight less than 20 lb. for to allow easy placement and removal. Total cost must be under $1000. This cost is based upon having pre-manufactured robot modules available for use.

This task description is translated into a series of constraints as follows:

- Minimum step height required: 0.2125 m
- Minimum span: 0.088 m
- Maximum weight: 9.09 kg
- Working environments: Dark, Wet
- Needed end-effectors: 4 feet, 1 gripper for moving 0.12 kg loose obstacles
- Maximum size allowed: 0.088 m x 0.1375 m x 0.088 m
- Maximum cost: $1000

Parts inventory

The modules implemented in this application were designed using traditional design methods. Although they have not been constructed, they are theoretically accurate in terms of materials, weights, dimensions, stresses, power consumption, torque values, speeds, cost, and motor availability. Costs were estimated based on relatively large-scale production (100 units of each module.)

The inventory includes 12 modules, as shown in Table 5: electric battery packs in two sizes, a gasoline-powered pneumatic power supply, a short link, three sizes of electric joints, two sizes of pneumatic joints, a pneumatic three-fingered gripper, a shaped-memory alloy electric two-fingered gripper that can be used as a foot, and a simple rubber-padded pivoting foot. In Table 5, the ID number is a unique identifier label. The acceptable working environments of each module a are denoted by capital letters: Wet, Dark, Vacuum, Radioactive, high Temperature. Dimensions listed are external, and an effective length (eff), if applicable, is the distance from connection point to connection point. Notes include energy capacity, number of interface ports, a maximum torque which the module can withstand across its effective length without failing or being backdriven (support torque), and the maximum applicable torque (apply torque.)

Table 5. Modules used for Constitution.

Module ID#	Type	Energy type	Materials	Environ ments	Mass (kg)	Cost ($)	Dimen-sions (cm)	Notes
110	power	electric	lead-acid, 2024 aluminum	WD	0.36	30	12x4x4	13.4 W-hr, 5 ports
112	power	electric	lead-acid, 2024 aluminum	WD	0.14	17	8x3x3	5 W-hr, 3 ports
130	power	pneu-matic	12-cc gas power plant, bulb pump	WD	1.1	70	16x9x9	100 W-hr, 6 ports
270	link		7075 aluminum	WRVD T	0.01	10	1x5.5x1.2 4.3 eff	361 Nm support
310	joint	electric	Micro-Mo #3557, 7075 aluminum	WVDT	0.7	175	3.5x15x6.5 12.5 eff	1.02 W, 10 Nm support, 5 Nm apply, 0.91 rpm
314	joint	electric	Micro-Mo #1624, 7075 aluminum	WVDT	0.08	80	1.5x6.4x5.7 5.0 eff	0.09 W, 1.2 Nm support, 0.3 Nm apply, 0.80 rpm
318	joint	electric	Micro-Mo #1016, 7075 aluminum	WVDT	0.02	60	1x4.2x4 3.0 eff	0.05 W, 0.9 Nm support, 0.1 Nm apply, 0.91 rpm
330	joint	pneu-matic	7075 aluminum	WVDT	0.9	70	5x20x7 17.5 eff	5 W, 30 Nm support, 20 Nm apply, 1.0 rpm
334	joint	pneu-matic	7075 aluminum	WVDT	0.6	50	4x10x5 8.5 eff	4 W, 20 Nm support, 10 Nm apply, 2.0 rpm
410	end-effector, gripper/ foot	electric	NiTiNol wire .150mm dia, 7075 aluminum	WRD	0.03	40	1x5.5x5.5 4.3 eff	0.06 W, 6.45 Nm support, 3.0 N grip force
430	end-effector, gripper	pneu-matic	lead screw, 7075 aluminum	RDT	0.5	150	3x10x10 9 eff	5 W, 30 Nm support, 30 N grip force
470	end-effector, foot		2024 aluminum, rubber	WRVD T	0.005	5	1x2x1.5 1.4 eff	126 Nm support

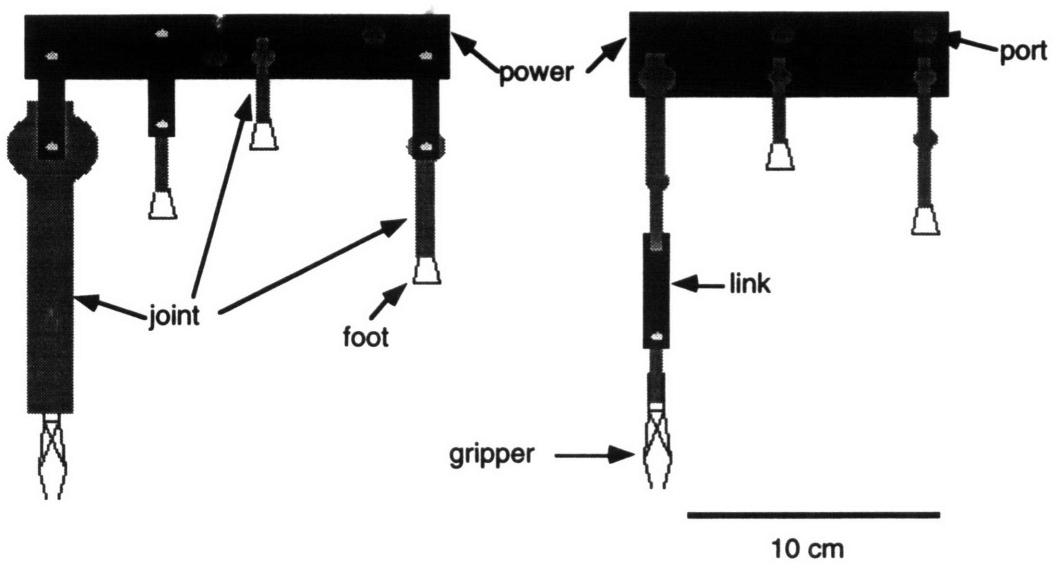

Figure 20. Modular robot output from the automatic designer

Figure 20 shows some modular robots that were assembled during the search. Limbs are shown attached vertically to various ports of the horizontal bodies. The bodies are made up of one or more power modules, and the limbs are made of different sizes of joint modules, links, feet, and grippers.

These robots are all bilaterally symmetric, so each limb appears on both sides of the body. Bilateral symmetry is not a requirement of the search method, but it seems to result in more reasonable robots for this task, and it certainly makes the graphical output of the code itself much more comprehensible.

The Search Process

The modular design methodology has been implemented in software (C++ on UNIX.) The following sections show how the algorithm works at a fairly high level. For more detailed look at the code, see "Appendix: modular design program", page 85.

Since each kit is evaluated sequentially, memory requirements are minimal; only the best few assemblies and the parts inventory need be remembered. The time required for the search is on the order of a few minutes to a few hours, depending on the inventory size.

C++ is particularly appropriate to this hierarchical design algorithm because of its class-handling abilities. There are direct analogues of the hierarchical assembly structures in the code. There is a general module class prototype with child classes of links, power modules, joints, and end-effectors. There is a module sub-group class that has a collection of one type of module. Then a group class is a collection of sub-groups, and is analogous to a kit. The sub-assembly prototype class takes a kit and adds connection information. It has child classes of body and limb. Finally, the assembly class connects subassemblies together.

At each level, details of lower levels are hidden. A particular detail can be specified by going through the chain of levels. Asking for the weight of a robot, the assembly class asks each of its legs what they weigh. Each leg asks the group of parts that makes up the leg what it weighs. The group then asks each module what they weigh. In this manner, code for each class is kept simple.

Describe Modules

The parts inventory is first read in from a file that describes each module in full detail. The constraints and weighting factors for the task are also read in.

The 12 modules available for the Constitution task yield 67,863,355 combinations when taken in kits of up to 17 modules (see the expression on page 14.) This number will be reduced by rules and filters throughout the search.

Module Filters

The modules are first filtered based on their individual size compared to the narrowest path requirement in the task description. In this case, the pneumatic power supply (ID# 130, 16x9x9 cm) is bigger than the narrowest path (given by the maximum size allowed (8.8x13.75x8.8 cm), so it is removed from the parts inventory. The pneumatic gripper (ID# 430) cannot operate in the task-required environment "Wet", and so is discarded. Each module weighs less than the maximum total weight, and each costs less than the maximum total cost. All of the modules are on an appropriate scale, determined by the range of dimensions given in the task within an order of magnitude.

Removing ID# 130 reduces the number of kit combinations to 30,421,300, and removing ID# 430 reduces them to 13,037,531.

Module Class Filters

After individual modules have been removed from consideration based on individual features, they are filtered functionally according to their module class. First, the power types of all modules are checked against power type availability from the power module class. In this case, since the only pneumatic power supply had been removed by the module filters above, modules requiring pneumatic power have no available power source, and so are discarded. The two pneumatic joints (ID# 330, 334) are removed from the available inventory in this manner. This reduces the number of possible combinations to 2,042,755 kits.

Second, the end-effector class is checked against the task's required list of end effectors. Any end-effectors that are not needed are discarded. In this case, end-effectors required by the task are grippers and feet, so both of the remaining end-effectors remain. Note that at this point, the number of task-required end-effectors is not important, because parts have not yet been selected for robot kits. Any end-effector that has a classification of 'foot' or 'gripper' is allowed to remain. A single end-effector can have more than one classification; for instance the remaining gripper (ID# 410) has both classifications according to the module description file, see Table 5.

Third, end-effectors are checked for specific requirements given in the task description - in this case the gripper must be able to lift objects weighing at least 0.25 lb. The remaining gripper (ID# 410) is capable of this, and so is not eliminated.

Choose the Next Kit

The first kit set up is the smallest that satisfies the kit rules below. Subsequent kits are insured to be unique and span the entire design space using the following hierarchical algorithm (each step is repeated until it cannot be, then the next step is executed):

1. Each module in each module class is swapped one for one into the kit, replacing another from that class, in ascending ID number order. There can be multiple instances of each module ID. For example, a kit with joint ID#s 310,310,310 would go to 310,310,314, to 310,310,318, to 310,314,314, to 310, 314, 318, to 310,318,318, to 314,314,314, etc. Note that order does not matter.

2. Module class types are changed from links to joints to end-effectors to power modules. A kit with module types power, link, joint, joint, end-effector goes to power, joint, joint, joint, end-effector, then step 1 is repeated, then goes to power, joint, joint, end-effector, end-effector, etc.

3. The kit is made larger by adding a link with the lowest ID number. A kit with module types power, power, power goes to power, power, power, link, then step 1 is repeated.

In practice, this algorithm has some of the kit rules, below, embedded in it so that kits that do not obey these rules are never generated in the first place. For instance, a power, power, power kit would be useless and is not actually generated.

The smallest kit is chosen first because cost generally increases with the number of modules in the assembly. Searches that are time-limited will therefore generally yield cheaper solutions.

Kit Rules

Each kit must have at least one power module to provide power, at least two end-effectors (on each side) to be able to stand up, and at least as many joints as end-effectors in order to move all its limbs. Each kit must contain power modules that can supply power for every type of module in the kit. There must be enough ports on the power modules to accommodate the number of limbs that there will be, which is the same as the number of end-effectors in the kit (because limbs are serial-link chains ending in an end-effector.) Finally, a provision for required groupings of modules has been included - each module can have a list of other modules that it needs to work with, and a list of modules that it should never work with.

In the Constitution robot design, there were 99,887 kits that passed all these kit rules.

Kit Filters

Kits are filtered on the following task requirements: total weight, total cost, total predicted reliability, operating time, end-effector requirements, step height, and span.

Mass and cost are simple sums over the kit: $\sum_i^n m_i$<M, $\sum_i^n c_i$<C, where n is the number of modules in the kit, m and c are the mass and cost of the individual modules, and M and C are the task-imposed limits.

Reliability is the product over the kit of the reliability for each module $\prod_i^n r_i$>R, where in a physically realized modular inventory, testing would determine module reliability r. For this Constitution task, reliability was roughly estimated by the complexity of the module (see Appendix for reliability values used.) Note that no minimum reliability R was specified in the task description, so R is effectively 0.

Operating time is the sum of the energy available from the power modules divided by the average power usage of all the modules: $\frac{\sum_i^p E_i}{\sum_j^n P_j}$>OTmin, where

p is the number of power modules, E is the energy available from each power module, and P is the average power usage of each module (energy divided by energy rate has units of time.) Again, no operating time is specified in the task.

The required numbers and types of end-effectors given by the task list must be met, so each kit must have 2 feet and 1 gripper or it will be skipped.

The span of the robot kit is an instance of a predictive filter: the actual span of the robot cannot be known until the robot is assembled, yet the robot is only in kit form at this point in the search. Even so, an estimate of the span can be made in the following manner. All of the non-power modules effective lengths are added together and divided by the number of limbs to get an average limb length. Two average limb lengths are added to the body length, which is taken to be the power modules connected in line, to get the predicted span: span = 2 x average limb length + body length. Unfortunately, this is not strictly a sufficiency test, because actual span may be less than or greater than this prediction. It turns that that in this case, because the required span is so small, all of the robots meet this requirement anyhow, see Effectiveness of Tests on Search Size. The step height is similarly predicted as twice the average limb length.

For the given task, the kit filters reduced the number of kits to be assembled down to 6,689.

Kit Evaluation

The kit evaluation objective function is the weighted sum of a series of factors: Kit_Eval $= \sum_i w_i \cdot f_i(s_i)$, where s_i is the numerical score from evaluation test i, f_i is a scaling function for test i, and w_i is the evaluation weight for the test. The evaluation weight can be either positive or negative. The scaling functions used are either linear or asymptotic, depending on the nature of the test, as discussed below. The two functions are $f_i(s) = \begin{cases} s; \\ 1 - \dfrac{1}{1+s}; \end{cases}$. The second definition provides an asymptotic limit (for positive s) of 1 with diminishing returns, see Figure 21. This is useful in cases such as operating time, where although more is generally better, after some point it becomes less and less important.

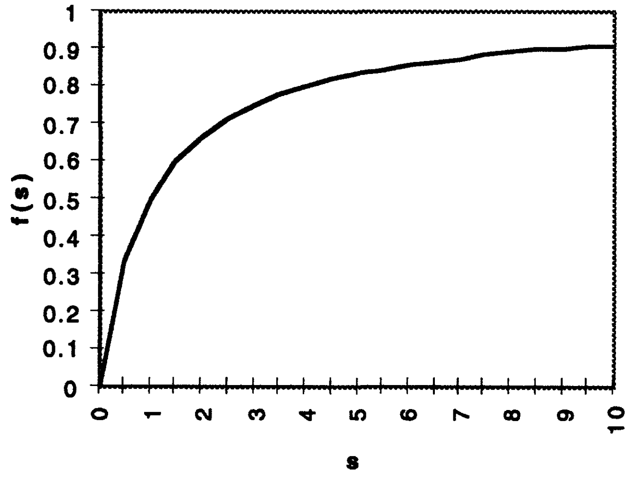

Figure 21. Asymptotic function

There are 7 kit evaluation factors used:

Kit_Eval = 5x Agility + 1x asymptotic Average_Limb_Length + 10x asymptotic Span + 10x Reliability + 10x asymptotic Operation_Time - 15x Mass - 10x Cost

Total weight and cost are linear with negative weights. Total reliability is linear and positively weighted.

Agility is taken to be the fraction of joints in the kit: $\frac{j}{n}$, where j is the number of joints and n is the number of modules. This encourages joint redundancy, because redundant robots are more robust to actuator failures and also can operate in more constrained spaces.

Average limb length and predicted span, as discussed above, are asymptotic and positively weighted, penalizing (relatively) robots that have extremely short legs or spans.

Operating time is also asymptotic and positively weighted, by reasoning that in general longer times are desirable, but with diminishing returns. This avoids 'fat' robots with many power modules when they are not specifically required by the task. (Operating time is a more practical evaluation than robot total energy efficiency because even a high-efficiency robot with a small battery may not last long enough to accomplish its task.)

Speed, although it is not yet included in the equation above, should be positive and asymptotic, and is measured as the average joint rotational speed times the average limb length.

If the kit evaluation meets a predefined minimum threshold, the kit is considered worthy of further exploration. A higher threshold results in reduced numbers of assemblies to be tried and therefore in faster search times, but at the expense of possibly missing some low-ranked 'sleepers', which might perform very well in the final assembly evaluation. In this case a very low threshold of 1.0 was chosen in order to further examine widely ranging possibilities.

Choose the Next Assembly

A kit can be assembled into many subassemblies, and these subassemblies can be arranged in many different ways. The following hierarchical algorithm insures that all possible assemblies of a given kit are examined. Again, this algorithm is simplified, because in practice some the subassembly rules, below, are embedded in it. Each step is repeated until its stopping point, then the next step in the sequence is executed:

1. Limb subassemblies are arranged about the body using all possible ports. For 4 ports and 3 limbs, limb 1 is first at port 1, limb 2 at port 2, limb 3 at port 3. For the next assembly, limb 1 is at port 1, limb 2 at port 2, limb 3 at port 4, etc. All permutations are covered.

2. New limb subassemblies are made. Step 1 is repeated. Limb subassemblies are made via three processes.

   a. A single subassembly is rearranged. A limb subassembly of joint 1, joint 2, link 1, end-effector goes to joint 1, link 1, joint 2, end-effector. The end-effector is always required to stay at the end of the limb. If all permutations of the limb have been tried, the next limb is rearranged.

   b. If no more limbs can be rearranged individually, modules are swapped to different subassemblies. Limb 1 and limb 2 exchange parts. All permutations are covered.

   c. Limbs with different numbers of modules are tried. Instead of swapping, one limb donates a module to another limb, skewing the limb sizes which originally started off as equal as possible.

3. A new body subassembly is made. Step 1 is repeated. New bodies are made via two processes.

   a. First the body is rearranged, as in 2.a.

   b. Modules are taken from the limb subassemblies and added to the body. A body with power 1, power 2, and a limb with joint 1, joint 2, link 1, end-effector are changed to a body of power 1, link 1, power 2, and a limb of joint 1, joint 2, end-effector. Only joints and links can be moved from a limb to the body. This process stops when only minimal limbs are left. Minimal limbs are a single joint plus an end-effector.

Subassembly Rules

The body subassembly may consist of power modules and links only. Limb subassemblies may consist of links and joints, and are required to have exactly one end-effector. Each limb subassembly is required to be able to lift itself (off the ground). This is tested in the following manner, as shown in Figure 22: each joint in the limb must be able to rotate through a set angle (30 degrees) and lift all modules attached below it, assuming all other joints are relaxed. This insures that each joint in the limb is not useless.

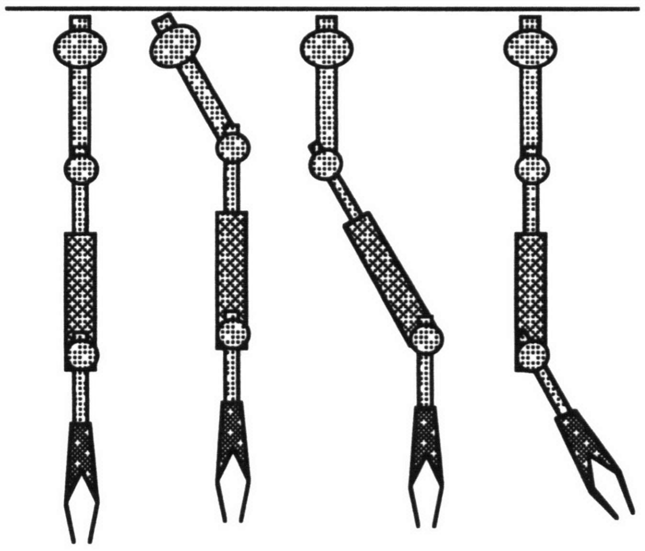

Figure 22. Each joint must lift the rest of the limb

A joint can lift the rest of the limb through the given angle if the joint's applicable torque > $l \bullet g \sin\theta \bullet \sum_{i=j}^{k} m_i$, where l is the effective length of the joint, g is gravity, j and k are the indices of the next and the last module in the limb, and m is the mass of the modules.

Also, all the limbs together must be capable of lifting the entire weight of the robot. This is determined by finding the greatest single-joint lifting force of each limb, with all other joints locked, and adding each of these limb forces together.

Grubler's mobility criterion must be met, insuring that every end-effector is capable of at least one controllable environmental force interaction [82]. n=3x(m-1)-2x f, where n is the number of degrees of freedom of the system, m is the number of rigid modules, and f is the number of joints.

Assembly Rules

Limbs may be rearranged only in statically stable conditions, as determined by straight-leg foot placement and center of mass. The center of mass must reside within the polygon formed by the foot locations. Also, geometric interferences between moving modules must be checked, but this is not yet implemented.

From the 6,689 kits remaining, 68,113 valid assemblies were constructed according to these subassembly and assembly rules.

Assembly Filters

Assemblies are filtered on task-based constraints.

First, the limbs with grippers are checked to insure they can provide enough lifting force to lift the required loose obstacles. The torque of the strongest joint in the limb is compared with the moment arm of the remaining limb modules and the required gripper weight for this test.

Second, assemblies are checked to see if they can be manipulated to fit within the maximum size constraints of the task. This has not yet been implemented.

These filters eliminated most (83%) of the assemblies from consideration, leaving only 11,840 assemblies to be evaluated.

Assembly Evaluation

The assembly evaluation is again a weighted sum of linear or asymptotic functions of component tests. Many of the component tests were previously used as filters, but now numerical results instead of pass/fail results are used for a measure of the quality of the result.

Assembly_Eval = Kit_Eval + 4x asymptotic DOF + 6x asymptotic Arm_Strength + 20x asymptotic Leg_Strength – 7x asymptotic Leg_Deviation

The assembly score builds on the kit score calculated earlier, because the kit features are applicable to its assemblies.

The number of degrees of freedom is considered a positive asymptotic factor.

Arms are limbs that end in a gripper, and arm strength is taken to be the lifting ability of the arm divided by the required gripper payload:

$$\text{Arm\_Strength} = \frac{\max_j \left(\dfrac{\tau_j}{\sum_{i=j}^{k} l_i} \right)}{P}$$

, where τ is the torque of joint j, l is the effective length of each module from the joint j to the end of the arm k, and P is the task-required gripper payload. The numerator of this term represents the largest lifting force available at the end of the arm due to a single joint.

A leg is a limb that does not end in a gripper. Leg strength is the sum of the lifting abilities of each leg divided by total robot mass:

$$\text{Leg\_Strength} = \frac{\sum \max_j \left(\dfrac{\tau_j}{\sum_{i=j}^{k} l_i} \right)}{\sum_{i=1}^{n} m_i}$$

, with the lifting ability as above. This rewards lean and "well-muscled" assemblies.

The deviation of the limb lengths is negatively weighted, discouraging widely varying limb lengths on a single assembly, which leads to uneven utilization.

Leg_Deviation = $\sum_i \left(l_i - l_{avg}\right)^2$, with l as the length of each leg i.

Robot speed, bandwidth, and accuracy tests should also be included but have not yet been implemented. A full dynamic simulation could also be added at this point in order to provide the most accurate evaluations.

Rank Assemblies

Based on the assembly evaluation described above, the best assembly for each kit is reported. The best few assemblies of all kits are remembered and reported at the end of the search.

Chapter 7: Results

Solution of the Constitution Task

Figure 23 shows the final output of the modular design algorithm for the Constitution inspection task. The best 10 solutions are shown with the highest scoring robot in the lower left corner. The assembly evaluation score is shown above each robot.

The total time taken for the search (kits of up to 17 modules) was 261.14 seconds on a Sun 4 workstation.

The top assemblies found had assembly evaluation functions from 40.7934 to 41.7653. The range of the top scoring robots is small because of the large number of evaluated robots. These ten represent the best of 11,840 assemblies, or the top 0.08%.

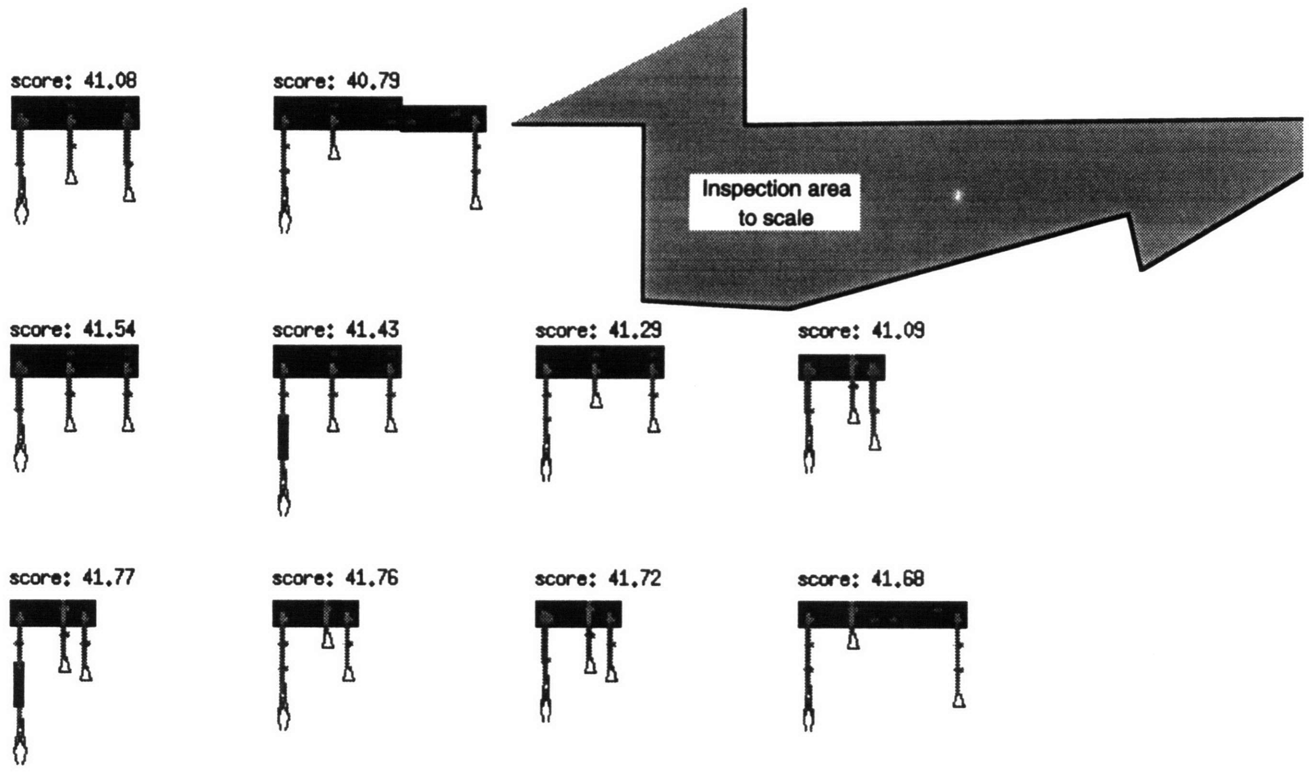

Figure 23. Design solutions for the task and inventory

These robots can all fit through the narrowest opening, the entranceway to the inspection area, see Figure 23. Also, the legs are long enough to reach from the top of the large step to the bottom of the inspection area. This results in somewhat "gangly" robots, with long arms and small bodies.

These robots all have a somewhat atrophied third leg near the center; this results from the given task requirement of two feet and one gripper. The center leg does not help the robot's span, and so it can remain short. Two feet were specified in the task under the assumption that the robot body should remain off the ground during object manipulations with the gripper, but the evaluation tests did not explicitly check for static stability without the gripper arm. Clearly, this needs to be done for a more useful robot.

Effects of Changing the Evaluation Function on Quality of Solutions

Changing the assembly evaluation function directly changes which assemblies are considered best for the task. In this manner, the appropriateness of the original evaluation function can be established.

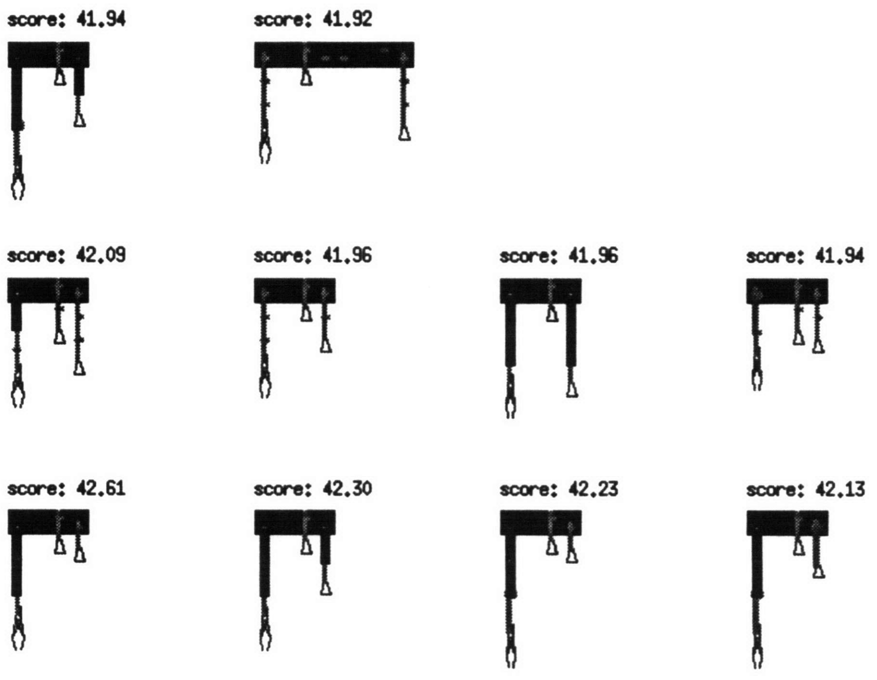

Figure 24. A different evaluation function

In Figure 24, the penalty for widely varying limb lengths was removed (Leg_Deviation), along with a rule forbidding links to be directly joined to bodies. This results in more "lopsided" robots. Robots with unequal length limbs and with directly attached links are not as dexterous as those in the original solution (Figure 23.) In Figure 25, the cost, mass, and operating time of the robot were all made less important ($w_{cost} = 5$, $w_{mass} = 10$, $w_{op\_time} = 0$.) Note that the robots still must meet all the constraints imposed by the task.

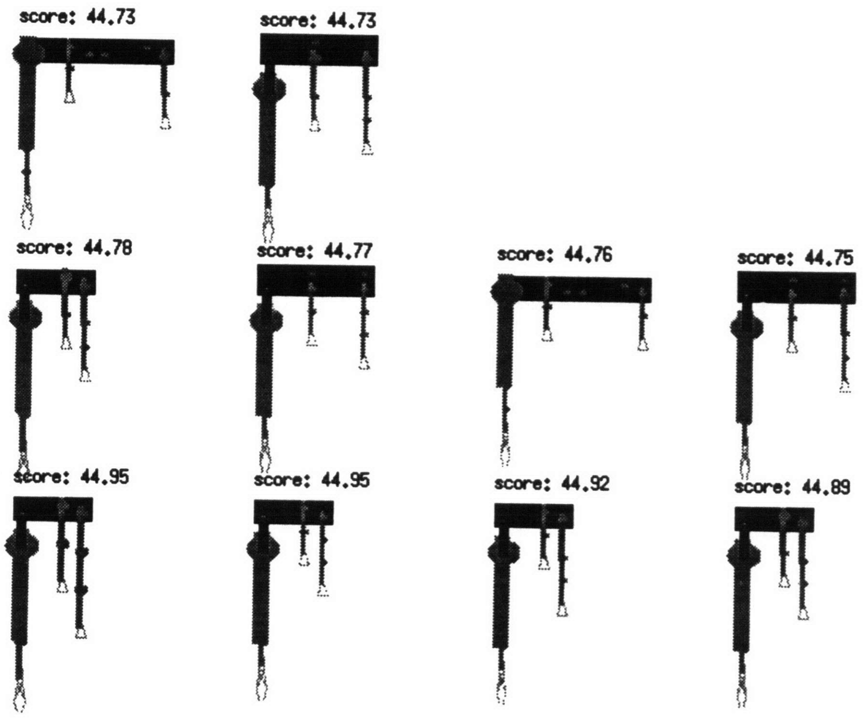

Figure 25. Another evaluation function

The resulting robot assemblies now have bigger, more expensive, and less efficient joints. These robots also have stronger limbs and can apply more torque than the original solutions. This demonstrates the physical results of changing objective function weights. These robots are not necessarily better or worse than those in Figure 23, but they do have different task performance characteristics. Performance tradeoffs such as the relative merits of strength versus operating time are dependent on the particular task at hand. Therefore, it makes sense to include the relative numerical weightings for factors like cost, operating time, and reliability as part of the task description.

Lowering the weights of cost and operating time allows latent characteristics like strength to increasingly manifest themselves in the solutions. For the Constitution task in particular, strength is not as important as operating time, and so weights were chosen accordingly, with the results shown in Figure 23.

Validity of Solutions

Because the algorithm naturally ranks robot assemblies, the effectiveness of the modular search technique can be demonstrated by examining which assemblies are ranked highly and which assemblies are ranked poorly for the given task.

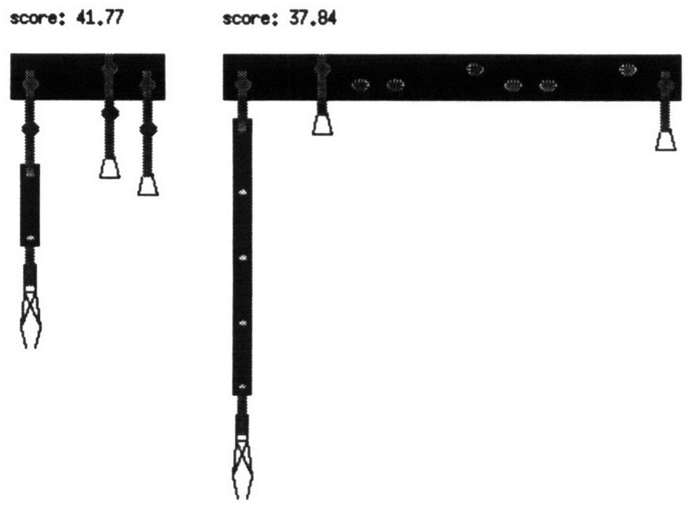

Figure 26. A higher ranked robot and a lower ranked robot

In Figure 26, two robots ranked by the automatic design method are shown. The robot on the right weighs twice as much as the robot on the left because of all its power modules. It also has a greater operating time (more power modules and fewer actuated modules.) The robot on the right also costs less ($381 as compared to $497.) The robot on the left scores better in terms of climbing ability, ability to maneuver through small openings, dexterity, and walking ability. It also meets the operating time and cost constraints.

Overall, the robot on the left has a higher evaluation score (41.77) than the robot on the right (37.84), even though both robots are still in the upper 2% of all assemblies. The robot on the left makes more physical sense for the Constitution task, which requires good climbing ability and the ability to maneuver through small spaces, and not an especially long operating time. Therefore, in this case the evaluation function has achieved its goal of ranking robots relatively according to task performance - the robot that is better for the task has a higher evaluation score then the robot that is not as good.

Growth of Search Space

An important result of this research is that the search space is physically limited. Although the number of possible combinations of modules grows exponentially with the number of modules in the kit, the number of filtered kits remains finite.

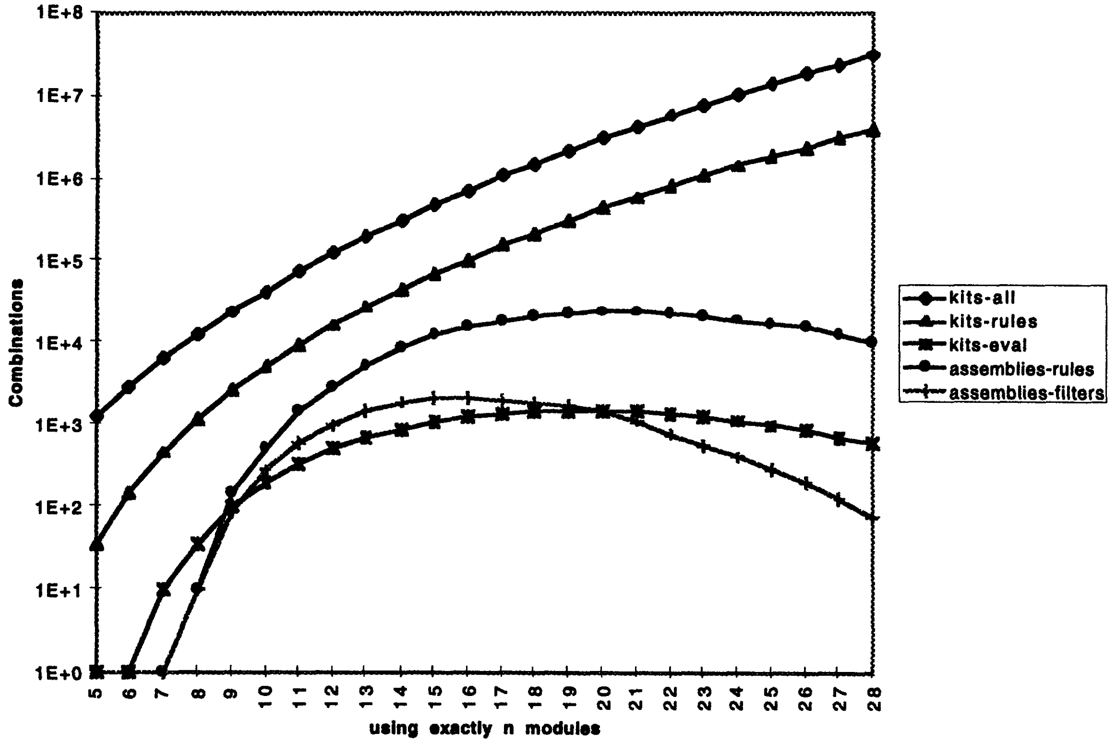

Figure 27. Combinatorial growth of search size with number of modules

In Figure 27, the first data series represents the number of possible kits logarithmically plotted against the number of modules in the robot. This factorial function (page 14) grows without bounds. The next series, kits-rules, is a constant fraction of the possible kits, limited by the task-independent kit selection rules, see Figure 9. The kits that pass the kit filters and meet the kit assembly threshold are shown as kits-eval. Note that this number starts to decrease after the number of modules in the kit exceeds some limit, in this case about 20 modules. The reason for this decrease is that task-based and structural limits are being approached, in areas such as total weight and total cost. The number of good kits returns to 0 again at 32 modules, at which point any possible combination of 32 modules costs more than the total cost limit as defined in the task. The number of assemblies made from the kits are likewise limited - as the number of kits decreases, the total number of assemblies also decreases. The valid assemblies, those that pass the assembly filters, are further limited by structural constraints, and their numbers decrease even more rapidly than the valid kits. Less than 100 valid assemblies are possible from kits made of 28 modules.

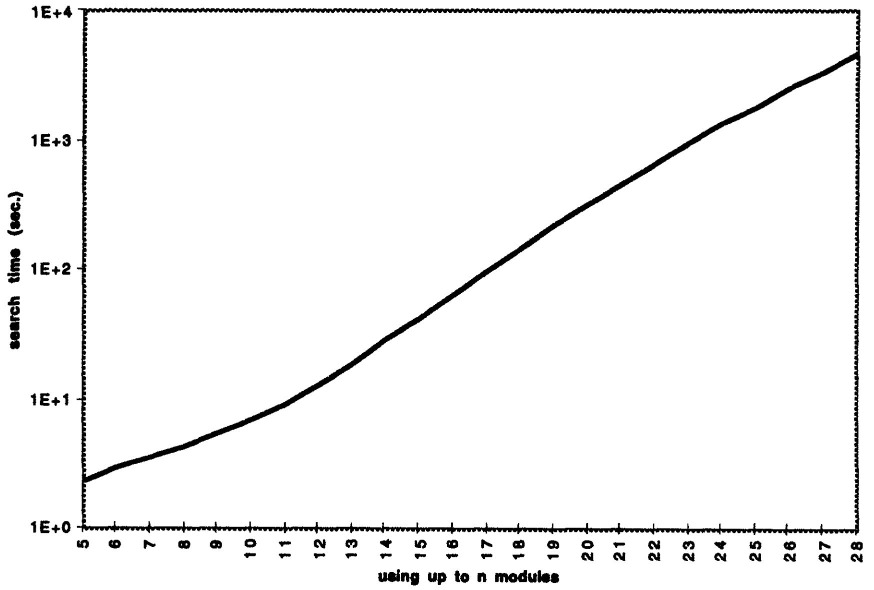

Figure 28. Search time versus number of kit size

Although the number of all possible kits grows without bounds, only those few thousand assemblies that pass the assembly filters are fully evaluated, resulting in a reasonable execution time for the fully explored search, see Figure 28.

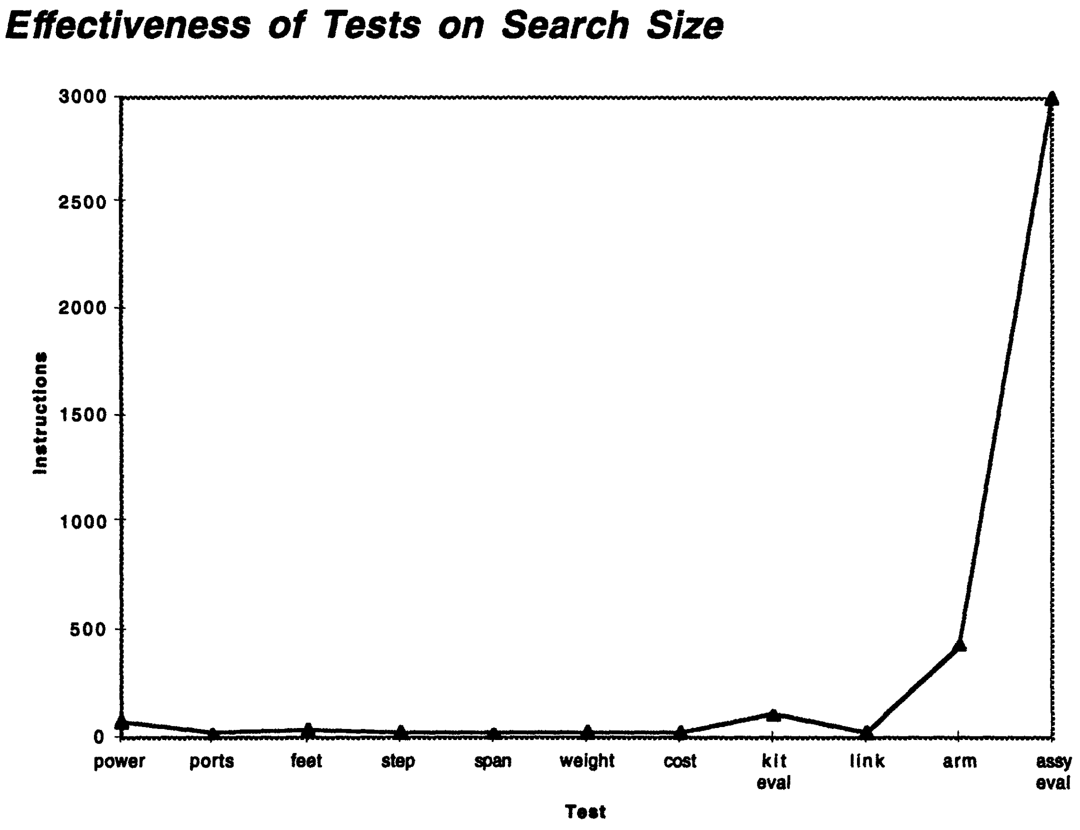

Figure 29. Computations for tests

Computationally cheap tests that are able to distinguish between good and bad structures early on in the search process save the computation of later tests. Figure 29 shows the number of computations required for a single instance of some tests. The tests on the left of the figure occur earlier in the search structure than the tests on the right. The computational cost (given here as an estimated number of instructions) is seen to generally increase as the robot becomes built up into more complex structures, see Figure 13.

Kit evaluation and assembly evaluation are included in this figure to show their high computational cost. Filters are implemented in order to avoid precisely these costs: early, simple computations avoid later, more intensive computations.

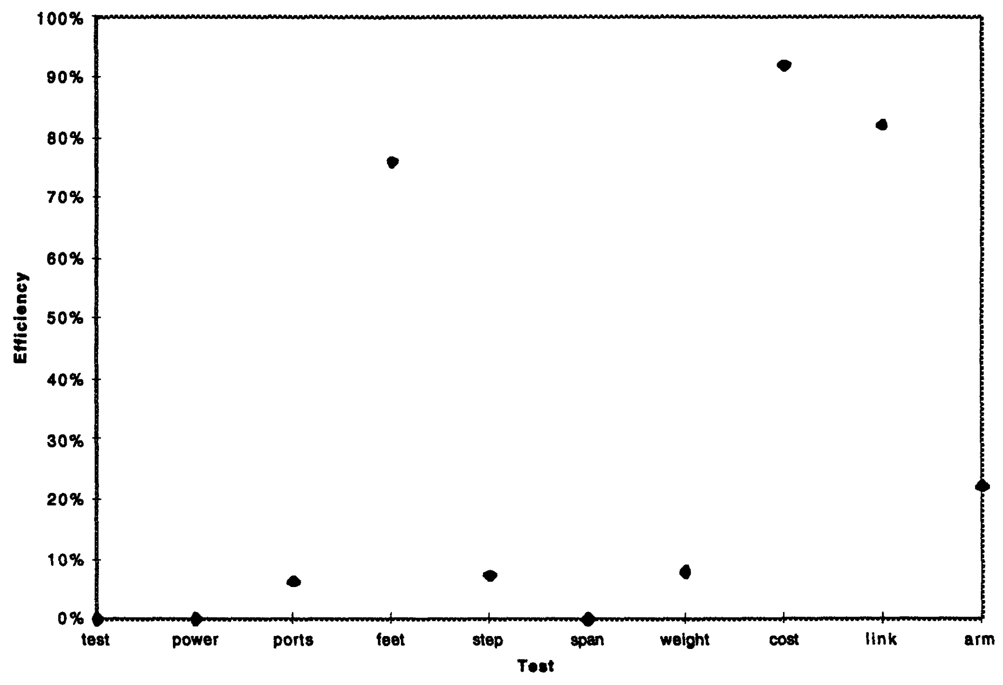

Figure 30. Filter efficiency

The culling efficiency of a filter, Figure 30, is the frequency with which it eliminates a structure from further consideration in the search process. Simple tests that are efficient at culling, applied early in the hierarchy, save many calculations per robot over many robots. As can be seen on the figure, several of the simple tests implemented do have good culling efficiencies for this task. Note that the efficiency of a filter may vary depending or the order in which filters are arranged.

Figure 31 shows the effects of some tests on the total calculations required by the search process. The number of instructions executed during the search is the product of the instructions per test and the number of times that test was executed. The tests are listed in the order they occur during the search, from left to right. Because the first tests eliminate some candidate robots, they are executed more frequently than later tests. The number of total executed instructions therefore falls during the search, until the increasing cost of each test (see Figure 29) drives it back up again.

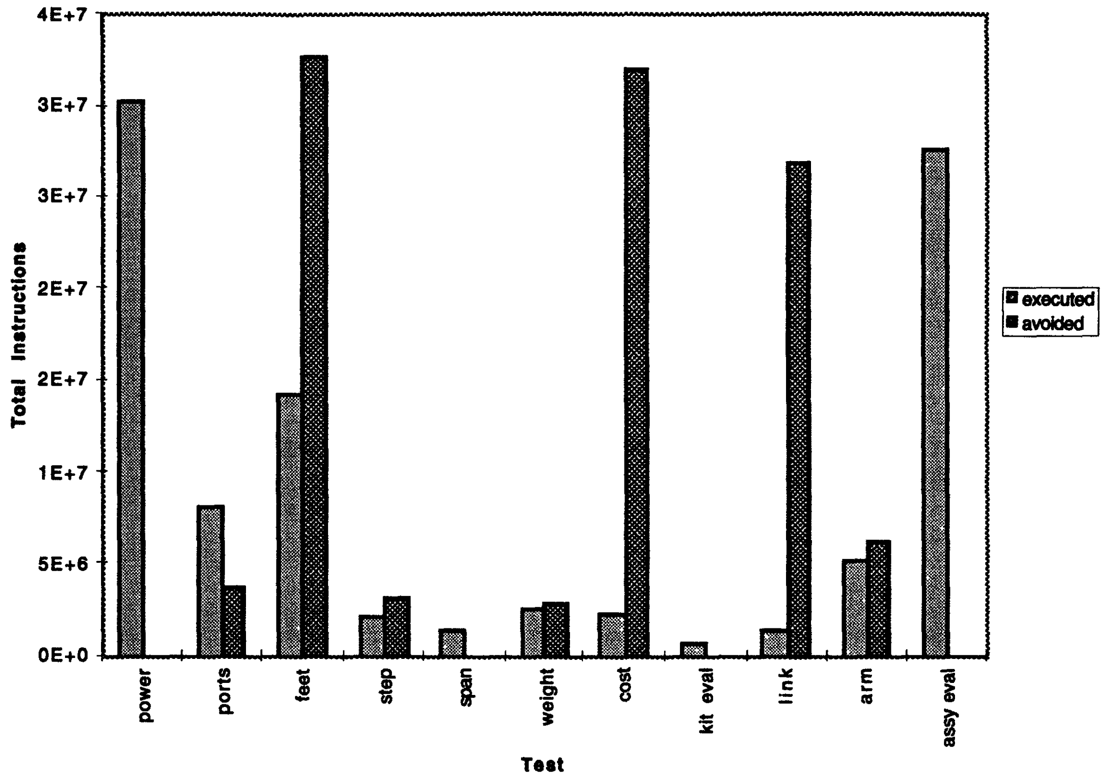

Figure 31. Filter effects on search computations

The number of instructions "avoided" by a particular test are those instructions that would have been executed during the search if that test had not been implemented. Earlier, more efficient tests avoid more instructions. If the number avoided is greater than the number executed, then the test is computationally worthwhile. (The test may still be required, even if it is not computationally worthwhile, to produce useful robots.) Most of the tests implemented are computationally worthwhile. The power test happened not to filter any kits because the module filters had already eliminated the possibility of incorrect power sources in this particular instance. The span constraint for this task just happened never to be violated. The kit and assembly evaluations are not filters and therefore do not avoid any instructions themselves.

If the final assembly evaluation becomes more computationally intensive, the number of instructions avoided by the tests increase. The computational efficiency of the entire hierarchical search process increases accordingly. A very expensive final assembly evaluation such as a full dynamic simulation makes even expensive filters computationally worthwhile.

Chapter 8: Comparison with Traditional Design Methods

Traditional Design of Constitution Robot

It is helpful to compare the modular design methodology with traditional design methods to help illustrate its advantages. Approaching the same Constitution task from a traditional standpoint, the following is a representative design process.

The task is first broken down into a series of requirements and constraints: size limits, self-containment, mobile walking robotic solutions, ability to climb over diagonal riders, and the other constraints listed in the task description above. In a general sense these are used to determine the realm of the design solution. The small dimensions, self-containment requirements, and lack of high loads coupled with some experience suggest an electric powered robot, as opposed to a hydraulic one. Critical design issues appear to be the ability to walk and to climb over obstacles. Keeping these in mind, typically many alternative designs are quickly proposed.

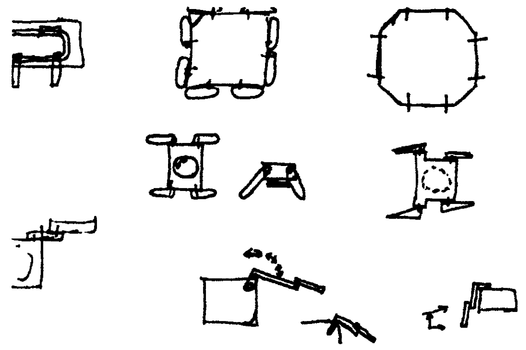

Figure 32. Alternative rough design sketches

A roughly dimensioned, general first cut of the most promising design is taken to be the baseline. This first-cut design would not include many of the design details, such as the kind of feet to be used for gripping damp wood. No quantitative data are yet known about the exact dimensions or the availability of any of the components.

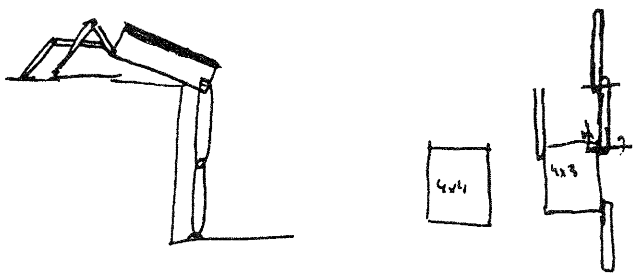

Figure 33. Most promising design is roughly dimensioned

The design is then modified to meet more general task requirements: walking, turning, sensing. Joints and legs are designed in more detail to be kinematically correct and geometrically interference-free. The ability to turn is added, and sensors and commun'cation equipment are located on the body. Dimensions are defined or refined.

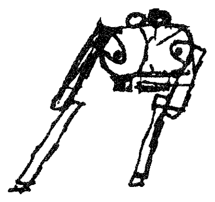

Figure 34. Refined overall robot design is sketched

Next, detailed mechanisms are fleshed out. Exact gearing and drive mechanisms are now included in the leg design, and the feet are designed in detail to be able to cope with a wet, wooden environment. Availability of subassemblies begins to become significant. Some part shapes may be qualitatively optimized for stress, and some rough analyses are performed. At this point the practicality of the design can more or less be determined, although many of the details and the availability of motors, batteries, and transmissions may remain unknown.

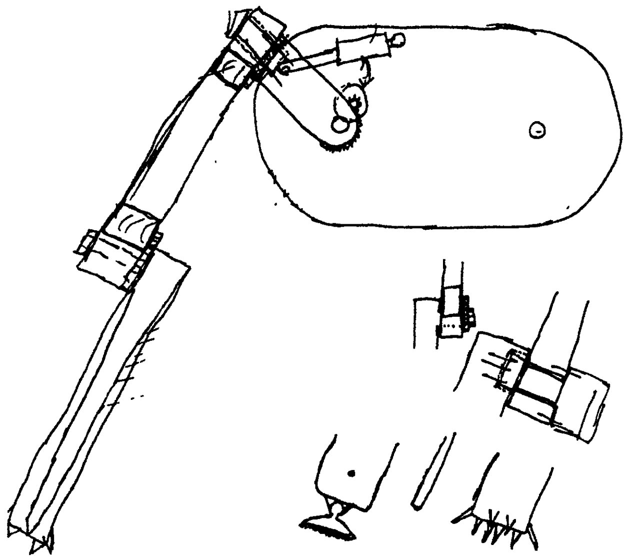

Figure 35. Detailed design sketches

At this point the design might be put aside for the moment in order to pursue some of the alternative designs suggested initially, or completely new concepts perhaps based on ideas that occurred during the earlier stages of the design. Because of the experience gained detailing the first design, the important design factors have been identified. Designs that appear to address these factors well are put through the detailing steps above. Those that cannot be achieved structurally and those that fail to meet a significant portion of the task requirements are abandoned.

The most promising of the remaining designs is then analyzed for sizing and torque requirements. Rough weight estimates and dimensions allow gross calculations of the torques for the climbing subtask. These torques in turn yield motor specifications, which are then checked against motor and gearing catalogues for availability. The available motor and gearhead lead to direct calculations of the walking speed and power requirements of the robot. The power requirements determine the battery weight, which is checked against the original weight estimates. If they do not agree, the weight estimates are revised and the motor is resized in an iterative fashion. When the estimate is accurate enough, the speed of the robot is calculated, and this information in turn is compared to the original task requirements (speed and weight requirements.)

The last few steps may have to be iterated to produce an acceptable design. If no acceptable design is forthcoming, iteration is pushed farther and farther back down the design path until an acceptable variation is produced or the design is scrapped entirely in favor of a more promising concept.

Comparison with Modular Design

Many parts of the traditional design method have analogs in the modular design method. The task is similarly decomposed into constraints and requirements. The module dimensional filters eliminate the larger hydraulic modules automatically in this case, with the same resulting realm of electric power. Like the traditional design methodology, the modular method uses increasingly accurate analysis through the analysis of a single robot, in order to minimize computational effort. The practicality of a particular design is determined finally by a complete suite of analyses on a fully specified design.

The major difference is that the human in the traditional design process already has an idea in mind for a robot that might work based on his experience, while the modular system has no such foreknowledge. It is characteristic of traditional design methods that evolutionary changes are made in small steps to the baseline design whenever possible. Major changes are avoided traditionally because of the time and effort involved in reanalyzing a new design. Minor revisions and refinements are acceptable so that previous analysis will be mostly still applicable. The design processes are illustrated conceptually in Figure 36 (upper figure from [83].) In the upper, traditional design method, task information is compiled and synthesized with designer experience. Analysis of the design then iteratively is fed back as information to be included in the next synthesis. Eventually this produces an acceptable design.

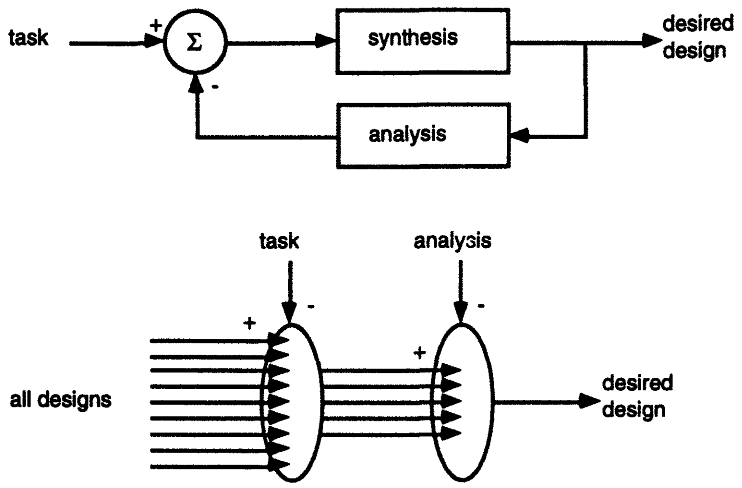

Figure 36. Design process comparison

In the lower portion of the figure, the modular design method is illustrated. All possible designs are filtered through the task requirements. Remaining designs are further analyzed and evaluated, producing a single optimal design. This method is linear and parallel, as compared to the iterative traditional process.

The differences between the traditional and the modular design methods are illustrated graphically in Figure 37 (this figure is for illustration purposes only and does not contain quantitative information.) The traditional design method is shown to produce successively improving results over a relatively long time, while the modular design method instead rapidly tests many unrelated solutions.

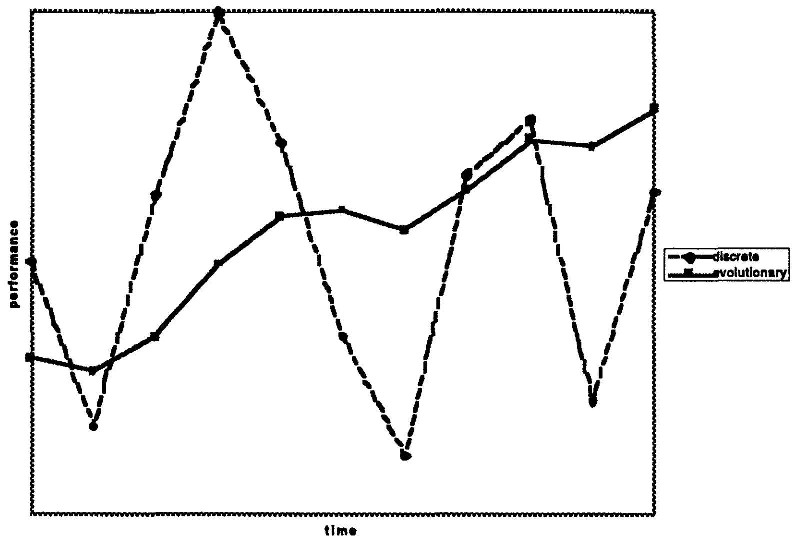

Figure 37. Modular versus traditional design methods

Executing the traditional design process, it becomes clear that the first few steps - the rough conceptual designs - are very rapidly and widely explored. But the later steps, such as detailed parts dimensioning, drawing, analysis, and component lookup, are extremely slow and burdensome to do in the traditional manner.

Comparatively, the modular design method can analyze fully designed robots very quickly (under 2 milliseconds). This is possible because the modules already have the full level of detail built in, and part availability has already been assured. Each component is known in detail in advance, and combining them is relatively straightforward.

However, the algorithm cannot recognize the potential of a robot that is "close" to being acceptable, and it does not know how to evolve that robot into one that is acceptable. For these two reasons, a genetic algorithm based search, as mentioned above, might be a good candidate for improved search performance.

Chapter 9: Conclusions

A framework has been established to permit the automatic generation and testing of modular robotic solutions for field environments based on specific task information, utilizing a hierarchical search technique.

Simple tests are able to discriminate between viable and infeasible solutions. Thus, the hierarchical search method is very efficient at trimming the design space down to a very manageable size. There is a fundamental tradeoff, however, between the variety and the quantity of solutions. The more tests that are included in the hierarchy, the smaller the solution space becomes, limiting both variety and quantity. In a broader sense, the dilemma is that many rules limit the designs to one possible set of solutions, while in fact other, unanticipated solutions may exist.

This hierarchical algorithm has been based on an exhaustive enumeration of all possible assemblies, which was then quickly pruned to a reasonable size. Other search schemes might be used instead of exhaustive enumeration, while still using the hierarchical rules, filters, and evaluations developed here. Specifically, branch and bound or genetic algorithms might be combined with this rule-based pruning technique for a faster, or alternately a broader, search. It is felt that without the rule-based pruning technique, results achieved by these other search methods alone will be very slowly forthcoming.

Direction of future work

With significant refinement and expansion of the rules, filters, and evaluations, the robots solutions' abilities and applicabilities to the task will continue to improve. Aside from the many still-needed refinements to the algorithm and better and more thorough tests, there are two specific areas that still need to be explored.

Expanding task and solution domains

This work, although developed with a particular task in mind, was purposefully kept as broad as possible so as to cover a wide range of robot applications. Expanded task and solution domains should eventually be included for a more general robot designer. Additional modules can be included easily as part of a datafile. Rules, filters, and evaluations can also be included at the cost of (generally) a few lines of code. Furthermore, additional characteristics of the modules or the task can be incorporated easily by modifying the appropriate class. Walking, multi-limbed robots might be augmented with snake-like, swimming, or climbing abilities.

Adding planning and control

The issue of motion planning and control has been avoided by using a fixed set of predefined actions for all tasks. Although additional actions can be included explicitly to increase the solution domain, an automatic approach would be more useful. Synthesis of automatic design, planning and control will allow the feedback of control and planning issues into the structural design, producing more diverse and effective robots.

Bibliography

1 Anderson, M. "Ecological robots," *Technology Review*, Jan. 1992. vol. 95, no. 1, pp. 22-3.

2 Goldsmith, S. "It's a Dirty Job, but Something's Gotta Do It," *Business Week*, Aug. 20, 1990. pp. 92-7.

3 Stone, H., and Edmonds, G. "Hazbot: A Hazardous Materials Emergency Response Mobile Robot," *Proceedings 1992 IEEE Rob. Automation*, Nice France, 1992. vol. 1, pp. 67-73.

4 Weisman, R. "GRI's Internal Inspection System for Piping Networks," *Proceedings, 40th Conference on Remote Systems Technology*, 1992. vol. 2, pp. 109-15.

5 Wehe, D.K., et al. "Intelligent Robotics and Remote Systems for the Nuclear Industry," *Nuclear Engineering and Design*, 1989. vol. 113, no. 2, pp. 259-67.

6 Akizono et al. "Field Test of an Aquatic Walking Robot for Underwater Inspection," *Mechatronic System Engineering*, 1990. vol. 1, no. 3, pp. 233-9.

7 Madhani, A. and Dubowsky, S. "Design and Motion Planning of Multi-Limb Robotic Systems: The Force Workspace Approach," *Proceedings of the 1992 ASME Mechanisms Conference*, Scottsdale, AZ, Sept. 13-16, 1992.

8 Song, S., and Waldron, K. Machines That Walk: The Adaptive Suspension Vehicle. MIT Press, Cambridge, MA, 1989.

9 Hirose, S. Biologically Inspired Robots: Snake like Locomotion and Manipulation. Oxford Scientific Publications, 1993.

10 Moore, C. "On the Design of an Untethered Climbing Robotic System," Master's Thesis, Department of Mechanical Engineering, MIT, Cambridge, MA, 1993.

11 Cole, J. "Automatic Planning for Modular Robots," Master's Thesis, Department of Mechanical Engineering, MIT, Cambridge, MA, 1995.

12 "NIST cranks up an incredible crane," *Science News*, June 20, 1992. vol. 141, no. 25, pp. 415.

13 Luk, B., Collie, A., and Billingsley, J. "ROBUG II: An Intelligent Wall Climbing Robot," *IEEE Proceedings*, 1991. pp. 2342-7.

14 Stix, G. "No tipping please," *Scientific American*, Jan. 1992. vol. 266, no. 1, pp. 141.

15 Engelberger, J. Robots in Service. Biddler, Ltd. Great Britain, 1989.

16 Hoffman, T. "Hospital Robots Have the Rx for Efficiency," *Computerworld*, Jan. 11, 1993. vol. 27, no. 2, pp. 69-70.

17 Glaskin, M. "Robot Jobsworths Go on Patrol," *New Scientist*, Jan. 29, 1994. vol. 141, no. 1910, pp. 19.

18 Bains, S. "Robot Cleaners Spit and Polish as they Go," *New Scientist*, Nov. 27, 1993. vol. 140, pp. 20.

19 Pope, G. "Homer Hoover," *Discover*, Mar. 1993. vol. 14, no. 3, pp. 28.

20 Concar, D. "Can Robots Come to Care for Us?" *New Scientist*, Oct. 2, 1993. vol. 140, no. 1893, pp. 40-2.

21 Port, O. "Grapevines Need a Gentle Trim; Call Robo-Pruner," *Business Week*, July 05, 1993. no. 3326, pp. 98.

22 Young, P. "Kinked Cable Crimps Dante's Erebus Debut," *Science News*, Jan. 9, 1993. vol. 143, no. 2, pp. 22.

23 Andeen, G. ed. Robot Design Handbook. McGraw Hill, New York, NY, 1988.

24 Cutkosky, M., and Wright, P. "Friction, Stability and the Design of Robotic Fingers," *International Journal of Robotics Research*, 1986. vol. 5, no. 4, pp. 20-37.

25 Kanade, T. "Very Fast 3- D Sensing Hardware," *Proceedings of the International Symposium on Robotics Research*, October 2-5, 1993. Hidden Valley, PA.

26 Hollis, R., Salcudean, S., and Allan, A., "A Six Degree-of-Freedom Magnetically Levitated Variable Compliance Fine Motion Wrist: Design, Modeling and Control," *IEEE Transactions on Robotics and Automation*, June 1991. vol. 7, pp. 320-32.

27 Lawrence, D. "Designing Teleoperator Architecture for Transparency," *Proceedings of the IEEE International Conference on Robotics and Automation*, Nice, France, May 1992. pp. 1406-11.

28 Flynn, A., et al. "Piezoelectric Micromotors for Microrobots," *Journal of Microelectromechanical Systems*, 1992. vol. 1, no. 1, pp. 44-51.

29 Anathusaresh, G. et al. "Design and Fabrication of Microelectromechanical Systems," *Robotics, Spatial Mechanisms and Mechanical Systems*, ASME, 1992. DE-vol. 45, pp. 251-8.

30 Lee, P. and Pisano, P. "Polysilicon Angular Microvibromotors," *Journal of Microelectromechanical Systems*, 1992. vol. 1, no. 2, pp. 70-6.

31 Ragulskis, K. et al. Vibromotors for Precision Microrobots. Hemisphere Publishing Corporation, 1988.

32 Shahinpoor, M. "Conceptual design, Kinematics and Dynamics of Swimming Robotic Structures Using a Ionic Polymeric Gel Muscles," *Smart Materials Structures*, 1992. vol. 1, no. 1, pp. 91-4.

33 Byrd J., and De Vries, K. "A Six-Legged Telerobot for Nuclear Applications Development," *International Journal of Robotics Research*, 1990. vol. 9, no. 2, pp. 43-52.

34 Furusho, J. and Sano, A. "Sensor-Based Control of a Nine-Link Biped," *International Journal of Robotics Research*, 1990. vol. 9, no. 2, pp. 83-98.

35 Hirose, S. et al. "Design of Prismatic Quadruped Walking Vehicle Titan VI," *Proceedings of the 5th ICAR*, 1991. pp. 723-8.

36 Krotkov, E. and Simmons, R. "Performance of a Six-Legged Planetary Rover: Power, Positioning, and Autonomous Walking," *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, Nice France, 1992. vol. 1, pp. 169-74.

37 Argaez, D. "An Analytical and Experimental Study of the Simultaneous Control of Motion and Force of a Climbing Robot," Master's Thesis, Department of Mechanical Engineering, MIT, Cambridge, MA, May 1993.

38 Chernusko, F. "Mechanics of Climbing Robot," *Mechatronics Systems Engineering*, 1990. vol. 1, no. 3, pp. 219-24.

39 Gradetsky, V. and Rachkov, M. "Wall Climbing Robot and its Application for Building Construction," *Mechatronics System Engineering,* 1990. vol. 1, no. 3, pp. 225-31.

40 Abarinov, A. et al. "Robot System for Moving over Vertical Surfaces," *Soviet Journal of Computer Systems,* 1989. vol. 27, no. 3, pp. 130-42.

41 Collie et al. "Design and Performance of the Portsmouth Climbing Robot," *Mechatronics System Engineering,* 1990. vol. 1, no. 2, pp. 139-47.

42 Stulce et al. "Conceptual Design of a Multibody Passive Legged Crawling Vehicle," *ASME,* 1990. DE-vol. 26, pp. 199-205.

43 Iagolnitzer, M. et al. "Locomotion of an All-Terrain Mobile Robot," *Proceedings of the 1992 IEEE International Conference on Robotics and Automation,* Nice France, 1992. vol. 1, pp. 104-9.

44 Kemurdjian, A., et al. "Small Marsokhod Configuration," *Proceedings of the 1992 IEEE International Conference on Robotics and Automation,* Nice France, 1992. vol. 1, pp. 165-8.

45 Pugh, D. et al. "Technical Description of the Adaptive Suspension Vehicle," *International Journal of Robotics Research,* 1990. vol. 9, no. 2, pp. 24-42.

46 Raibert, M. Legged Robots That Balance. MIT Press, Cambridge, MA, 1986.

47 Jones, J. and Flynn, A. Mobile Robots: Inspiration to Implementation. A.K. Peters, MA, 1993.

48 Clement, W. and Inigo, R. "Design of a Snake-Like Manipulator," *Robotics and Autonomous Systems,* 1990. vol. 6, pp. 265-82.

49 Fichter E., and Kerr D. "Walking Machine Design Based on Certain Aspects of Insect Leg Design," *Robotics, Spatial Mechanisms and Mechanical Systems,* ASME, 1992. DE-vol. 45, pp. 561-6.

50 Kubo, Y., Shimoyama, I., and Miura, H. "Study of Inspect-Based Flying Microrobots," *Proceedings IEEE International Conference on Robotics and Automation,* Atlanta, GA, May 1993.

51 Dubowsky, S., Moore, C. and Sunada, C., "The Power Map Method for Designing Field Robotic Systems," *ANS 6th Topical Meeting on Robotics and Remote Systems,* 1994.

52 Chirikian, G., and Burdick, J. "Design, Implementation and Experiments with a Thirty Degree-of-Freedom 'Hyper-Redundant' Robot," *Proceedings of the IEEE International Conference on Robotics and Automation,* Atlanta, GA, May 1993.

53 Tesar, D., and Butler, M. "A Generalized Modular Architecture for Robot Structures," *Manufacturing Review,* 1989. vol. 2, no. 2.

54 Cohen, R., Lipton, M., Dai, M., and Benhabib, B. "Conceptual Design of a Modular Robot." *Journal of Mechanical Design,* Mar 1992. vol. 114, pp. 117-25.

55 Benhabib, B., Zak, G. and Lipton, M., "A Generalized Kinematic Modeling Method for Modular Robots," *Journal of Robotics Systems,* 1989. vol. 6, no. 5, pp. 545-71.

56 Kelmar, L., and Khosla, P. "Automatic Generation of Forward and Inverse Kinematics for a Reconfigurable Modular Manipulator System," *Journal of Robotics Systems,* 1990. vol. 7, no. 4, pp. 599-619.

57 Chirikian, G., and Burdick, J., "Parallel Formulation of the Inverse Kinematics of Modular HyperRedundant Manipulators," *Proceedings of the IEEE International Conference on Robotics and Automation*, Sacramento, CA, April, 1991.

58 Chen, I. and Burdick, J. "Enumerating the Non-Isomorphic Assembly Configurations of Modular Robotic Systems," *Proceedings of the 1993 IEEE International Conference on Intelligent Robotics and Systems*, Yokohama, Japan, July 26-30, 1993. pp. 1985-92.

59 Agrawal, V., Kohli, V., and Gupta, S. "Computer Aided Robot Selection: the Multiple Attribute Decision Making Approach," *International Journal of Production Research*, 1991. vol.29, no. 8, pp. 1629-44.

60 Gardone, B., and Ragade, R., "IREX: An Expert System for the Selection of Industrial Robots and its Implementation in Two Environments," *Proceedings of the Third International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Charleston, SC, July 15-18, 1990. vol.2, pp. 1086-95.

61 Paredis, C. and Khosla, P. "Synthesis Methodology for Task Based Reconfiguration of Modular Manipulator Systems," *Proceedings of the International Symposium on Robotics Research*, Hidden Valley, PA, Oct. 2-5, 1993.

62 Kim, J., and Khosla, P. "Design of Space Shuttle Tile Servicing Robot: an Application of Task Based Kinematic Design," *Proceedings of the IEEE International Conference on Robotics and Automation*, Atlanta, GA, May 1993. vol. 3, pp. 867-74.

63 Au, W., Paredis, C., and Khosla, P. "Kinematic Design of Fault Tolerant Manipulators," *Proceedings of the Allerton Conference*, Urbana-Champagne, IL, Oct. 2, 1992.

64 Hamel, W., Richardson, B. and Killough, S. "Evaluating Telerobotics for Battlefield Support Operations," *Proceedings of the Remote Systems and Robotics and Hostile Environments*, ANS Society, Paso, WA, Mar. 29-Apr. 2, 1987. pp. 284-9.

65 Dubowsky, S., and Vance, E. "Planning Mobile Manipulator Motions Considering Vehicle Dynamic Stability Constraints," *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, Scottsdale, AZ, May 14-19, 1989.

66 Sidney, R., Research Assistant in Mathematics, School of Science, MIT, Cambridge, MA. From a private conversation of September 2, 1994.

67 Shiller, Z., and Dubowsky, S. "On Computing the Global Time Optimal Motions of Robotic Manipulators in the Presence of Obstacles," *IEEE Journal of Robotics and Automation*, Dec. 1991. vol. 7, no. 6, pp. 785-97.

68 Beretta, R. Mechanism Analysis Pack: Kinematic and Dynamic Mechanism Analysis in Mathematica. Wolfram Research, Champagne, IL, 1995.

69 Corke, P. Robotics Toolbox for MATLAB. CSIRO, Preston, Austrailia, July 1994.

70 Grubel, G., Finsterwalder, R., Joos, H., Lewald, A., and Otter, M. "ANDECS: A Computation Environment for Robot-Dynamics Design Automation," *Proceedings of the IEEE International Conference on Robotics and Automation*, 1994. pp. 1088-93.

71 Muck, R., and Mammern, J. "Modular mechanical engineering - a revolution in engineering industry," *Proceedings of International Conference on Advanced Manufacturing*, 1984. pp. 271-82.

72 Yim, M. "New Locomotion Gaits," IEEE 1050-4729, 1994. pp. 2508-14.

73 Schmitz, D., Khosla, P., and Kanade, T. "The CMU Reconfigurable Modular Manipulator System," Carnegie Mellon University technical report CMU-RI-TR-88-7, May 1988.

74 Dario, P., Valleggi, R., Carrozza, M.C., Montesi, M.C., and Cocco, M. "Microactuators for Microrobots: a Critical Survey," IOP Publishing, 0960-1317/92/030131 + 17, 1992. pp. 141-57.

75 Rogers, C. "Intelligent Material Systems - the Dawn of a New Materials Age," *Journal of Intelligent Material Systems and Structures,* January 1993. vol. 4, pp. 4-12.

76 Elwenspoek, M. "Active Joints for Microrobot Limbs," *Journal of Micromechanics and Microengineering,* September 1992. vol. 2, no. 3, pp. 221-3.

77 Newnham, R., and Ruschau, G. "Electromechanical Properties of Smart Materials," *Journal of Intelligent Material Systems and Structures,* July 1993. vol. 4, pp. 289-94.

78 Weiss, K., Carlson, J., and Coulter, J. "Material Aspects of Electrorheological Systems," *Journal of Intelligent Material Systems and Structures,* January 1993. vol. 4, pp. 13-31.

79 Winston, P. Artificial Intelligence. Addison-Wesley, Reading, MA 1992.

80 Arora, J. Introduction to Optimum Design. McGraw-Hill, New York, 1989.

81 Goldberg, D. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, MA 1989.

82 Sunada, C. "Coordinated Jacobian Transpose Control and its Application to a Climbing Robot," Master's Thesis, Department of Mechanical Engineering, MIT, Cambridge, MA, 1994.

83 Suh, N. The Principles of Design. Oxford University Press, NY, 1990.

Appendix: modular design program

Usage

Most of the functions described in this thesis are implemented in the following C++ code. Some remain yet to be implemented. The code is organized into many interacting files with related procedures.

search_main.C is the main program. It filters the modules, sets up the initial test kit, performs the search, and prints out the results. It uses routines in three other files to perform the search. *choose_in.C* is used to decide which kit to form next in the search. *kit.C* contains the kit filters and evaluations. *assemble.C* assembles the kit into all the possible assemblies, filters and evaluates them, and returns the best one.

mod_desc.C describes the modules to be used in the search. Modules are defined by the file *mod.dat*, in the following form.

```
This file read by mod_desc.C, modules are defined in modclass.h, modclass.C
All units are MKS.

Id    Cost  ModT      EngyT OKEnvs   szX   szY   szZ   Wt    Reli  EngyUse
                                                       (power)     EnCap      #Port PortList
                                                       (link)      SprtTrq ArmLen
                                                       (joint)                    AplyTrq Speed AngMin
AngMax
                                                       (endef)                          EeFn
!
110   30.00 P     E   WD         0.12  0.04  0.04  0.36  0.96  0.002 13.4  5     0.01 0.01 0.112 0.01 0.057 0.01
0.058 0.03 0.112 0.03
112   17.00 P     E   WD         0.08  0.03  0.03  0.14  0.96  0.001 5.0   3     0.01 0.01 0.072 0.01 0.052 0.021
210   10.00 L     Z   WRVDT      0.01  0.055 0.012 0.01  0.99  0.0   361.0 0.043
310   175.0 J     E   WVDT       0.035 0.15  0.065 0.7   0.93  1.02  10.0  0.125 5.0   0.91  -1.57 1.57
314   80.00 J     E   WVDT       0.015 0.064 0.057 0.08  0.93  0.09  1.2   0.05  0.3   0.80  -1.57 1.57
318   60.00 J     E   WVDT       0.01  0.042 0.04  0.02  0.92  0.05  0.9   0.03  0.1   0.91  -1.57 1.57
410   40.00 E     E   WRD        0.01  0.055 0.055 0.03  0.89  0.06  6.45  0.043 3.0   0.1   G
430   150.0 E     H   RDT        0.03  0.10  0.10  0.5   0.88  5.0   30.0  0.09  30.0  0.5   HG
470   5.00  E     Z   WRVDT      0.01  0.02  0.015 0.005 0.99  0.0   126.0 0.014 0.0   0.0   H
```

The first column is a unique module identification number. Second is its cost in dollars. The third column determines which module classification it is: (P)ower, (L)ink, (J)oint, or (E)nd-effector. Fourth is the type of energy the module uses or supplies: (E)lectric, (H)ydraulic, (P)neumatic, or (Z) for none. The OKEnvs column is a list of environments that the module can work in: (W)et, (D)ark, (R)adioactive, etc. Next are dimensions, X is horizontal, Y is vertical, and Z is into the plane of the drawings in this thesis. Next is weight, then reliability, then energy usage rate (required power.) The following numbers are dependent on the module class. Power modules have an available energy capacity and a list of X,Y port positions. Links have a maximum torque they can withstand, and an active arm length (not including overlapping fastening areas.) Joints have a minimum backdriving torque, active arm length, maximum active torque they can apply, average loaded rotational speed, and minimum and maximum angle they operate through. End effectors have minimum backdriving force, active arm length, maximum active force they can apply, average loaded linear speed, and the type of end-effector it is.

The task as of now is simply hard-coded into the routine *mod_desc.C* also.

The files *modclass.C*, *groupclass.C*, and *assyclass.C* contain the C++ class constructs used in the search: modules (*modclass.C*), kits (*groupclass.C*), and subassemblies and assemblies (*assyclass.C*).

The remaining files are support programs. *mymath.h* includes some useful math routines, *stats.C* keeps track of statistics on the search, and *graphics.C* and *service.c* provide the XWindows graphics routines.

The *makefile* is set up for the Cygnus g++ compiler.

Output

In addition to the graphical output of the best robot assemblies, the program provides statistical data on the number of kits and assemblies produced during the search as well as statistics on the effectiveness of filters and evaluations and the amount of time taken. The program also periodically reports a description of which robot it happens to be analyzing along with its evaluation components numerical results.

Code

The programs are listed in a more or less hierarchical order, with the highest levels first.

search_main.h

```
/*
 Nathaniel Rutman 10/28/94

   search_main.h

 just lets search_main know about these functions, described
elsewhere

  1/24/95 new constants
*/

#include "modclass.h"
#include "assyclass.h"

#ifndef _SEARCH_MAIN_H_
#define _SEARCH_MAIN_H_


#define DUMPFRQ 5000000     /* frequency of screen reports */
#define ASSYTHRSHLD 1.0   /* assembly threshold.
                    Kit eval must be > to be assembled */
#define MAXLENGTH 17      /* max robot length.   MAXMODS is
unique mods */
#define MIN_FEET 2      /* minimum number of limbs */

#define TOPTEN 10       /* best n assemblies to remember */


void get_task(Task_Type*);
int reduce_space(Task_Type);
void keep_best(Assembly, Assembly best[TOPTEN]);
void set_dump(int, int);
float get_time();

#endif
```

search_main.C

```
/* Nathaniel Rutman  MS '95
  rutman@mit

   Search_main.C


  Main routine to generate and analyze modular robot assemblies.
   Reads in modules and task, steps through all kits and assemblies,
   reports best assembly and search statistics.

   change history:
  10/12/94  original NZR
  12/15/94  revised classes into Group, Assembly, etc NZR
  01/24/95  remember top assemblies NZR
*/



#include  "search_main.h"
#include "mod_desc.h"
#include "assemble.h"
#include  "kit.h"
#include "choose.h"
#include  "graphics.h"
#include  "stats.h"
#include <time.h>
#include <stdio.h>

Group fullgrp;       /* global group of all available mods */
bool datadump=TRUE;   /* global flag for reporting info on screen */
extern float MAGNIFY;  /* graphics magnification in graphics.c */
Stat_Type stats;       /* statistics on search */
```

```
int main(void) {
 get_time(); // CPU time in seconds
 MAGNIFY = 1200;
 draw_start();  // set up drawing canvas

 /* module set up */
  stats.initial_mods = get_module_desc();
  Task_Type task;
   get_task(&task);
  fullgrp.print();

 /* module filters */
 stats.reduced_mods = reduce_space(task);
  fullgrp.print();

 /* kit set up */
 Group test_kit;
 Assembly test_assy;
 Assembly best_assys[TOPTEN+1];
  int test_kit_ary[GRPS][MAXLENGTH];
       /* for ease of assembly, int array of mods in test_kit */
   if(!initialize_kit(test_kit_ary))  {
    cout << "Can't initialize kits\n";
   return 0;
 }

 cout << "Begin testing with ";
   print_test_kit(test_kit_ary);

 /* main search routine */
  float evl_test, evl_best=BAD_SCORE;
  while (choose_next(test_kit_ary)) {
   stats.cnt_kit++;
     convert_to_grp(test_kit_ary,&test_kit);
      if(kit_rules(&test_kit)  &&  kit_rules_task(&test_kit,&task))  {
     stats.cnt_kit_rules++;
    set_dump(stats.cnt_kit_rules, DUMPFRQ);
     evl_test = evaluate_kit(&test_kit, &task);
     evl_best = (evl_best>evl_test) ? evl_best:evl_test;
    if (datadump) {
     cout << "score " << evl_test << "  best kit so far " << evl_best;
     cout << "\n  time " << get_time() << " sec\n";
   }
    if (evl_test > ASSYTHRSHLD) {
     test_assy = assemble(&test_kit, &task);
     keep_best(test_assy, best_assys);
   }
  }
 }

 /* print out statistics */
 cout << endl;
  stats.print_all();
 cout << "Total time taken " << get_time() << " seconds.\n";
 cout << "Best kit found: fitness " << evl_best << endl;
cout << "Top assemblies found: fitness " << best_assys[TOPTEN-
1].score;
 cout << " ... " << best_assys[0].score << endl << endl;
 Point_Type origin;
 draw_clean();
 MAGNIFY = 600;
 for(int i=0; i<TOPTEN; i++) {
  //best_assys[i].print();
  //cout << endl;
  origin.x = 0.25*(i%4) + 0.02;
  origin.y = 0.3*(i/4) + 0.3;
   best_assys[i].draw(origin);
 }
 cout << "\n\nPress 'return' key to end.";
 char c;
 scanf("%c",&c);  // wait so graphics don't go away
}
```

```
int reduce_space(Task_Type task) {
 cout << "reducing module space\n";
 int i;
  fullgrp.reset_next();
 Mod_Base* curmod;
 bool do_remove;
  if(fullgrp.get_length() < 1) {
   cout << endl << "main: Failure: no objects in group\n";
   return 0;
 }

 /* generic task-based tests */
 while((curmod = fullgrp.get_next_mod()) != NULL) {
  do_remove = FALSE;
  /* size */
   if(!curmod->size.fits_thru(task.maxsize)) {
   do_remove = TRUE;
    cout << " Reduce_space filtered mod " << curmod->get_id() << "
on size\n";
  }

  /* environment */
   else if(!curmod->check_env(task.reqenv)) {
   do_remove = TRUE;
    cout << " Reduce_space filtered mod " << curmod->get_id() << "
on env\n";
  }
  if(do_remove)
    fullgrp.remove_mod();
 }

 /* insure EE works with task - required feet, gripper must meet grip
weight.
    Also, insure task is completable with these ee's.   TBD */
 /* kill all mods that don't have a possible power TBD */
```

```
  return(fullgrp.get_length());
}

void keep_best(Assembly test, Assembly best[TOPTEN]) {
 int i;
  for(i=TOPTEN-1; i>=0; i--) {
   if(test.score > best[i].score) {
    best[i+1] = best[i];
    best[i] = test;
   }
  }
}

inline void set_dump(int i, int n) {
/* for screen dump of intermediate data */
 if ((i % n) == 0)
   datadump = TRUE;
 else
   datadump = FALSE;
}


/* this function finds the CPU time (sec) spent since the first call,
  as long as it is called at least once every 35 min.
  The fn clock is only a long int, can't remember long times, so need
  this hack. */
bool time_tg = FALSE;
float time_tot = 0.0;

float get_time() {
  float time_cur = (float)clock() / CLOCKS_PER_SEC;
  if(time_cur >= 0)
   time_tg = FALSE;
  else if(!time_tg) {
   time_tg = TRUE;
   time_tot += 4296.0; // clock flips negative at 2148 seconds
  }
  return (time_cur + time_tot);
}
```

assemble.h

```
/*  Nathaniel Rutman

    assemble.h
    header file for assemble.C

   12/12/94  created

*/

#ifndef  _ASSEMBLE_H_
#define  _ASSEMBLE_H_

#include  "assyclass.h"

Assembly assemble(Group*, Task_Type*);
float  evaluate_assy(Assembly*, Group*, Task_Type*);


#endif
```

assemble.C

```
/*  Nathaniel Rutman

   assemble.C

   assembles all robots from a kit, evaluates them, and returns best
    evaluated assembly for that kit.

   change history:
   12/12/94  created NZR
   01/25/95  revised fitness function, scorekeeping NZR
   02/09/95  add 2-D ports NZR
       some of the assembly searches have not yet been implemented
        (see the empty procedures)
*/


#include "assemble.h"
#include "mymath.h"
#include "graphics.h"
#include "stats.h"

#define DRAWFREQ 100

extern bool datadump;
extern float MAGNIFY;
extern Stat_Type stats;

bool choose_next_assy(Assembly*);
bool assembly_rules(Assembly*);
bool choose_next_limbs(Assembly*);
bool set_first_limbs(Assembly *, int);
bool choose_next_body(Assembly*);
bool reorder_limb(Assembly*);
bool reorder_body(Assembly*);
bool rearrange_limbs(Assembly*);
bool swap_limb_parts(Assembly*);
bool swap_body_parts(Assembly*);
bool longer_limbs(Assembly*);
bool longer_body(Assembly*);

bool assydump=FALSE;

Assembly assemble(Group* kit, Task_Type* task) {
  /* Assembles kit into all possible assemblies,
    tests, evaluates assemblies. */
  Assembly test_assy(*kit);    /* create assembly from module group
*/
  Assembly best_assy;
  assydump=FALSE;
  //if (datadump) assydump=TRUE;   assydump prints out assy
details

  // first time, set up # legs
  if (test_assy.num_limbs == 0) {
   test_assy.num_limbs        =        test_assy.parts_bin.sg('E')-
>get_length();
    set_first_limbs(&test_assy,0);
  }
  // set up leg attach points (2nd leg at far end of robot)
  Point_Type far_pt(10.0,0.0);
  test_assy.body.attach_limb[1]                                  =
test_assy.body.get_attach_near(far_pt);

  // search all assemblies
  best_assy = test_assy;
   while(choose_next_assy(&test_assy)) {
   stats.cnt_assy++;
    if (assembly_rules(&test_assy)) {
    stats.cnt_assy_rules++;
      evaluate_assy(&test_assy, kit, task);
    if(test_assy.score>best_assy.score)
     best_assy = test_assy;
  }
 }

 // gather statistics
  stats.good_kit(kit->get_length());

 // print
 if (datadump) {
  cout << "Assembly scored " << best_assy.score << ":\n";
   best_assy.print();
  cout << endl;
 }

 // draw the robot
 if(stats.good_kits%DRAWFREQ == 0) {
  Point_Type origin(0.05,0.4);
  draw_clean();
  best_assy.draw(origin);
 }

 return best_assy;
}


bool choose_next_assy(Assembly* assy) {
 /* chooses next assembled configuration of the kit.
   body made up of all PWRs connected with JNTs, LNKs.
   # limbs = # EE, made up of remaining parts */

  if(!rearrange_limbs(assy))
   if(!choose_next_limbs(assy))
    if(!choose_next_body(assy))
     return FALSE;
 return TRUE;
}

bool rearrange_limbs(Assembly* assy) {
 /* rearrange limbs around body.
   Not implemented yet */
 //cout <<"rearranging  limbs: " << assy->num_limbs << " limbs
and ";
  //cout << assy->body.get_attach_points() << " ports.\n";
 return FALSE;
}

bool choose_next_limbs(Assembly* assy) {
 /* longest leg start from (l+j)/legs rounded up
   next longest leg: (remaining parts/remaining legs) ...
    s.t. all legs have at least 1 JNT, 1 EE
   leg = EE + x(JNT/LNK) + PWR.
   Rearrange mods within 1 leg
   Try different arrangements of leg set on body. */
  if(!reorder_limb(assy))
   if(!swap_limb_parts(assy))
    if(!longer_limbs(assy))
     return FALSE;
 return TRUE;
}

bool reorder_limb(Assembly* assy) {
 /* rearrange a single limb's mods.
   If no more, try next limb.  End when out of limbs.
   Needs to start with elements in order: 1 2 2 3
   Sequence is 1 2 3 4, 1 2 4 3, 1 3 2 4, 1 3 4 2, 1 4 2 3,
   1 4 3 2, 2 1 3 4, etc */
 int i,j,n,l=0,swp;
 Mod_Base *mod_tmp, *endeff;
 Mod_Base *ary[MAXMODS];
 while(l < assy->num_limbs) {
   assy->limb[l].reset_next();
```

```
endeff = assy->limb[l].get_next_mod();  // EE is first mod in limb
 n = assy->limb[l].get_length()-1;
// put id # into ary starting at ary[0] to ary[n-1]
i = 0;
 while((mod_tmp = assy->limb[l].get_next_mod()) != NULL)
 ary[i++] = mod_tmp;
// find next sequence
i = n-2;
 while (i>=0) {
   if ((ary[i]->get_id()) < (ary[i+1]->get_id())) {
   // swap for smallest subsequent mod > i
  swp = i+1;
   for(j = i+2; j<n; j++)
     if( (ary[j]->get_id() < ary[swp]->get_id())
         && (ary[j]->get_id() > ary[i]->get_id()) )
     swp = j;
   mod_tmp = ary[i];
   ary[i] = ary[swp];
   ary[swp] = mod_tmp;
   // reverse remaining #s
  j = 1;
   while(i+j < n-j) {
    mod_tmp = ary[i+j];
    ary[i+j] = ary[n-j];
    ary[n-j] = mod_tmp;
    j++;
   }
   //cout << "Old limb " << l << ": ";  assy->limb[l].print();
   assy->limb[l].clear();
   assy->limb[l].add_mod(endeff);
   for(i=0; i<n; i++)
     assy->limb[l].add_mod(ary[i]);
   //cout << "New: ";  assy->limb[l].print();
   return TRUE;
 }
 i--;
```

```
   }
   l++;  /* now goes through all limb 0, then all limb 1, etc.
          Needs to do all 0, then change limb 1 once, then do all 0
again... */
  }
  return FALSE;
}

bool swap_limb_parts(Assembly* assy) {
 /* swap parts across limbs.
   first one gets max jnts, lowest id's, swaps for higher id's, then
   for links.
   if out of swaps, try next leg up.
   End when out of legs.
   Not implemented yet. */
 return FALSE;
}

bool longer_limbs(Assembly* assy) {
 /* taking from shortest limb, increase next limb
   making sure to reset all subsequent limbs evenly.
   Stop when limb equals next longer limb.
   End when longest limb uses all available:
   all lnks, all but #legs jnts (leave 1 jnt for each leg) */

 // skew leg sizes
 Mod_Base* tmp;
 bool too_long;
 int i;
 do {
  if (assydump) cout << "skew shorter\n";
  /* move an extra j or l from a short leg to the next longer one */
  i = assy->num_limbs - 2;
  do {
      if((assy->limb[i].set_length < assy->limb[i-1].set_length)
      && assy->limb[i+1].set_length > 2)  // shortest length is 2
mods
        { assy->limb[i].set_length++;
          if (set_first_limbs(assy,i+1))
          return TRUE;
        }
    } while (--i > 0);

    if (assydump)  cout << "lengthen longest\n";
    /* can't move any, so increase first leg until out of parts */
     assy->limb[0].set_length++;
    /* need to leave 2 (j+e) for each limb */
    too_long = (assy->limb[0].set_length  + 2 * (assy->num_limbs-1))
>
         assy->parts_bin.get_length();
    if (too_long)
     return FALSE; // end condition
     if (set_first_limbs(assy,1))
     return TRUE;
  } while(1);
}

bool set_first_limbs(Assembly* assy, int level_pos) {
  /* makes the first limbs of a series based on set_length.
    level_pos is point at which the remaining mods are evenly
distributed */
  int legs = assy->num_limbs;  /* which is also # ee's */
   int parts = assy->parts_bin.sg('L')->get_length() +
    assy->parts_bin.sg('J')->get_length() + legs;
  int i;

  for (i = level_pos; i<legs; i++)
    assy->limb[i].set_length = 0;
  /* set rest of the leg lengths */
  i=0;
  do {
```

```
    if(assy->limb[i].set_length == 0)  // set evenly if allowed
     assy->limb[i].set_length = parts / legs + ((parts % legs) > 0);
    parts -= assy->limb[i].set_length;
   legs--;
   i++;
  } while(legs > 0);

  /* construct legs to length set above */
   assy->parts_bin.reset_next();
  for(i=0; i < assy->num_limbs; i++) {
    assy->limb[i].clear();
    assy->limb[i].add_mod(assy->parts_bin.get_next_mod(EE));
    assy->limb[i].add_mod(assy->parts_bin.get_next_mod(JNT));
  }
  Mod_Base* tmp;
  for (i=0; i < assy->num_limbs; i++) {
     while(assy->limb[i].get_length() < assy->limb[i].set_length) {
      if((tmp = assy->parts_bin.get_next_mod()) == NULL)
       goto bad_limb;
     assy->limb[i].add_mod(tmp);
   }
  }
  if (assydump) {
   cout << "valid\n";
   for(i=0; i < assy->num_limbs; i++) {
   cout << "limb " << i << " set to " << assy->limb[i].set_length
<< " ";
     assy->limb[i].print();
   }
  }
  return TRUE;

 bad_limb:
 if (assydump) {
   cout << "\ncan't make limbs with these parts:\n";
    assy->parts_bin.print();
    for(i=0; i < assy->num_limbs; i++) {
    cout << "limb " << i << " set to " << assy->limb[i].set_length
<< " ";
      assy->limb[i].print();
    }
  }
  return FALSE;
}


bool choose_next_body(Assembly* assy) {
  /* rearrange body, then try longer body */
   if(!reorder_body(assy))
     if(!swap_body_parts(assy))
      if(!longer_body(assy))
        return FALSE;
  return TRUE;
}

bool reorder_body(Assembly* assy) {
  /* rearrange body mods */
  return FALSE;
}

bool swap_body_parts(Assembly* assy) {
  /* swap out a j or l for another.
    dump limbs back into parts bin, choose from bin. */
  return FALSE;
}

bool longer_body(Assembly* assy) {
  /* increase body with j&l until only #legs jnts left
    dump limbs back into parts bin, choose from bin. */
  return FALSE;
}
```

```
bool assembly_rules(Assembly* assy) {
 /*  Assembly  rules  based  on  task,  etc  not  inherent  in
choose_next_assy
    pass/fail tests.
    geometry - can fit through smallest hole
   Grubler controllable
   force generation
    static stability
   mobility
   */
 /* check if last mod in limb is a link.
   Don't want them connected to body. */
 Sub_Assy* l_tmp;
 Mod_Base* tmp;
 Mod_Base* last;
  stats.test(TLINK);
 for (int i=0; i < assy->num_limbs; i++) {
   l_tmp = &(assy->limb[i]);
   l_tmp->reset_next();
   while((tmp = l_tmp->get_next_mod()) != NULL)
   last = tmp;
    if(last->get_type()=='L') {
     stats.fail();
   return FALSE;
  }
 }
 return TRUE;
}


float  evaluate_assy(Assembly*  assy,  Group*  kit,  Task_Type*
task) {
 /* rank assy based on non-dimensional quantities:
   controllability - bandwidth       vs. req speed
   end-effector lift, push force    vs. req manip forces
   speed                         vs. req speed
  accuracy                    vs. characteristic dim
  */

 /* mobility analysis - this one doesn't use environmental
interactions */
 int f,m,n;
  m = kit->get_length();
  f = kit->sg('J')->get_length();
 m -= f;
 n = 3*(m-1) - 2*f;

 /* leg strength, 2 evalutations.
   1. leg must lift itself: each joint must be able to lift ee through
   theta degrees from vert with all subsequent joints relaxed.
   2. leg must carry payload: lifting force of strongest joint,
   all others locked. */
 stats.test(TARM); // contraints below are test
 const float sintheta=0.5;  // 30 deg
 const float g=9.81;
 int i;
 float    arm_len[assy->num_limbs],    avg_arm_len=0.0,    dev,
st_dev=0.0;
  float arm,mass,leg_lift,body_lift,body_lift_mass,ap_torque;
 float total_arm=0.0, strong_arm=0.0;
 Mod_Base* tmp;
 for (i=0; i < assy->num_limbs; i++) {
   assy->limb[i].reset_next();
  arm = 0.0;   body_lift = 0.0; body_lift_mass = 0.0;
  mass = 0.0;   leg_lift = 0.0;
  arm_len[i]=0.0;
   while((tmp = assy->limb[i].get_next_mod()) != NULL) {
   arm += tmp->get_arm(); // arm segment length joint to joint
    arm_len[i] += tmp->get_arm();
    body_lift += tmp->weight * arm;
   mass += tmp->weight;  // total mass up to this point
    leg_lift += mass * tmp->get_arm();
      if(tmp->get_type() == 'J') {
       ap_torque = ((Mod_Joint *)tmp)->apply_torque/g/sintheta;
       if (ap_torque < leg_lift)
        goto bad_arm;  /* wimpy joint can't lift rest of leg */
       leg_lift = 0.0;
       body_lift_mass     =     max(body_lift_mass,(ap_torque     -
body_lift) / arm);
       arm = 0.0;
      body_lift = 0.0;
    }
   }
   strong_arm = max(strong_arm,body_lift_mass);
    total_arm += body_lift_mass;
   avg_arm_len += arm_len[i];
 }
 strong_arm = strong_arm / task->grip_weight;
 total_arm = 2 * total_arm  / kit->score.weight;   /* bilaterally
symmetric */

 /* limb length deviation */
 for (i=0; i < assy->num_limbs; i++) {
  dev = avg_arm_len - arm_len[i];
  st_dev += dev*dev;
 }
 st_dev = st_dev / assy->num_limbs;

 /* constraints */
 if((strong_arm<1) || (total_arm<1))
  goto bad_arm;
 /* fitness function */
 assy->score = kit->score.overall
 +      asymp(n)*4.0       +       asymp(strong_arm)*6.0       +
asymp(total_arm)*20.0
   - asymp(st_dev)*7.0;
 if (assydump) {
  cout << "DOFs: " << n << "  total leg payload " << total_arm;
  cout << "  max payload " << strong_arm << "\tleg dev " <<
st_dev;
   cout << " Score: " << assy->score << endl << endl;
 }
 return assy->score;

bad_arm:
 if (assydump) {
   cout << "failed assy eval.\n  total leg payload " << total_arm;
  cout << " max payload " << strong_arm;
 }
 assy->score = BAD_SCORE;
  stats.fail();
 return BAD_SCORE;
}
```

kit.h

```
/*
  Nathaniel Rutman

   kit.h

  just lets search_main know about these functions, described
 elsewhere

  12/15/94 created
*/


#ifndef _KIT_H_
#define _KIT_H_

#include "search_main.h"

void print_test_kit(int kit[GRPS][MAXLENGTH]);
bool kit_rules(Group*);
bool initialize_kit(int kit[GRPS][MAXLENGTH]);
bool convert_to_grp(int kit[GRPS][MAXLENGTH], Group*);
bool kit_rules_task(Group*, Task_Type*);
float evaluate_kit(Group*, Task_Type*);

#endif
```

kit.C

```
/*  Nathaniel Rutman


   kit.C

   Tests and evaluates robot kits, pre-assembly

   change history:
   11/14/94 created NZR
   01/24/95 better evaluation NZR
*/

#include  "kit.h"
#include  "mymath.h"
#include "groupclass.h"
#include  "search_main.h"
#include  "stats.h"
#include <string.h>

extern Group fullgrp;
extern bool datadump;
extern  Stat_Type  stats;

bool kit_rules(Group* test_kit){
  /* TRUE if kit satisfies simple kit grouping rules.
     Check for 1 pwr (satisfied always with choose_in)
     Check for #jnt >= #ee (satisfied with choose_in)
     Need 2 ee, 2 jnt (satisfied with choose_in)
    Check power types agree
    Check for enough ports for limbs
    Each mod has at least 1 other mod it likes to connect to
   */

  test_kit->reset_next();
  Mod_Base* tmp;
  Mod_Power* p_tmp;
  int port_tot = 0;
  stats.test(TPOWER);  // has power for everybody
  while((tmp = test_kit->get_next_mod()) != NULL) {
    if(!test_kit->has_power_for(tmp)) {
     //test_kit->print();
    //cout << "has no power for " << tmp->get_id() << endl;
     stats.fail();
    return FALSE;
   }
   if(tmp->get_type() == 'P') {
    p_tmp = (Mod_Power *)tmp;
    port_tot = port_tot + p_tmp->num_ports;
   }
  }
  stats.test(TPORTS);  // body has enough ports for limbs
  if(port_tot  <  test_kit->sg('E')->get_length()) {
   //cout << "kit rules:  not enough ports.  want ";
   //cout  <<   test_kit->sg('E')->get_length()   <<   "  have   "  <<
port_tot<<endl;
    stats.fail();
   return FALSE;
  }
  return TRUE;
}

bool   convert_to_grp(int   kit[GRPS][MAXLENGTH],   Group*
kit_MG) {
  /* put kit components into Group format */
  int i,j;
   kit_MG->clear_all();
  for(i=0; i<GRPS; i++)
   for(j=1; j<=kit[i][0]; j++)
      kit_MG->add_mod(fullgrp.get_mod(i,kit[i][j]-1));
  //cout << "converted to Group\n";
  //kit_MG->print();
 return TRUE;
}

bool kit_rules_task(Group* kit, Task_Type* task){
  /* TRUE if kit satisfies task - Absolute contraints.
    weight, cost, reliability, power consumption limits,
   span minimum, required feet.
   Many of these are done in evaluate_kit, and so to avoid
   duplication are omitted here. */

 /* make sure kit feet cover task requirements */
  stats.test(TFEET);
 char t[MAXENVIR];  // task required ee types
 char m[MAXENVIR];  // ee types available
  strcpy(t,task->req_ee);
  kit->sg((int)EE)->reset_next();
 Mod_Base *tmp;
  while((tmp=kit->sg((int)EE)->get_next_mod()) != NULL) {
    if(tmp->get_type()=='E')
      strcat(m,((Mod_EndEff *)tmp)->ee_type);
 }
  //cout << "task :"<<t<<":\tmods :"<<m<<":\n";
 int t_len = strlen(t);
 int m_len = strlen(m);
 int i,j;
 for(i=0; i<t_len; i++) {
  j=0;
   while(m[j]!=t[i])
    if(j++ >= m_len) {
      //cout << "No '"<<t[i]<<"' in :"<<m<<":\n";
      stats.fail();
      return FALSE;
   }
  m[j]=' ';
 }
```

```
  return TRUE;
}


float evaluate_kit(Group* kit, Task_Type* task) {
  /* fitness function based on which modules are in kit,
    (not on topology of assembled robot)
     average limb length, weight, speed, agility (dof), cost,
    reliability, power consumption, span.
     These must be reconciled with assemble.C's evaluate_assy */
  float limb_avg, span, step, agility, rslt;
  float body_length=0, limb_tot=0;
  float weight=0;
  float cost=0;
  float reli=1.0;
  float energy_req=0, energy_avail=0;
 int num_limbs = kit->sg((int)EE)->get_length();  /* num limbs =
num EEs */
 int symm, i;
 Mod_Base* tmp;
  kit->reset_next();
  while((tmp = kit->get_next_mod())!=NULL) {
  symm = 2;
   if(tmp->get_type() == 'P') {
    body_length += tmp->size.x;
     energy_avail += ((Mod_Power *)tmp)->energy_cap;
   symm = 1;  // bilaterally symmetric except for body
  }
  else {
    limb_tot += tmp->size.y;
     reli *= tmp->reliability;
  }
   weight += symm * tmp->weight;
  cost += symm * tmp->cost;
  energy_req += symm * tmp->energy_use;
   reli *= tmp->reliability; // could have used pwr(x,symm)
}
 kit->score.weight = weight;
kit->score.cost = cost;
 kit->score.reliability = reli;
kit->score.energy = energy_avail / energy_req;

  /* average limb length */
limb_avg = limb_tot / num_limbs;
  /* average span is 2 limbs plus body */
span = (body_length + 2*limb_avg) / task->ch_span;
step = 2*limb_avg / task->max_step;
  /* agility is percent joints */
agility        =        (float)kit->sg('J')->get_length()/(float)kit-
>get_length();
weight = weight / task->max_weight;
cost = cost / task->max_cost;

/* constraints */
 if((stats.test(TSTEP),step<1) || (stats.test(TSPAN),span<1)
  ||             (stats.test(TWEIGHT),weight>1)             ||
(stats.test(TCOST),cost>1)) {
   stats.fail();
  rslt = BAD_SCORE;
 }
 /* fitness function */
 else
  rslt = agility*5.0  + asymp(limb_avg)*1.0  + asymp(span)*10.0 +
reli*10.0
    + asymp(kit->score.energy)*10.0 - weight*15.0 - cost*10.0;
 kit->score.overall = rslt;
 if (datadump) {
   kit->print();
  cout  <<  "raw  limb  "  <<  limb_avg  <<  "\traw  body  "  <<
body_length << endl;
   cout << "step " << step << "\tspan " << span;
    cout << "\tweight " << weight << "\tagility " << agility;
    cout << "\ncost " << cost << "\treliability " << reli;
   cout << "\tenergy " << kit->score.energy;
   cout << "\trslt " << rslt << endl;
 }
 return(rslt);
}


bool initialize_kit(int kit[GRPS][MAXLENGTH]) {
 int i;
   /* initial kit: 1pwr, 4jnt, 4ee */
 kit[PWR][0] = 1;  /* number of PWRs in kit */
 kit[PWR][1] = 1;
 kit[EE][0] = MIN_FEET;  /* number of EE's and JNT's in kit */
 kit[JNT][0] = kit[EE][0];
 for(i=1; i<=kit[EE][0]; i++) {
  kit[JNT][i] = 1;
  kit[EE][i] = 1;
 }
 kit[LNK][0] = 0;
 for(i=0; i<GRPS; i++)
   if( (kit[i][0]>0) && (fullgrp.sg(i)->get_length() == 0) ) {
    cout << "Can't make any kits - no parts of group " << i << endl;
   return FALSE;
  }
 return TRUE;
}

void print_test_kit(int kit[GRPS][MAXLENGTH]) {
 cout << "Test kit  ";
 int i;
 cout << "Power: ";
 for (i=1; i<=kit[PWR][0]; i++)
  cout << kit[PWR][i] << " ";
 cout << "Joint: ";
 for (i=1; i<=kit[JNT][0]; i++)
```

```
   cout << kit[JNT][i] << " ";
  cout << "Link : ";
  for (i=1; i<=kit[LNK][0]; i++)
   cout << kit[LNK][i] << " ";
  cout << "EndEf: ";
  for (i=1; i<=kit[EE][0]; i++)
   cout << kit[EE][i] << " ";
  cout << endl;
}
```

choose.h

```
/*
 Nathaniel Rutman 03/20/95

  choose.h

  defines the sequence choosing procedure for kits
*/

#include "modclass.h"    /* for bool */
#include "groupclass.h"  /* for GRPS  */
#include "search_main.h" /* for MAXLENGTH */

#ifndef  _CHOOSE_H_
#define  _CHOOSE_H_

bool choose_next(int kit[GRPS][MAXLENGTH]);

#endif
```

choose_in.C

```
/*  Nathaniel Rutman

   choose_in.C

   chooses next robot kit.  This implementation chooses IN ORDER.
   exhaustive, but no memory required.
   # of combos is (n+l-1) choose k, or (n+k-1)!/k!(n-1)!
   Does use some basic assembly rules to avoid useless robots:
   need 4 ee always
   need as many jnts as ee always
   fiddles with lnks, then jnts, then ees, then pwrs.  This insures
   the slightest change from 1 kit to the next.  Maybe it would be
   better to go for max change instead?

   change history
   11/01/94  created NZR
   12/17/94 j/l sequencing bug fixed NZR
   02/13/94 j/l alternation removed NZR
*/

#include  "choose.h"
#include "search_main.h"    // for constants
#include  "kit.h"
#include "stats.h"        // just for printing

extern Group fullgrp;
extern  Stat_Type  stats;

bool sequence(int*, int);
bool trade_in(int kit[GRPS][MAXLENGTH]);
bool make_longer(int kit[GRPS][MAXLENGTH]);

bool choose_next(int kit[GRPS][MAXLENGTH]) {
  /* choose a new kit.  The present system chooses in order,
    but might want to choose randomly with a memory.
    This sequence changes the "least important" modules
    first, but this may be wrong.
    Returns TRUE if a new kit is made */
   if(!sequence(kit[LNK],  fullgrp.sg('L')->get_length()))
     if(!sequence(kit[JNT],fullgrp.sg('J')->get_length()))
      if(!sequence(kit[EE],  fullgrp.sg('E')->get_length()))
      if(!sequence(kit[PWR],  fullgrp.sg('P')->get_length())) {
      //  cout << "out of different mods, trying different types" <<
endl;
          if(!trade_in(kit))
           if(!make_longer(kit))
           return(FALSE);
      }
  //  print_test_kit(kit);
  return(TRUE);
}

bool make_longer(int kit[GRPS][MAXLENGTH]) {
 /* tried all combos this length, so need to make longer.
   This needs to add 1 of the first type swapped OUT from
   trade_in, decided from calling fn.  Reset PWRs.
   Maximum robot length is MAXLENGTH  */
 int lng;
 lng = kit[PWR][0]+kit[JNT][0]+kit[LNK][0]+kit[EE][0];
 stats.print(lng);
 cout << "out of trades, trying longer robot " << lng+1 << endl;
 if(lng < MAXLENGTH) {
  kit[LNK][0]=kit[PWR][0];
  kit[PWR][0]=1;
  for(int i=1; i<=kit[LNK][0]; i++)
   kit[LNK][i] = 1;
  return(TRUE);
 }
 return(FALSE);
}
```

```
bool trade_in(int kit[GRPS][MAXLENGTH]) {
 /* keep same length robot, but try different types:
   2pwr, 4jnt instead of 1pwr, 5jnt.
   trade order: lnk to jnt to ee to pwr
   Returns FALSE if out of trades. */
 if(kit[LNK][0] > 0) {       /* trade a LNK to a JNT  */
  kit[JNT][0]++;
  kit[JNT][kit[JNT][0]] = 1;
  kit[LNK][0]--;
  return(TRUE);
 }
 /* out of LNKs.  Trade a JNT to a EE */
 if(kit[JNT][0] > kit[EE][0]+1) {
  /* min #ee JNTs.  Add 1 because new #ee = old #ee + 1 */
  kit[EE][0]++;
  kit[EE][kit[EE][0]] = 1;
  kit[LNK][0] = kit[JNT][0] -kit[EE][0] -1;  /* change extras back to
LNKs */
  kit[JNT][0] = kit[EE][0];
  return(TRUE);
 }
 if((kit[EE][0] > MIN_FEET) | | (kit[JNT][0] > MIN_FEET)) {
  /* extra JNT or EE, change to PWR */
  kit[PWR][0]++;
  kit[PWR][kit[PWR][0]] = 1;
  kit[LNK][0] = kit[EE][0]-MIN_FEET + kit[JNT][0]-MIN_FEET -1;
  kit[EE][0] = MIN_FEET;
  kit[JNT][0] = MIN_FEET;
  return(TRUE);
 }
 return(FALSE);
}

bool sequence(int* a, int max) {
 /* finds next sequence of mods of a particular type.
   Order is irrelevant, choice up to max for each mod,
   a[0] is the number of mods, a[i] is which mod.
   TRUE if next sequence found.  Exits clean for no mods. */
 int i,j;
 bool carry = TRUE;
 i = a[0];   /* last "decimal" place */
 while (carry && i>0) {
  carry = FALSE;
  if(++a[i] > max) {
   carry = TRUE;
   i--;  /* move left 1 "decimal" place */
  }
 }
 if(i>0)
  for(j=i; j<a[0]; j++)
   a[j+1] = a[j];     /* sequence is: 1 1 3 3 to 1 2 2 2 */
 else
  for(j=1; j<=a[0]; j++)
   a[j] = 1;      /* overflow, reset to: 1 1 1 1 */
 return(!carry);
}
```

mod_desc.h

```
/*
 Nathaniel Rutman 03/20/95

  mod_desc.h

*/


#ifndef _MOD_DESC_H_
#define _MOD_DESC_H_


int get_module_desc();


#endif
```

mod_desc.C

```
/*
 Nathaniel Rutman

 mod_desc.C

 Describes all available modules from datafile mod.dat
and groups into classes
 Describes task

 revision history:
 10/28/94 created NZR
 12/17/94 added jacobian NZR
 02/09/95 removed jacobian, added power ports NZR
*/

#include "mod_desc.h"
#include "modclass.h"
#include "groupclass.h"
#include <stdio.h>


extern Group fullgrp;
extern bool datadump;

int get_module_desc() {
 cout << "descibing modules" << endl;
 Mod_Base* bp;

 /* file read */
 FILE *infile;
 char scan_ch;
 int numcase=0;
 int m_id,i,j;
 char modclass, power_type;
 Env_Type okenv[MAXENVIR];
 float cost, sz1, sz2, sz3, wt, rl, en_u, x,y;
 infile = fopen("mod.dat", "r");
 while(getc(infile) != '!'); // comments until this mark
  while(!feof(infile)) {
  numcase++;
   fscanf(infile,"%d %f %c %c ", &m_id, &cost, &modclass,
&power_type);
   fscanf(infile, "%s ", okenv);
   fscanf(infile, "%f %f %f %f %f %f ", &sz1, &sz2, &sz3, &wt, &rl,
&en_u);
  switch (modclass) {
  case 'P':
   Mod_Power * pp;  /* create with new so they live forever */
   pp = new Mod_Power;
   bp = pp;
    fscanf(infile, "%f %d ", &pp->energy_cap, &i);
   pp->num_ports = i;  // read in ports
   for(j=0; j<i; j++) {
     fscanf(infile, "%f %f ", &x, &y);
     pp->port[j].x = x;
     pp->port[j].y = y;
  }
   break;
  case 'L':
   Mod_Link * lp;
   lp = new Mod_Link;
   bp = lp;
    fscanf(infile, "%f %f ", &lp->support_torque, &lp-
>arm_length);
   break;
  case 'J':
   Mod_Joint * jp;
   jp = new Mod_Joint;
   bp = jp;
```

```
    fscanf(infile, "%f %f %f ", &jp->support_torque, &jp-
>arm_length, &jp->apply_torque);
    fscanf(infile, "%f %f %f ", &jp->speed, &jp->angle_min, &jp-
>angle_max);
    break;
   case 'E':
    Mod_EndEff * ep;
    ep = new Mod_EndEff;
    bp = ep;
    fscanf(infile, "%f %f %f ", &ep->support_torque, &ep-
>arm_length, &ep->apply_torque);
     fscanf(infile, "%f %s ", &ep->speed, &ep->ee_type);
    break;
   default:
    cout << "Unknown module type " << modclass << endl;
     cout << "Bailing..." << endl;
   return 0;
    fclose(infile);
   break;
  }
   bp->set_id(m_id, modclass);
  bp->cost = cost;
   bp->reliability = rl;
  bp->pwrtype = power_type;
  bp->okenvs = okenv;
  bp->size.x = sz1;
  bp->size.y = sz2;
  bp->size.z = sz3;
  bp->weight = wt;
  bp->energy_use = en_u;
  fullgrp.add_mod(bp);
  if (datadump)
   bp->print();
  else
   bp->print_id();
  fscanf(infile,"\n");
  }
   fclose(infile);
  cout << "Described " << numcase << " modules." << endl;
  return(numcase);
}



/* all units are MKS */

void get_task(Task_Type* task) {
  task->max_cost = 1000.00;
  task->max_weight = 9.09;
  task->max_step = 0.2125;
  task->ch_span = 0.088;
  task->reqenv = "DW"; // all mods must work in Dark and Wet
  task->req_ee = "HHG"; // need 2 feet for Horizontal ground, 1
Gripper for moving junk
  task->grip_weight = 0.12; // max weight for gripper
  task->maxsize.x = 0.088;     // maximum internal dimensions
allowed
  task->maxsize.y = 0.1375; // narrowest horiz path
  task->maxsize.z = 0.088;
  if (datadump)
    task->print();
}
```

assyclass.h

```
/***************************************************************
********
 Nathaniel Rutman

 assyclass.h

 describes robot assembly class, which is a group of mods +
connections

 12/13/94 created

***************************************************************
*******/

#include "modclass.h"
#include "groupclass.h"
#include <iostream.h>

#ifndef _ASSYCLASS_H_
#define _ASSYCLASS_H_

#define MAXLIMBS 10        /* max number of legs/arms per robot */
#define MAX_ASY_PORTS 100   /* max # ports to keep track of an
assy */

class Sub_Assy {
 protected:
  char type;
  Sub_Group grp;
 public:
  Sub_Assy();
  char get_type();
  int get_length();
  virtual void add_mod(Mod_Base *) = NULL;
  void remove_mod();
  void reset_next();
  Mod_Base* get_next_mod();
  void clear();
  virtual void print() = NULL;
  virtual void draw(Point_Type) = NULL;
};

class Body_Type:public Sub_Assy {
 private:
  int attach_points;
  Point_Type point_array[MAX_ASY_PORTS+1];
 public:
  int attach_limb[MAXLIMBS];
  Body_Type();
  void add_mod(Mod_Base *);
  void print();
  void draw(Point_Type);
  Point_Type get_attach_point(int);
  int get_attach_near(Point_Type);
  int get_attach_points();
};

class Limb_Type:public Sub_Assy {
 public:
  int set_length;
  Limb_Type();
  void add_mod(Mod_Base *);
  void print();
  void draw(Point_Type);
};



class Assembly {
 public:
  Limb_Type limb[MAXLIMBS];
  Body_Type body;
  Group parts_bin;
  int num_limbs;
  float score;
  Assembly();
  Assembly(Group);
  void print();
  void draw(Point_Type);
};


#endif
```

assyclass.C

```
/*
 Nathaniel Rutman

 assyclass.C

 Assembly class methods

 12/16/94 created
 01/11/95 limb module ordering added
*/


#include "assyclass.h"
#include "graphics.h"
#include <iostream.h>
#include <stdio.h>

/**************************                    Sub_Assy
***********************/

Sub_Assy::Sub_Assy() {
  type = 'U';
  grp.set_type('A'); // accept all mods
}

int Sub_Assy::get_length() {
  return grp.get_length();
}

char Sub_Assy::get_type() {
  return type;
}

void Sub_Assy::add_mod(Mod_Base * m) {
  grp.add_mod(m);
  /* make some connections here */
}

void Sub_Assy::reset_next() {
  grp.reset_next();
}

Mod_Base* Sub_Assy::get_next_mod() {
  return grp.get_next_mod();
}

void Sub_Assy::remove_mod() {
/* may need to specify which mod */
/* break connections, too */
  grp.remove_mod();
}

void Sub_Assy::clear() {
  grp.clear();
}



/*********************** Sub_Assy children ***************/
/********** Body_Type **********/

Body_Type::Body_Type() {
  type = 'B';
  for(int i=0; i<MAXLIMBS; i++)
    attach_limb[i]=i;
  attach_points=0;
}

void Body_Type::add_mod(Mod_Base * m) {
  if(m->get_type() == 'E')
  cout << "\nBody_Type::add_mod - Trying to add an End Eff to
the body\n";
  grp.add_mod(m);

  // collect port info
  if(m->get_type() == 'P') {
   Mod_Power *p_tmp;
   p_tmp = (Mod_Power *)m;
    for(int i=0; i<p_tmp->num_ports; i++) {
       point_array[attach_points]=point_array[MAX_ASY_PORTS] +
p_tmp->get_port(i);
      attach_points ++;
       if(attach_points >= MAX_ASY_PORTS)
        cout << "Body_Type::add_mod:  warning - too many attach
points!!\n";
    }
  }
  point_array[MAX_ASY_PORTS].x += m->size.x;
}

Point_Type Body_Type::get_attach_point(int limb_num) {
  return point_array[attach_limb[limb_num]];
}

int Body_Type::get_attach_near(Point_Type near) {
  if(attach_points < 1) {
   cout << "Body_Type::get_attach_near:  no attach points\n";
   return 0;
  }
  int close_port;
  float dx,dy,dist;
  float min_dist = 9999999.0;
  for(int i=0; i<attach_points; i++) {
   dx = near.x - point_array[i].x;
   dy = near.y - point_array[i].y;
   dist = (dx*dx + dy*dy);
    if (dist<min_dist) {
     min_dist=dist;
     close_port = i;
    }
  }
  return close_port;
}

int Body_Type::get_attach_points() {
  return attach_points;
}

void Body_Type::print() {
  cout << "body: ";
  grp.print();
}

void Body_Type::draw(Point_Type pnt) {
  reset_next();
  Mod_Base *tmp;
  Point_Type t_pnt = pnt;
  while((tmp=get_next_mod()) != NULL) {
    tmp->draw(t_pnt);
   t_pnt.x += tmp->size.x;
  }
}

/********** Limb_Type ***********/

Limb_Type::Limb_Type() {
  type = 'L';
  set_length = 0;
}

void Limb_Type::add_mod(Mod_Base * m) {
  if(m->get_type() == 'P')
  cout << "\nLimb_Type::add_mod - Trying to add power to a
limb\n";
  grp.add_mod(m);
}

void Limb_Type::print() {
  cout << "limb: ";
  grp.print();
}

void Limb_Type::draw(Point_Type pnt) {
  Mod_Base *tmp;
  Point_Type t_pnt;
  t_pnt = pnt;
  float high = 0.0;
  reset_next();
  while((tmp=get_next_mod()) != NULL)     // find arm height
    high += tmp->get_arm();
  t_pnt.y -= high;
  reset_next();
  while((tmp=get_next_mod()) != NULL) {  // draw arm bottom up
    t_pnt.y += tmp->get_arm();
    tmp->draw(t_pnt);
  }
}


/*************************    Assy    *********************/


Assembly::Assembly() {
  num_limbs = 0;
  score = BAD_SCORE;
}

Assembly::Assembly(Group g) {
```

```
  num_limbs = 0;
  score = g.score.overall;
  parts_bin = g;
  Mod_Base* tmp;
   while((tmp=parts_bin.get_next_mod()) != NULL)
    if (tmp->get_type() == 'P') {
     body.add_mod(tmp);
     parts_bin.remove_mod();
    }
  }

void Assembly::print() {
  cout << "Assembly:\n";
  cout << "parts bin: ";
   parts_bin.print();
  body.print();
  for(int i=0; i<num_limbs; i++) {
    cout << "#" << i << " ";
    limb[i].print();
  }
  cout << "score :" << score <<endl;
}

void Assembly::draw(Point_Type p0) {
  Point_Type pnt;
  body.draw(p0);
  for(int i=0; i<num_limbs; i++) {
    pnt = body.get_attach_point(i);
    //cout<<"Assembly::draw: leg "<<i<<" attached at port
"<<body.attach_limb[i]<<endl<<endl;
    limb[i].draw(p0+pnt);
  }

  //print score
  draw_color(BLACK);
 char msg[50];
```

```
  sprintf(msg,"score: %.2f",score);  // print to a string
  draw_string(p0.x,p0.y+0.05,msg);

  draw_update();
}
```

groupclass.h

```
/************************************************************
********
  Nathaniel Rutman

  groupclass.h

  describes Group and Sub_Group (only 1 type of mod) classes
  Groups are bins of unassembled modules, subdivided into subgroups
by
  function: pwr, lnk, ee, jnt

  change history:
  12/13/94  created NZR

************************************************************
*******/

#include "modclass.h"
#include <iostream.h>


#ifndef _GROUPCLASS_H_
#define _GROUPCLASS_H_

#define BAD_SCORE -100.0

enum modtypes {PWR, LNK, JNT, EE, GRPS};   // GRPS is # of
groups

class Score_Type {
 public:
  float overall;
  float weight;
  float reliability;
  float cost;
  float energy;
  Score_Type();
};

class Sub_Group {
 /* made up of only one type of module */
 private:
  char mod_type;
  Mod_Base* mod_list[MAXMODS];
  int length;
  int index;
 public:
  Sub_Group();
  Sub_Group (char);
  void set_type(char);
  char get_type();
  int get_length();
  Mod_Base* get_next_mod();
  Mod_Base* get_mod(int);
  void reset_next();
  Mod_Base* get_different_mod();
 bool add_grp(Sub_Group *);
 bool add_mod(Mod_Base *);
 bool remove_mod();
  void clear();
  void print();
};


class Group {
 protected:
 Sub_Group subg[4];
 int num_subg;
 int subg_index;
 int tot_length;
```

```
  bool recalc;
 public:
  Score_Type score;
  Group();
  Sub_Group* sg(char);
  Sub_Group* sg(int);
  void add_a_sg(Sub_Group);
  void add_mod(Mod_Base *);
  void remove_mod();
   void clear_all();
  int get_length();
  void reset_next();
  Mod_Base* get_next_mod();
  Mod_Base* get_next_mod(int);
  Mod_Base* get_mod(int,int);
  bool has_power_for(Mod_Base*);
 void print();
};


#endif
```

groupclass.C

```
/* Nathaniel Rutman

  groupclass.C

  Sub_Group, Groupclass methods

  01/11/95  created
*/


#include "groupclass.h"
#include <iostream.h>


/**************************            Score_Type          class
*********************/
 Score_Type::Score_Type() {
   overall = BAD_SCORE;
 }

/**************************            Sub_Group           class
**********************/

Sub_Group::Sub_Group () {
  mod_type = 'U';  // undefined so far
  length = 0;
  index = 0;
}

Sub_Group::Sub_Group (char c) {
  mod_type = c;
  length = 0;
  index = 0;
}
```

```
void Sub_Group::set_type(char c) {
  mod_type = c;
}

char Sub_Group::get_type() {
  return mod_type;
}

int Sub_Group::get_length() {
  return length;
}

void Sub_Group::reset_next() {
  index=0;
}

Mod_Base* Sub_Group::get_mod(int n) {
  if(n<0 || n>=length) {
    cout << "Sub_Group::get_mod:  out of range\n";
    return NULL;
  }
  return mod_list[n];
}

Mod_Base* Sub_Group::get_next_mod() {
  if(index >= length) {
//    cout << "Sub_Group::get_next_mod():  Out of mods\n";
    reset_next();
    return NULL;
  }
  index++;
  return mod_list[index-1];
}

Mod_Base* Sub_Group::get_different_mod() {
  if(index >= length) {
   cout << "Sub_Group::get_different_mod():  Out of mods\n";
   reset_next();
   return NULL;
  }
  if(index==0) return mod_list[0];
  int olddex=index;
  while(mod_list[index]==mod_list[olddex]) {
   index++;
   if (index>=length) {
    cout << "Sub_Group::get_different_mod():  Out of mods\n";
    reset_next();
    return NULL;
   }
  }
  return mod_list[index];
 }

 bool Sub_Group::add_mod(Mod_Base * a) {
  if (length>=MAXMODS) {
   cout  <<  "\n\nSub_Group::add_mod:    Exceeded   maximum
 length\n";
   return FALSE;
  }
  if(a == NULL) {
   cout << "\n\nSub_Group::add_mod:  Adding NULL module\n";
   return FALSE;
  }
  if((mod_type != 'A') && (a->get_type() != mod_type))  // A is
any type
  cout << "\n Sub_Group::add_mod: Warning-wrong type " << a-
>get_type()
    << " in subgroup of " << get_type() << endl;
  mod_list[length++] = a;
  return TRUE;
}
```

```
 bool Sub_Group::remove_mod() {
  /* removes last mod checked, which is index-1 */
 //  cout << "Sub_Group::remove_mod " << index-1;
  if (index<1 || index>length) {
   cout << "Sub_Group::remove_mod: index " << index-1 << " Out of
range\n";
   return(FALSE);
  }
  else {
   index--;
//    cout << " removing id "<< mod_list[index]->get_id() << endl;
   int i;
   length--;
   for(i=index; i<length; i++)
    mod_list[i] = mod_list[i+1];
    /* this may leave mod with nothing pointing to it.
      may want to delete if this is the last reference */
   return(TRUE);
  }
 }

 void Sub_Group::clear() {
  index=0;
  length=0;
 }

 bool Sub_Group::add_grp(Sub_Group * a) {
  /* copies an entire group */
  cout << "Sub_Group::add_grp(Sub_Group * a)" << endl;
  a->reset_next();
  int i;
  if(a->length > 0)
   for(i=0; i<a->length; i++)
    add_mod(a->get_next_mod());
  return(TRUE);
 }

 void Sub_Group::print() {
  cout << length << " " << mod_type << "'s: ";
  int i;
  for(i=0; i<length; i++) {
    mod_list[i]->print_id();
   cout << " ";
  }
  cout << endl;
 }


/**************************  Group  *************************/

int lookup_type(char c) {
  if(c=='P') return PWR;
  if(c=='J') return JNT;
  if(c=='L') return LNK;
  if(c=='E') return EE;
 cout << "Unknown group type.  Treating as Power\n";
 return 0;
}

Group::Group() {
  tot_length=0;
 num_subg=4;
 subg_index=0;
  subg[PWR].set_type('P');
  subg[LNK].set_type('L');
  subg[JNT].set_type('J');
  subg[EE].set_type('E');
 recalc=TRUE;
}

Sub_Group* Group::sg(char c) {
```

```
 /* returns a pointer to the subgroup of type c */
  return &subg[lookup_type(c)];
 }

Sub_Group* Group::sg(int i) {
/* returns a pointer to subgroup i */
  if((i>3) || (i<0)) {
   cout << "Group: SubGroup #"<<i<<" out of range.\n";
   return NULL;
  }
 return &subg[i];
}

void Group::add_a_sg(Sub_Group newsg) {
 /* choose subg number by newsg's type */
 char c = newsg.get_type();
 if(num_subg<lookup_type(c))
  num_subg=lookup_type(c);

  tot_length -= sg(c)->get_length();
 subg[lookup_type(c)] = newsg;
 subg[lookup_type(c)].reset_next();
  tot_length += sg(c)->get_length();
 recalc=TRUE;
}

int Group::get_length() {
 return tot_length;
}

Mod_Base* Group::get_next_mod() {
 Mod_Base* tmp;
 if ((subg_index >= num_subg) || (subg_index<0)) {
  cout << "Group::get_next_mod - subg_index out of range\n";
  return NULL;
 }
```

```
  while((tmp = subg[subg_index].get_next_mod()) == NULL) {
   subg_index++;
   if (subg_index>=num_subg) {
     reset_next();
    return NULL;
   }
  }
  return tmp;
 }

 /*
 Mod_Base* Group::get_next_mod(char c) {
  subg_index = lookup_type(c);
  return subg[subg_index].get_next_mod();
 }
 */

 Mod_Base* Group::get_next_mod(int i) {
  return subg[i].get_next_mod();
 }

 Mod_Base* Group::get_mod(int grp, int ndx) {
  return subg[grp].get_mod(ndx);
 }

 void Group::reset_next() {
  subg_index=0;
  for(int i = 0; i<num_subg; i++)
    subg[i].reset_next();
 }

 void Group::add_mod(Mod_Base* mod) {
  /* add mod to subg of right type */
  char c = mod->get_type();
  sg(c)->add_mod(mod);
  recalc=TRUE;
  tot_length++;
 }

 void Group::remove_mod() {
 // cout << "Group::remove_mod ";
  sg(subg_index)->remove_mod();
  recalc=TRUE;
  tot_length--;
 }

 void Group::clear_all() {
  subg_index = 0;
  for(int i = 0; i<num_subg; i++)
    subg[i].clear();
  tot_length = 0;
 }

 void Group::print() {
  if (tot_length == 0)
   cout << " group empty.\n";
  else {
   cout << "Group length " << tot_length << endl;
   for(int i = 0; i<num_subg; i++)
     if (subg[i].get_length() > 0)
      subg[i].print();
  }
 }

 bool Group::has_power_for(Mod_Base* mod) {
  /* looks through all PWR looking for a match */
  char c = mod->pwrtype;
  if(c == NO_POWER)
   return TRUE;
  int n=0;
  Mod_Base* tmp;
  while (n < sg('P')->get_length()) {
     tmp = sg('P')->get_mod(n++);
     if(tmp->pwrtype == c)
      return TRUE;
    n++;
   }
   return FALSE;
  }

  /*
  float Group::get_weight() {
    if (!recalc)
     return weight;
   reset_next();
   Mod_Base* tmp;
   float w=0, c=0;
   while((tmp = get_next_mod())!=NULL) {
    w += tmp->weight;
    c += tmp->cost;
   }
   weight = w;
  cost = c;
  return w;
 }

 float Group::get_cost() {
   get_weight();
  return cost;
 }
 */
```

modclass.h

```
/*
  Nathaniel Rutman

  modclass.h

  header file class descriptor for module-level classes.
  These describe in detail a single module.
  These are used by the grouping classes (assyclass).

  12/13/94 created
  11/31/95 Jacobian added
  02/09/95 Jacobian removed, 2D ports added, task revised
*/


#ifndef _MODCLASS_H_
#define _MODCLASS_H_

#include "mymath.h"
#include <iostream.h>

#define TRUE 1
#define FALSE 0

#define MAXENVIR 10
#define MAXMODS 30
#define MAXPORTS 10
#define NO_POWER 'Z'     /* when no power is required, this is
pwrtype */

#define DRAW_PORT_CLR 0.003    /* minimum size around port
when drawing */

typedef int bool;
```

```
typedef char Env_Type;
typedef char Foot_Type;

class Size_Type {
 public:
  float x,y,z;
  Size_Type();
  void print();
  bool fits_thru(Size_Type);
};


class Mod_Base {
 protected:
  int mod_id;
  char mod_class;
 public:
  char pwrtype;
  float energy_use;
  Env_Type okenvs[MAXENVIR];
  Size_Type size;
  float weight,cost,reliability;

  Mod_Base();
  void set_id(int,char);
  char get_type();
  void print_id();
  void print_base();
  int get_id();
  bool check_env(Env_Type*);
  virtual void print() = NULL;
  virtual void draw(Point_Type) = NULL;
  virtual float get_arm() = NULL;
};
```

```
class Mod_Power:public Mod_Base {
 public:
  float energy_cap;
  int num_ports;
  Point_Type port[MAXPORTS];
  Point_Type get_port(int);
  float get_arm();
  void print();
  void draw(Point_Type);
};

class Mod_Link:public Mod_Base {
 public:
  float support_torque;
  float arm_length;
  float get_arm();
  void print();
  void draw(Point_Type);
};

class Mod_Joint:public Mod_Base {
 public:
  float speed, apply_torque, support_torque;
  float arm_length;
  float angle_min, angle_max;
  float get_arm();
  void print();
  void draw(Point_Type);
};

class Mod_EndEff:public Mod_Base {
 public:
  float speed, apply_torque, support_torque;
  float arm_length;
  Env_Type ee_type[MAXENVIR];
  float get_arm();
  void print ();
  void draw(Point_Type);
};


class Task_Type {
 public:
  float max_cost;
  float max_weight;
  float max_step;
  float ch_span; // maximum span that robot must cross
  Env_Type reqenv[MAXENVIR];  // all mods must work in these
envs
  Foot_Type req_ee[MAXMODS];  // need this list of end effector
types
  Size_Type maxsize; // maximum size for robot in a minimum size
configuration
  float grip_weight; //  gripper must lift this weight
  void print();
};


#endif
```

modclass.C

```
/*
 Nathaniel Rutman

 modclass.C

 function bodies of module-level classes described in modclass.h

 change history:
 12/13/94  created NZR
 01/31/95  Jacobian added NZR
 02/09/95  Jacobian removed, 2D ports added NZR
 02/22/95  Drawing functions added NZR
*/

#include  "mymath.h"
#include  "modclass.h"
#include  "graphics.h"
#include  <iostream.h>
#include  <fstream.h>
#include  <string.h>


/**********************                              Size_Type
******************************/


 Size_Type::Size_Type() {
  x=1.0; // default sizes
  y=1.0;
  z=1.0;
 }

bool Size_Type::fits_thru(Size_Type s2) {
 /* checks if a fits through b horiz or vert */
 float a = min(x,y);    // smallest mod dim
 float b = min(s2.x,s2.y); // smallest hole dim
 if ( (a > b) || (z > s2.z) )
  return(FALSE);
 else
  return(TRUE);
}

void Size_Type::print() {
 cout << "\t(" << x << "," << y << "," << z << ")m";
}


/**********************                              Mod_Base
******************************/


 Mod_Base::Mod_Base() {
  mod_id = 999;
  mod_class = 'U';
 }

void Mod_Base::set_id(int i, char c) {
 mod_id = i;
 mod_class = c;
}

char Mod_Base::get_type() {
 return mod_class;
}

bool Mod_Base::check_env(Env_Type* want_env) {
 /* make sure a mod can work in all wanted environments */
 int i=0,j=0;
 bool found=TRUE;
  while(i<strlen(want_env) && found) {
   found = FALSE;
   j = 0;
    while(j<strlen(okenvs) && !found) {
     if(want_env[i]==okenvs[j]) found=TRUE;
    j++;
   }
   i++;
  }
  return(found);
 }


void Mod_Base::print_id() {
  cout << mod_class << " #" << mod_id << "  ";
}

void Mod_Base::print_base() {
  cout << "\t $" << cost;
  cout << "\t" << weight << " kg";
  cout << "\t" << pwrtype << " pwr";
  cout << "\trely " << reliability;
  size.print();
 cout << "\n\t";
}

int Mod_Base::get_id() {
 return mod_id;
}

/************************          Mod_Base     child      classes
******************/

/********* Mod_Power *********/

float Mod_Power::get_arm() {
 cout  <<  "\nWarning!!  Power  mod  doesn't  have  an  arm
length.\n\n";
```

```
  return 0;
 }

 Point_Type Mod_Power::get_port(int num) {
  if(num>=num_ports) {
   cout << "Mod_Power::get_port:  port number out of range\n";
   return port[0];
  }
  return port[num];
 }

void Mod_Power::draw(Point_Type pnt) {
  draw_color(RED);
  draw_rect(pnt.x,pnt.y,size.x,size.y);
  draw_color(PINK);
 for(int i=0; i<num_ports; i++)
    draw_circle(pnt.x+port[i].x,pnt.y+port[i].y,0.005);
  draw_color(BLACK);
  draw_box(pnt.x,pnt.y,size.x,size.y);
}

void Mod_Power::print() {
 cout << "Pwr " << mod_id;
 print_base();
 cout << energy_cap << " watt-hrs";
 cout << "\t" << num_ports << " ports\n";
}

/********* Mod_Link *********/

float Mod_Link::get_arm() {
 return arm_length;
}

void Mod_Link::draw(Point_Type pnt) {
 const float a = size.x/2.0;
```

```
  const float b = (size.y - arm_length)/2.0;
  draw_color(BLUE);
   draw_rect( pnt.x-a,pnt.y-arm_length-b,size.x,size.y);
  draw_color(LT_BLUE);
  draw_circle( pnt.x, pnt.y, a/2.0);
  draw_circle( pnt.x, pnt.y-arm_length, a/2.0);
}

void Mod_Link::print() {
 cout << "Lnk " << mod_id;
  print_base();
 cout << endl;
}


/********* Mod_Joint *********/

float Mod_Joint::get_arm() {
 return arm_length;
}

void Mod_Joint::draw(Point_Type pnt) {
 const float a = size.x/2.0;
 const float b = (size.y - arm_length)/2.0;
 draw_color(GREY);
  draw_circle(pnt.x,pnt.y,a);
  draw_rect(pnt.x-a/2.0,pnt.y-arm_length-
DRAW_PORT_CLR,a,size.y);
/*
 draw_color(MAROON);
 draw_line(pnt.x+DRAW_PORT_CLR, pnt.y+b, pnt.x+a, pnt.y);
 draw_line_to( pnt.x+DRAW_PORT_CLR, pnt.y-b);
 draw_line_to( pnt.x+DRAW_PORT_CLR, pnt.y-arm_length-b);
 draw_line_to( pnt.x-DRAW_PORT_CLR, pnt.y-arm_length-b);
 draw_line_to( pnt.x-DRAW_PORT_CLR , pnt.y-b );
 draw_line_to( pnt.x-a , pnt.y );
 draw_line_to( pnt.x-DRAW_PORT_CLR , pnt.y+b );
 draw_line_to( pnt.x+DRAW_PORT_CLR , pnt.y+b );
*/
}

void Mod_Joint::print() {
 cout << "Jnt " << mod_id;
  print_base();
 cout << apply_torque << " Nm\n";
}


/********* Mod_EndEff *********/

float Mod_EndEff::get_arm() {
 return arm_length;
}

void Mod_EndEff::draw(Point_Type pnt) {
 const float a = size.x/2.0;
 const float b = size.y - arm_length;
  draw_color(BROWN);
 /* gripper */
 if(mod_id < 420) {
  draw_line(  pnt.x-a/2.0  ,  pnt.y-arm_length,  pnt.x-a  ,  pnt.y-
arm_length*0.666 );
  draw_line_to(      pnt.x-DRAW_PORT_CLR          ,       pnt.y-
arm_length*0.125 );
  draw_line_to( pnt.x-DRAW_PORT_CLR , pnt.y+b );
  draw_line_to( pnt.x+DRAW_PORT_CLR , pnt.y+b );
  draw_line_to(      pnt.x+DRAW_PORT_CLR          ,       pnt.y-
arm_length*0.125 );
  draw_line_to( pnt.x+a , pnt.y-arm_length*0.666 );
  draw_line_to( pnt.x+a/2.0 , pnt.y-arm_length );
 draw_line(     pnt.x+a,      pnt.y-arm_length*0.666,       pnt.x-
DRAW_PORT_CLR, pnt.y-arm_length*0.125 );
```

```
    draw_line_to(       pnt.x+DRAW_PORT_CLR        ,        pnt.y-
  arm_length*0.125 );
     draw_line_to( pnt.x-a , pnt.y-arm_length*0.666 );
   }
   /* foot */
   else {
     draw_line( pnt.x+a/4.0 , pnt.y+b, pnt.x+a , pnt.y-arm_length );
     draw_line_to( pnt.x-a , pnt.y-arm_length );
     draw_line_to( pnt.x-a/4.0 , pnt.y+b );
     draw_line_to( pnt.x+a/4.0 , pnt.y+b );
   }
 }

 void Mod_EndEff::print() {
   cout << "Eef " << mod_id;
   print_base();
   cout << "torque " << apply_torque;
   cout << "\ttype "<< ee_type << endl;
 }


 /*******************************************          Task_Type
 *****************/

 void Task_Type::print() {
   cout << "Task_Type::print()" << endl;
 }
```

mymath.h

```
/* Nathaniel Rutman


   mymath.h

   Commonly used math functions

   change history:
   02/14/95 created NZR
   02/24/95 added Point NZR
*/

#ifndef _MYMATH_H_
#define _MYMATH_H_

#include <iostream.h>

inline float max(float a, float b) { return (a > b) ? a : b; }
inline float min(float a, float b) { return (a < b) ? a : b; }

inline float asymp(float x) { return (1-1/(x+1)); }


class Point_Type {
public:
  float x,y;
  Point_Type() {
   x = 0.0;
   y = 0.0;
  };
  Point_Type(Point_Type& a) {
   x = a.x;
   y = a.y;
  };
  Point_Type(const float sx, const float sy) {
   x = sx;
   y = sy;
  };
  Point_Type& operator=(Point_Type& a) {
   x = a.x;
   y = a.y;
   return *this;
  };
  Point_Type operator+(Point_Type& a) {
   Point_Type res;
   res.x = x + a.x;
   res.y = y + a.y;
   return res;
  };
  friend ostream& operator<<(ostream& s, Point_Type& a) {
    return s << '(' << a.x <<','<< a.y << ')';
  };
};


#endif
```

stats.h

```
/*
 Nathaniel Rutman 03/20/95

   stats.h

   statistics on search

*/

#include "modclass.h"   // for bool
#include "search_main.h" // for MAXLENGTH

#ifndef _STATS_H_
#define _STATS_H_

enum                                              test_names
{TDUM,TPOWER,TPORTS,TFEET,TSTEP,TSPAN,TWEIGHT,
          TCOST,TLINK,TARM,TMAX};

class Stat_Type {
 private:
  int last_assy_rules;
  int test_hit[TMAX];
  int test_fail[TMAX];
  int pending_test;
 public:
  int initial_mods;
  int reduced_mods;
  int cnt_kit;
  int cnt_kit_rules;
  int good_kits;
  int kits[MAXLENGTH+1];
  int cnt_assy;
  int cnt_assy_rules;
  int assys[MAXLENGTH+1];

  Stat_Type();
  void good_kit(int);
  void test(int);
  void fail();
  void print(int);
  void print_all();
};


#endif
```

stats.C

```
/*
 Nathaniel Rutman 03/20/95

  stats.C

  statistics on search

*/

#include  "stats.h"
#include "search_main.h"  /* constants, get_time */

float combos(int n, int k) {
 /* n different mods, choose k of them
    (n+k-1)!/k!(n-1)!
 */
 int i,j=k;
 float tmp = 1.0;
 for(i=n; i<(n+k); i++) {
  tmp *= i;
  if(j>1)
   tmp /= j–;
 }
 for(i=j; i>1; i–)
  tmp /= i;
 return(tmp);
}

float combos_plus(int n, int k) {
 /* include kits of shorter lengths also */
 float tmp=0;
 for(int i=5; i<=k; i++)   // 5 in min size
  tmp += combos(n,i);
 return tmp;
}

/**********************                                   Stat_Type
*****************************/

Stat_Type::Stat_Type()  {
/* initialize */
 for(int i=0; i<=MAXLENGTH; i++) {
  kits[i]=0;
  assys[i]=0;
 }
 last_assy_rules=0;
 cnt_kit=0;
 cnt_kit_rules=0;
 good_kits=0;
 for(i=0; i<TMAX; i++) {
   test_hit[i]=0;
   test_fail[i]=0;
 }
 pending_test=TDUM;
}

void Stat_Type::good_kit(int l) {
/* got a good kit of length l.  Must be called after assys are made */
 kits[l]++;
 good_kits++;
 assys[l] += cnt_assy_rules-last_assy_rules;
 last_assy_rules = cnt_assy_rules;
}

void Stat_Type::test(int testnum) {
/* new test, assumed passed unless fail() is called */
 test_hit[testnum]++;
 pending_test = testnum;
}
```

```
 void Stat_Type::fail() {
   test_fail[pending_test]++;
  pending_test=TDUM;
 }

 void Stat_Type::print(int l) {
 /* stats for kits with length l, plus running totals */
   cout << "\nStats: " << initial_mods << " modules, length " << l;
  cout << " gives " << combos(initial_mods, l) << " combos.\n";
  cout << "      reduced to " << reduced_mods << " task-useful mods
(";
  cout << combos(reduced_mods, l) << " combos)" << endl;
  cout << "there are " << kits[l] << " good kits, with " << assys[l];
 cout << " assemblies (for an avg " << (float)assys[l]/kits[l] <<
")\n";
  cout << "running totals: searched " << cnt_kit << " kits, ";
 cout << cnt_kit_rules << " valid kit-rules, " << good_kits << "
assembled\n";
  cout << " into " << cnt_assy << " assemblies, of which ";
  cout << cnt_assy_rules << " are valid.\n";
  cout << "Elapsed time: " << get_time() << " seconds.\n\n";
  for(int i=0; i<=TMAX; i++)
    if(test_hit[i]>0) {
      cout << "test "<<i<<" hit "<<test_hit[i]<<" filtered ";
     cout << (test_fail[i]*100.0)/test_hit[i] << "%\n";
   }
}

void Stat_Type::print_all() {
/* called by search_main at the end */
 cout << "\nFinal stats:\n";
 cout << initial_mods << " modules, up to length " <<
MAXLENGTH << " gives ";
 cout << combos_plus(initial_mods, MAXLENGTH) << " total
combos.\n";
  cout << "      reduced to " << reduced_mods << " task-useful mods
(";
  cout << combos_plus(reduced_mods, MAXLENGTH) << " combos)"
 << endl;
  cout << "searched " << cnt_kit << " kits, ";
 cout << cnt_kit_rules << " valid kit-rules, " << good_kits << "
assembled\n";
  cout << " into " << cnt_assy << " assemblies, of which ";
  cout << cnt_assy_rules << " are valid.\n";
  for(int i=1; i<=MAXLENGTH; i++)
    if(kits[i]>0) {
     cout << "Of length " << i;
    cout << ", there are " << kits[i] << " good kits, with " <<
assys[i];
    cout << " assemblies (for an avg " << (float)assys[i]/kits[i] <<
")\n";
   }
  for(i=0; i<TMAX; i++)
    if(test_hit[i]>0) {
      cout << "test "<<i<<" hit "<<test_hit[i]<<" filtered ";
     cout << (test_fail[i]*100.0)/test_hit[i] << "%\n";
   }
  cout << "Total elapsed time: " << get_time() << " seconds.\n\n";
}
```

graphics.h

```
/*
  Nathaniel Rutman

  graphics.h

  C++ to C interface for graphics

  02/22/95 created
*/


#ifndef _GRAPHICS_H_
#define _GRAPHICS_H_


enum                                                          colors
{BLACK=1,WHITE=0,PURPLE=5,GREEN=4,MAROON=10,GREY
=8,PINK=15,RED=2,BLUE=7,ORANGE=11,LT_BLUE=3,BROWN
=12};

extern "C" {
  void draw_start();
  void draw_color(unsigned long);
  void draw_update();
  void draw_clean();
   void draw_line(float,float,float,float);
   void draw_line_to(float,float);
   void draw_circle(float,float,float);
   void draw_rect(float,float,float,float);
   void draw_box(float,float,float,float);
  void draw_string(float,float,char *);
}

#endif
```

graphics.c

```
/*
  Nathaniel Rutman

  graphics.c

  graphics drawing routines.  Coordinate system is cartesian, 0,0 low
left

  02/22/95 created
*/

/* X graphics commands:
   XDrawPoint(my_display,my_drawing,my_context,i,50);
   XDrawLine(my_display,my_drawing,my_context,10,70,40,90);

XDrawRectangle(my_display,my_drawing,my_context,100,100,10
0,100);
   XDrawString(my_display,my_drawing,my_context,110,110,"Hi
Steve",8);
   XFillRectangle(my_display,        my_drawing,        my_context,
200,200,50,50);
  XSetForeground(my_display, my_context, 700);

XFillArc(my_display,my_drawing,my_context,140,140,20,20,0,360
*64);
*/


#include  "service.h"
#include  <string.h>

#define DRW_WIDTH 600
#define DRW_HEIGHT 500
#define ASPECT 0.788            /* aspect ratio */
```

```
/* globals for drawing */
Pixmap my_drawing;              /* buffer to draw into        */
Display *my_display;             /* pointer to X display variable */
GC my_context;                 /* X graphics context          */
XButtonEvent my_event;           /* X Button event */
float MAGNIFY=1000;              /* scale to 1 unit per pixel */
int track_x, track_y;            /* last point draw to */


void draw_clean() {
  XSetForeground(my_display, my_context, 0);
   XFillRectangle(my_display,        my_drawing,        my_context,
0,0,DRW_WIDTH,DRW_HEIGHT);
  XSetForeground(my_display, my_context, 600);
}

void draw_start()
{
  int i,j,k;
  char c;
  char name[26];
  my_drawing                                                         =
start_graphics(DRW_WIDTH,DRW_HEIGHT,&my_display,&m
y_context);
  draw_clean();
  XSetForeground(my_display, my_context, 1);

XDrawString(my_display,my_drawing,my_context,20,10,"Waitin
g for first assembly",26);
  for(i = 20; i<DRW_HEIGHT; i+= 10) {
   j = i/10-2; k = (i+DRW_HEIGHT)/10;
   XSetForeground(my_display, my_context, j);
    sprintf(name,"%d",j);

XDrawString(my_display,my_drawing,my_context,20,i,name,26);
```

```
    XSetForeground(my_display, my_context, k);
    sprintf(name,"%d",k);

XDrawString(my_display,my_drawing,my_context,DRW_WIDT
H/2,i,name,26);
 }
  update_graphics();
}

void draw_color(unsigned long color) {
  XSetForeground(my_display, my_context, color);
}

void draw_update() {
  update_graphics();
}

void draw_line(float x0, float y0, float x1, float y1)
{
  track_x = (int)(x1*MAGNIFY);
  track_y = DRW_HEIGHT-(int)(y1*MAGNIFY*ASPECT);

XDrawLine(my_display,my_drawing,my_context,(int)(x0*MAGN
IFY),DRW_HEIGHT-
(int)(y0*MAGNIFY*ASPECT),track_x,track_y);
}

void draw_line_to(float x1, float y1) {
  int tmp_x = (int)(x1*MAGNIFY);
  int tmp_y = DRW_HEIGHT-(int)(y1*MAGNIFY*ASPECT);

XDrawLine(my_display,my_drawing,my_context,track_x,track_y
,tmp_x,tmp_y);
  track_x = tmp_x;
  track_y = tmp_y;
}

void draw_circle(float x0,float y0,float r) {
  int ulx = (int)((x0-r)*MAGNIFY);
  int uly = DRW_HEIGHT-(int)((y0+r)*MAGNIFY*ASPECT);
  int rad = (int)(2*r*MAGNIFY);

XFillArc(my_display,my_drawing,my_context,ulx,uly,rad,rad*A
SPECT,0,360*64);
}

void draw_rect(float x1, float y1, float x_width, float y_height)
{
  track_x = (int)(x1*MAGNIFY);
  track_y                    =                    DRW_HEIGHT-
(int)((y1+y_height)*MAGNIFY*ASPECT);

XFillRectangle(my_display,my_drawing,my_context,track_x,trac
k_y,(int)(x_width*MAGNIFY),(int)(y_height*MAGNIFY*ASPE
CT));
}

void draw_box(float x1, float y1, float x_width, float y_height)
{
  track_x = (int)(x1*MAGNIFY);
  track_y                    =                    DRW_HEIGHT-
(int)((y1+y_height)*MAGNIFY*ASPECT);

XDrawRectangle(my_display,my_drawing,my_context,track_x,tr
ack_y,(int)(x_width*MAGNIFY),(int)(y_height*MAGNIFY*AS
PECT));
}

void draw_string(float x1, float y1, char *strng) {
  int x = (int)(x1*MAGNIFY);
  int y = DRW_HEIGHT-(int)(y1*MAGNIFY*ASPECT);
```

```
XDrawString(my_display,my_drawing,my_context,x,y,strng,strle
n(strng));
}
```

service.h

```
/*****************************************
  Nathaniel Rutman

  Graphics service routines
  stolen from 1.00

  2/21/95
*****************************************/

#ifndef _SERVICE_C_
#define _SERVICE_C_

#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

Pixmap start_graphics(unsigned int, unsigned int, Display **, GC
*);
void get_mouse_event(XButtonEvent *);
void update_graphics(void);

#endif
```

service.c

```
#include "service.h"

                        /* yes they are all globals! */
Display *display;       /* pointer to the display          */
Window   win;           /* window to be used for graphics     */
int   screen_num;       /* screen number                  */
Window   root;          /* root window of display          */
XSizeHints size_hints;    /* size hint structure for window manager
*/
GC the_GC;              /* graphics context to be used       */
GC copyGC;              /* main GC                    */
XGCValues the_GC_values; /* values in graphics context          */
Screen *screen_point;    /* pointer to screen             */
                    /* Removed FILE declarations           */
Pixmap buff;            /* Pixmap used as drawing buffer      */
XEvent event;           /* structure to hold an XEvent        */


#define INIT_NAME "start_graphics"    /* name of startup module
*/

#define E_NOOPEN   -1            /*   error--unable   to   open   X
connection */
#define E_NOFILE   -2        /* error--unable to open data file  */
#define E_NOEVENTS  -3           /* error--no events in file        */
#define BORDER 2         /* size of window border          */

#define TRUE  1
#define FALSE 0

/*----------------------------------------------------------------------
*/
/*  This routine starts up the X connection, creates and maps a
window,
    creates a Pixmap which gets returned to the caller and generally
    initializes things for the 1.00 student. The display pointer and
the
    graphics context are returned to the user through pointer
arguments */

Pixmap start_graphics(width,height,p_display,p_context)
    unsigned int width;      /* this is the width of window */
    unsigned int height;     /* this is the height of window */
    Display **p_display;      /* address of display pointer(returned)
*/
    GC *p_context;          /* pointer to graphics context(returned) */
{

                        /* Open display and set key variables */

  if((display=XOpenDisplay(NULL)) == NULL)  {
    fprintf(stderr,"%s: can't open display",INIT_NAME);
    exit(E_NOOPEN);
  }
  screen_num = DefaultScreen(display);
  root = DefaultRootWindow(display);
  screen_point = XScreenOfDisplay(display,screen_num);

   SetSizeHints(width,height);

                        /* create the window */
 win                                                              =
XCreateSimpleWindow(display,root,size_hints.x,size_hints.y,
                    size_hints.width, size_hints.height, BORDER,
                    WhitePixel(display,screen_num),
                    BlackPixel(display,screen_num));

                        /* set properties for window manager */
  XSetStandardProperties(display,win,"GraphicsTool","1.00
Problem Sets", None,
                 NULL,0,&size_hints);

                    /* elect events of interest and map window */

 XSelectInput(display,win,ExposureMask | ButtonPressMask | Button
 ReleaseMask);

                    /* setup GC */
  SetupGC();
   XMapWindow(display,win);

                    /* create blanked out Pixmap buffer for user */
  buff = XCreatePixmap(display,win,width,height,
              DefaultDepthOfScreen(screen_point));

  ClearPixmap(width, height);
 XSynchronize(display, TRUE); /* Allow other processes to share
                   CPU              */
  XNextEvent(display, &event);
                  /* load return values */
  *p_display = display;
 *p_context = the_GC;

 return(buff);
}

ClearPixmap(w, h)
int w ,h;
{
 GC gc;

 gc = XCreateGC(display, win, None, &the_GC_values);
 XSetGraphicsExposures(display, gc, False);
 XSetForeground(display, gc, BlackPixel(display, screen_num));
 XFillRectangle(display, buff, gc, 0, 0, w,  h);
}
```

```
                        /* this routine handles XEvents */
                        /* This is changed so that the  */
                        /* function now requires that   */
                        /* a pointer to a XButtonEvent */
                        /* is passed as an argument and */
                        /* loaded with a call to       */
                        /* XNextEvent.                */

void get_mouse_event(XButtonEvent * event)
{                       /* Replaced: 4/91 ROC         */
                        /* XButtonEvent *xb_event=NULL; */
                        /* with XEvent an_event; so    */
                        /* entire event struct can be  */
                        /* used.                      */
 XEvent an_event;
                        /* set up event pointer for mouse */
                        /* events check for XEvents to be */
                        /* handled or returned to caller  */

                        /* Removed update graphics call as   */
                        /* it has nothing to do with the event */
                        /* handler call. It is still called if */
                        /* an expose event occurs.          */
                        /* update_graphics();              */

 short int button = FALSE;    /* control var to skip over non-button
*/
                        /* events.                   */

 while(!button) {            /* Wait till a button is pressed */
  XNextEvent(display, &an_event);  /* Blocks until event occurs
*/

  switch(an_event.type){    /* Select event of interest.        */
  case(ButtonPress):
      case(ButtonRelease):{
        *event = an_event.xbutton; /* load xbutton struct into     */
                          /* callers XButtonEvent struct  */
        button = TRUE;          /* allow exit from loop.       */
      }
     break;

                          /* if window obscured and then exposed */
                          /* redraw contents of buffer to screen */
     case(Expose):{
       update_graphics();
     }
     break;
    default:{
   }
    break;
   }
  }
 }

 void update_graphics()
 {
  XCopyArea(display, buff, win, copyGC, 0, 0,
        size_hints.width,size_hints.height, 0, 0);

  XFlush(display);
}

SetSizeHints(width,height)
int width, height;
{
 size_hints.x = 200;
 size_hints.y = 50;
 size_hints.min_width = width;
 size_hints.min_height = height;
 size_hints.max_width = width;
 size_hints.max_height = height;
 size_hints.width = width;
 size_hints.height = height;
 size_hints.flags = USSize | USPosition | PMinSize | PMaxSize;
}

SetupGC()
{
 the_GC_values.foreground = WhitePixel(display, screen_num);
 the_GC_values.function = GXcopy;  /* GXequiv  is for black on
white */

 the_GC = XCreateGC(display,win,
            (GCFunction | GCForeground),
             &the_GC_values);

 copyGC = XCreateGC(display, win, None, &the_GC_values);

 XSetGraphicsExposures(display, the_GC, False);
 XSetGraphicsExposures(display, copyGC, False);
}
```

makefile

```
#      makefile
#
#     builds target from source files
#     Nathaniel Rutman 10/28/94
#
# If you run out of disk space, try uncommenting the line
#     BINDIR = /usr/tmp
# This will build all binaries (object files and executables) in
/usr/tmp
# and set up symbolic links to these files from your current working
directory
# i.e. the directory containing this makefile.
#
# You can cleanup all binaries and symbolic links by typing
#     make -f <makefilename> clean
# where <makefilename> is the name of this makefile.


PROGS = search

# List of C++ source files.

CPPSRC = modclass.C groupclass.C assyclass.C kit.C assemble.C
mod_desc.C choose_in.C stats.C search_main.C


# List of C source files (if any).

CSRC =  graphics.c service.c

#**************************************************************
*****************
# End of section.
```

```
#**************************************************************
*****************

# Name of the C++ compiler/translator.
# We will use the Cygnus g++ compiler, so be sure to "add cygnus" at
your
# athena% prompt. Also remember to "detach gnu" if you have
attached it before.
# (You do not have to specify the include paths for the C++ header
files; the
# Cygnus g++ compiler is invoked by a script which will take care
of this.)

CPP = /mit/cygnus/${hosttype}bin/g++

# Name of the C compiler.  (See above comments.)

CC = /mit/cygnus/${hosttype}bin/gcc

# Directory in which to maintain binaries.
# This has been set to the current working directory i.e. the
directory from
# which the makefile was invoked.  However, if you run out of disk
space, you
# may want to change it to /usr/tmp.

BINDIR = .
#BINDIR = /usr/tmp

# Macro for setting up symbolic links in the event that ${BINDIR}
is not the
# current working directory.

SETUPLINKS = @ if (test ${BINDIR} != .) then \
        ln -s ${BINDIR}/$@ $@; \
       else \
        break; \
        fi

# Paths for X11 libraries
X_DIR        = /mit/x11
X_LIB        = ${X_DIR}/lib
X_INCLUDE    = ${X_DIR}/include

# Paths for the tcl and tk libraries and header files.

TCLTK_DIR    = /mit/tcl
TCLTK_LIB    = ${TCLTK_DIR}/lib
TCLTK_INCLUDE = ${TCLTK_DIR}/include
COMPAT       = ${TCLTK_DIR}

# List of include file paths required by the compiler.

CPPINCLUDE  =  -I.  -I${TCLTK_INCLUDE}  -I${X_INCLUDE}  -
I${COMPAT}
CINCLUDE    =  -I.  -I${TCLTK_INCLUDE}  -I${X_INCLUDE}  -
I${COMPAT}

# List of library paths required at link time.

LIBDIRS  =  -L${TCLTK_LIB}  -L${X_LIB}

# Compile time options.

CPPFLAGS = ${CPPINCLUDE}
CFLAGS   = ${CINCLUDE}

# Link time options.

LDFLAGS =

# Libraries to be linked.

LDLIBS =  -lm -lX11
# -lm -ltk -ltcl -lX11

# Create lists of object files from the source file lists.

CPPOBJ = ${CPPSRC:.C=.o}
COBJ = ${CSRC:.c=.o}

#--------------------------------------------------------------------
---
#
# Define a list of significant suffixes as well as some suffix rules.
#
#--------------------------------------------------------------------
---

.SUFFIXES:        # Delete the default list of significant suffixes.
.SUFFIXES:  .o .C .c # Add .o .C .c to the current list of significant
suffixes.

# Construct a .o file from a .C file with same name.
# Binaries go in ${BINDIR}.

.C.o:
     @ echo "Building target $@:"
     ${CPP} ${CPPFLAGS} -o ${BINDIR}/$@ -c $<
     ${SETUPLINKS}

# Construct a .o file from a .c file with same name.
# Binaries go in ${BINDIR}.

.c.o:
     @ echo "Building target $@:"
     ${CC} ${CFLAGS} -o ${BINDIR}/$@ -c $<
     ${SETUPLINKS}
```

```
# Construct an executable from a .C file with same name.
# Binaries go in ${BINDIR}.

.C:
      @ echo "Building target $@:"
      ${CPP}    ${CPPFLAGS}    ${LDFLAGS}    ${LIBDIRS}    -o
${BINDIR}/$@  $<  ${LDLIBS}
      ${SETUPLINKS}

# Construct an executable from a .c file with same name.
# Binaries go in ${BINDIR}.

.c:
      @ echo "Building target $@:"
      ${CC}    ${CFLAGS}    ${LDFLAGS}    ${LIBDIRS}    -o
${BINDIR}/$@  $<  ${LDLIBS}
      ${SETUPLINKS}

#------------------------------------------------------------------------
---
#
# Make targets.
#
#------------------------------------------------------------------------
---

# Ensure that all programs have been built by making target "all".

all:  ${PROGS}
      @ echo "Done!"

# Making following target will remove all the object files and
executables
# in your current directory, as well as those in ${BINDIR}.   To
perform this
```

```
# cleanup operation, type
#           make -f <makefilename> clean
# where <makefilename> is the name of this makefile.

clean:
      @ rm -f ${PROGS} *.o
      @ echo "Cleaned up current working directory."
      @ if (test ${BINDIR} != .) then \
      cd ${BINDIR}; rm -f ${PROGS} *.o; \
      echo "Cleaned up ${BINDIR}."; \
      else \
      break; \
      fi

# Build the final executable(s) from the object files.  If the list
${PROGS}
# contains multiple programs, then each program will be assumed to
depend on
# all of the object files in the lists ${CPPOBJ} and ${COBJ}.  (To
avoid this,
# you could list each of the targets separately with its respective
# dependencies, using a format similar to the one used here.)

${PROGS}: ${CPPOBJ} ${COBJ}
      @ echo "Building target $@:"
      cd ${BINDIR}; \
      ${CPP} ${LDFLAGS} ${LIBDIRS} -o $@ ${CPPOBJ} ${COBJ}
${LDLIBS}
      ${SETUPLINKS}

# The following section modifies itself to reflect your file
dependencies.

depend:
      makedepend ${CINCLUDE} ${CSRC} ${CPPSRC}
```

```
# DO NOT DELETE THIS LINE -- make depend depends on it.

graphics.o: service.h /usr/include/stdio.h
/usr/include/sys/feature_tests.h
graphics.o: /mit/x11/include/X11/Xlib.h /usr/include/sys/types.h
graphics.o: /usr/include/sys/machtypes.h
/usr/include/sys/select.h
graphics.o: /usr/include/sys/time.h /usr/include/sys/time.h
graphics.o: /mit/x11/include/X11/X.h
/mit/x11/include/X11/Xfuncproto.h
graphics.o: /mit/x11/include/X11/Xosdefs.h /usr/include/stddef.h
graphics.o: /mit/x11/include/X11/Xutil.h /usr/include/string.h
service.o: service.h /usr/include/stdio.h
/usr/include/sys/feature_tests.h
service.o: /mit/x11/include/X11/Xlib.h /usr/include/sys/types.h
service.o: /usr/include/sys/machtypes.h /usr/include/sys/select.h
service.o: /usr/include/sys/time.h /usr/include/sys/time.h
service.o: /mit/x11/include/X11/X.h
/mit/x11/include/X11/Xfuncproto.h
service.o: /mit/x11/include/X11/Xosdefs.h /usr/include/stddef.h
service.o: /mit/x11/include/X11/Xutil.h
modclass.o: mymath.h modclass.h graphics.h
/usr/include/string.h
modclass.o: /usr/include/sys/feature_tests.h
groupclass.o: groupclass.h modclass.h mymath.h
assyclass.o: assyclass.h modclass.h mymath.h groupclass.h
graphics.h
assyclass.o: /usr/include/stdio.h /usr/include/sys/feature_tests.h
kit.o: mymath.h groupclass.h modclass.h search_main.h
assyclass.h kit.h
kit.o: /usr/include/string.h /usr/include/sys/feature_tests.h
assemble.o: assemble.h assyclass.h modclass.h mymath.h
groupclass.h
assemble.o: graphics.h stats.h search_main.h
mod_desc.o: mod_desc.h modclass.h mymath.h groupclass.h
/usr/include/stdio.h
mod_desc.o: /usr/include/sys/feature_tests.h
choose_in.o: choose.h search_main.h modclass.h mymath.h
assyclass.h
choose_in.o: groupclass.h kit.h stats.h
stats.o: search_main.h modclass.h mymath.h assyclass.h
groupclass.h stats.h
search_main.o: search_main.h modclass.h mymath.h assyclass.h
groupclass.h
search_main.o: mod_desc.h assemble.h kit.h choose.h graphics.h
stats.h
search_main.o: /usr/include/sys/time.h /usr/include/stdio.h
search_main.o: /usr/include/sys/feature_tests.h
```