actions on Systems, Man, and Cybernetics, SMC-22(2):224–241, March/April 1992.

P. Chedmail and E. Ramstein. Robot mechanism synthesis and genetic algorithms. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, pages 3466–3471, 1996.

I. Chen and J. Burdick. Determining task optimal modular robot assembly configurations. In *Proceedings of the 1995 IEEE International Conference on Robotics and Automation*, pages 132–137, 1995.

J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley, 2nd edition, 1989.

S. Farritor, S. Dubowsky, and N. Rutman. On the design of rapidly deployable field robotic systems. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conferences*, 1996.

D. Goldberg and K. Deb. *Foundations of Genetic Algorithms*, chapter. A comparative analysis of selection schemes used in genetic algorithms. Morgan Kaufmann, 1991.

L. Kelmar and P. Khosla. Automatic generation of forward and inverse kinematics for a reconfigurable modular manipulator system. *Journal of Robotic Systems*, 7(4): 599-619, 1990.

O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1):90–98, Spring 1986.

J.-O. Kim. *Task Based Kinematic Design of Robot Manipulators*. PhD thesis, Carnegie Mellon University, 1992.

J.-O. Kim and P. Khosla. Design of space shuttle tile servicing robot: An application of task-based kinematic design. In *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, pages 867–874, 1993.

J. Koza. *Genetic programming: On the programming of computers by means of natural selection*. MIT Press, 1994.

J. Koza, F. Bennet, D. Andre, and M. Keane. Four problems for which a computer program evolved by genetic programming is competitive with human performance. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, 1996.

C. Leger. Automated synthesis and optimization of robot configurations, Ph. D. Thesis Proposal. The Robotics Institute, Carnegie Mellon University, 1997. Available on the WWW from *http://www.frc.ri.cmu.edu/~blah/papers/*.

A. McCrea. Genetic algorithm performance in parametric selection of bridge restoration robot. In *Proceedings of the 14th International Symposium on Automation and Robotics in Construction*, pages 437–441, 1997.

D. Messuri and C. Klein. Automatic body regulation for maintaining stability of a legged vehicle during rough-terrain locomotion. *IEEE Journal of Robotics and Automation*, 1(3):132–141, 1985.

P. Muir. *Modelling and Control of Wheeled Mobile Robots*. PhD thesis, Carnegie Mellon University, Dept. of Electrical and Computer Engineering, 1988.

C. Paredis. *An Agent-Based Approach to the Design of Rapidly Deployable Fault Tolerant Manipulators*. PhD thesis, The Robotics Institute, Carnegie Mellon University, 1996.

G. Roston. *Genetic Methodology for Configuration Design*. PhD thesis, The Robotics Institute, Carnegie Mellon University, 1994.

K. Sims. Evolving virtual creatures. In *1994 Computer Graphics Proceedings*, pages 15–22, 1994.

cating open chains; the method must be able to derive a forward dynamic model (one which computes each link's linear and rotational velocity and acceleration given actuator forces) from a description of a robot's links and joints. One possible implementation would use a non-penetrating rigid body simulation package (Baraff 1996), which would compute the motions of a robot and the contact forces generated by collisions between the robot and objects in its environment. This would simplify interaction between the robot and other objects (such as a robot picking up a payload), since the physical simulation would directly model interactions between the robot's tool and the environment. Such a method could also compute contact forces between a mobile robot's locomotion system and the underlying terrain, which may be useful for detecting tipover conditions and simulating a robot's propulsion system.

Optimizing Non-Kinematic Properties

One of the most important benefits of adding a dynamic model to the simulator is the ability to optimize non-kinematic properties of a robot. It is possible to include actuator size in a module's parameters, but the current system cannot model the effects of actuator size on performance. A dynamic simulator will allow us to model the effects of actuator saturation. If a robot has actuators which are undersized, its accuracy and speed will be inferior to a robot with optimally sized actuators. On the other hand, a robot with oversized actuators will be more expensive, and each oversized actuator may require that other actuators be made larger. An oversized actuator near the end of a manipulator will put greater demands on actuators near the base of the serial chain, thus either degrading performance or requiring other actuators to be enlarged. By modeling the effects of actuator size, the system should be able to optimize actuators in the same way kinematic dimensions are optimized.

Another non-kinematic property that our system can potentially optimize is the cross section of a manipulator's links. In the material handler example, we had to fix the dimensions of several modules' cross sections because the simulator could not model the strength of differently-sized beams. If the simulator could estimate a beam's strength from its dimensions and compute the forces and moments acting on the beam, it would be able to detect failure of the beam. An undersized link would thus cause a robot to fail at its task, while an oversized link would increase the torque required from actuators. We believe that each link's cross section can optimized by extending the simulator to compute beam strength and applied forces and moments.

Other Improvements

Terrain modeling is crucial for realistic simulation of many mobile robot applications. The antenna pointing example used a simple terrain model with sinusoidal profiles for left and right wheel pairs. A more general terrain model is desirable for other applications. Simulating attitude changes as a vehicle traverses terrain is of primary importance; detailed modeling of soil interaction may not be required for many tasks.

Another general area of improvement is motion planning and control. In a kinematic simulation, a planner can predict the outcome of a set of commands with high accuracy. A dynamic simulation introduces uncertainty into planning unless the planner itself uses a dynamic model for prediction. This may prove to be too computationally expensive; a better approach may be to use a kinematic planner, but re-plan when error in plan execution exceeds a threshold.

The motion planner used for computing base motion in the material handler example assumes a planar environment. Free-flying robots and manipulators moving in cluttered workspaces may also require motion planning, but in three dimensions. While many algorithms exist such planning, we require a reasonably efficient approach since tens of thousands of configurations may be evaluated in a single synthesis run. Local, Jacobian-based methods may be sufficient if the workspace contains relatively few obstacles (Khatib 1986).

The current simulator does not prevent self-intersection, and our current controller and planner do nothing to avoid it. This should be remedied in future work. In some cases a link or joint may need to be redesigned to avoid collision with another part of the robot. While this can be done manually, doing so may alter the optimality of a design and may require another run of the synthesis process. It would be desirable for the system to disallow self-intersection so that configurations can be evolved which avoid it.

CONCLUSION

We have developed an extensible framework for robot configuration synthesis. By combining a genetic optimization algorithm with an object-oriented software architecture, we allow new capabilities to be easily added to the system. The system uses a flexible representation for robot configurations that can allow mobile and fixed base robots including robots with multiple or branching manipulators and free-flying robots. A parallel architecture for execution allows heavy computational loads to be distributed across a heterogeneous network of computers. We plan to make several important extensions to the system including dynamic simulation, optimization of non-kinematic properties, and effective optimization of multiple metrics. These additions will bring the system closer to being a practical, general-purpose tool for robot configuration synthesis.

REFERENCES

D. Baraff. *Coriolis Documentation*, 1996.

J. Barraquand, B. Langlois, and J.-C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Trans-*

error. After rewriting the controller to fix the first bug (which was an error in our formulation of the control task), we noticed that the system again seemed to prefer a certain topology. Again, the system produced acceptable designs, but *all* configurations with acceptable pointing error had the same topology (the *x-y* mechanism in figure 6a), and we knew mechanisms of the type in figure 6b should be able to complete the task, though perhaps requiring higher joint velocities. After fixing a simple but serious controller bug, the system did indeed produce *az-el* configurations for some tasks.

This example demonstrates the effect of controller bias: if a controller happens to work best with one type of configuration, then the optimization process will prefer configurations of that type, even though the configurations are not inherently superior. While the bias in our example was caused by programming errors, it is possible for a bug-free controller or planner to be biased towards certain configurations. A example of this is a controller which does not deal well with a manipulator's singularities; the controller will be biased towards configuration which, by chance, do not approach any singularities during task execution.

A similar problem exists with respect to controllers and tasks: a controller may have a bias towards a certain task or metric. For example, the SRI controller implicitly and simultaneously optimizes error and joint velocity. A redundant manipulator's extra degrees of freedom can also be used to optimize a function of the joint values; this can be used for joint limit avoidance, for example. Using the SRI and avoiding joint limits through redundancy is a combination well suited for accurately following trajectories. However, if we are synthesizing a mobile manipulator for a material handling task, stability (avoiding tipover of the robot) could be more important than accuracy. Using a controller that tries to avoid joint limits but ignores stability will cause the stability of some configurations to appear worse than it actually is. This type of controller bias is not as serious, since it affects all configurations; its main impact is that the results of the simulation may underestimate the true capabilities of a robot.

The synthesis process generates a multitude of configurations, and a controller must be able to control them. Thus, we can't depend on having a well-optimized controller which makes each configuration perform at the limits of its capabilities. It is important to be aware of the potential impact that a controller can have on the evaluation of a configuration. Perhaps the best that one can hope for is a controller that preserves relative performance between configurations: if configuration *A* has measurably better performance than configuration *B* when both configurations are controlled by optimal controllers, then a general controller should ideally cause *A* to have better performance than *B* on a simulated task. This would ensure that controller bias has minimal impact on the synthesis process.

Task-Specific Synthesis

The two examples take a *task-specific* approach: the system evaluates each configuration with respect to a certain task, rather than using inherent properties of each configuration. A different approach would be to evaluate a configuration's performance based on task-independent properties such as the size of a manipulator's dextrous workspace, the fraction of the workspace that is close to a singularity, or the turning radius of a mobile base. A general-purpose evaluation can be used in our system by writing a new evaluator object, and may be useful when synthesizing a general-purpose robot whose ultimate applications will vary widely.

The task-specific approach to synthesis is a two-edged sword: it assumes that the simulated task is representative of the application for which the robot will be used. If this assumption is correct, then synthesis can result in a configuration that is optimized for the desired application. But if the task or metrics do not characterize the end use of the robot, then the synthesis process will not produce a well-optimized configuration Thus, the designer must ensure that the task used in simulation accurately reflects the intended use of the robot. If the desired result is a general-purpose robot, the evaluation might consist of a number of varied subtasks in an attempt to exercise many different capabilities of each configuration. For special-purpose robots, the simulated task should be as close as possible to the real task. In both bases, the designer needs to ensure not only that the task is representative of the application, but that the metrics reflect the important aspects of each robot's performance. In summary, the outcome of the system is heavily dependent on the evaluation process specified by the designer.

FUTURE WORK

We plan to pursue a number of improvements and extensions to out current synthesis framework. The first area of improvements lies in optimizing multiple metrics; our approach to this was detailed earlier. Some other areas are:

- Extending the simulator to include dynamics
- Optimizing non-kinematic properties
- Adding a terrain model to the simulator
- Improving motion planning and control

Dynamic modeling

Our current system uses a kinematic simulator: joint velocities and accelerations can have arbitrary values. While this is adequate for many tasks, the inclusion of dynamics into the simulator will allow us to more accurately model a robot's performance on tasks where inertial and working forces are significant compared to the robot's actuator forces. We require a dynamic simulation method that can deal with arbitrary bifur-

handler configuration can complete the task but another cannot, the fact that the latter configuration has lower torque requirements is irrelevant. We should always select the first configuration over the second. This can be implemented by letting the designer write a *configuration decision function* (CDF) which decides which of two configurations is best, based on their metrics. The CDF could be written in a simple interpreted language, allowing changes to be made without requiring recompilation of the population manager. The use of a designer-specified CDF should speed convergence towards an acceptable solution.

Using this method would require that we abandon fitness proportionate selection, in which configurations are selected with frequency directly related to their fitness. Two alternative schemes are *rank-based selection*, in which the entire population is sorted and configurations are selected based on their rank; and *tournament selection*, in which two configurations are chosen at random from the population and the better of the two is reproduced (Goldberg and Deb 1991). Each of these uses direct comparisons between solutions, which is what our designer-specified function provides.

Another way to intelligently optimize multiple metrics is to decide which metric to use for fitness-proportionate selection based on statistical properties of the entire population. For example, if no configurations have completed the entire task, we might want to optimize for task completion. But after all configurations can complete the task, selecting based on task completion alone amounts to selection with uniform probability, which will not produce any improvement.

One way of choosing which metric to use for selection is to use a statistical feature such as variance to determine which metric can best discriminate between good and bad configurations at a given point in time. This turns out not to work very well in our experience. One reason is that different metrics have different ranges--sometimes by orders of magnitude. This is true even though we use adjusted fitness, which is bounded between zero and one. Another reason is that fitness values often do not follow a normal distribution. But the most important cause of failure is that statistics do not capture the desires of the designer. Different metrics have different priorities in the mind of the designer; a standard deviation of x may be meaningless in one metric, but it may be the difference between acceptable and unacceptable designs in another. A better solution might be to let the designer write simple rules which are used for selecting metrics-- a *metric decision function* (MDF). For example, if the average pointing error of a population is greater than the acceptable error, we might select configurations based on pointing error. But once pointing error drops below acceptable levels, we could select based on peak joint torque. The MDF approach would still use fitness-proportionate selection, but should produce acceptable results more quickly and reliably than using a weighted sum of metrics, or choosing a selection metric by statistical methods.

Both of the proposed approaches require the designer to write a selection function. In the first case, the designer specifies the function which decides which of two configurations is best; in the second case, the designer's function decides which metric is best for selection. While we could have the designer write these functions in C++ (the language used by the rest of the system), we feel that it will be easier for the designer to write the rules in a simple, interpreted language so that changes can be made more easily.

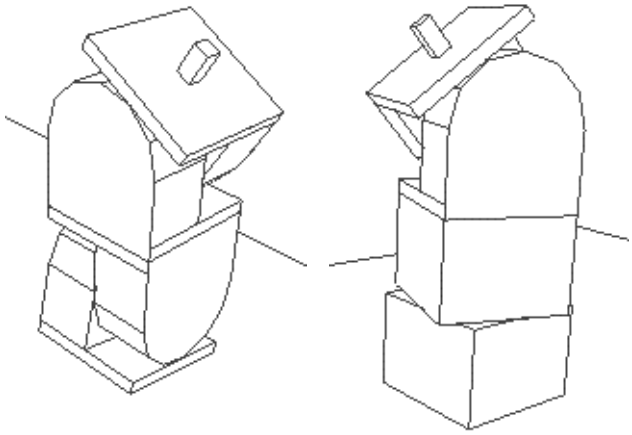
We are not sure of the relative advantages and disadvantages of the two approaches, so we will implement both of them and decide which is more appropriate through experimentation. Fortunately, we can use the same parsing code for both techniques. The common UNIX utilities *lex* and *yacc* can be used to quickly build interpreters and compilers; we expect to use these tools to write the interpreter. By allowing the designer to tell the system how to select configurations in way that expresses the particular needs of a design problem, we hope to improve the ability of the system to robustly generate acceptable configurations.

Dependence on Planning and Control

The inherent qualities of each configuration generated by the synthesis system are not the only factors which determine the final result. If the metrics chosen by the designer do not reflect the important aspects of a robot's performance, then the system will not produce a configuration that is suited for the task at hand. Similarly, if the designer chooses a set of modules that cannot be assembled to produce any acceptable configuration, the program clearly has no chance of generating a good design.

A more serious problem is that the results of each evaluation are dependent on the planning and control code used during simulation. If a controller is poorly designed or is not free of bugs, it may cause an acceptable design to perform poorly (though it is unlikely to make an unacceptable design perform well). Any planning or control algorithms used in the synthesis system must be able to work with thousands or even millions of different robot configurations; it is difficult to guarantee that an algorithm (and more importantly, a specific implementation) works correctly in all cases. Fortunately, it is not crucial that a planner or controller works well with poorly-designed mechanisms, since these would have little chance of being reproduced even with a good planner. But a controller should be able to successfully control any robot that can potentially complete the task at hand in an acceptable manner.

We discovered two bugs in early versions of the controller we used for the antenna pointing mechanism. The controller could acceptably control certain topologies, but not others. The result was that the synthesis process always produced designs with that particular topology--many of which could complete the pointing task with an acceptable level maximum pointing



(a) (b)

Figure 6. The x-y pointing mechanism on the left has better performance at high elevation angles, while the azimuth-elevation mechanism on the right has better performance at low elevations.

pointing error.

We performed several runs of the synthesis process, with differing elevation angles in each (azimuth was not changed, since the rover moves through 360 degrees of heading in each run). For higher receiver elevations (45 through 90 degrees), the Nomad-style chassis and x-y pointing mechanism shown in figure 6a consistently had the best performance. For a lower elevation (20 degrees), the rocker-bogie chassis and azimuth-elevation pointing mechanism shown in figure 6b were consistently best. The reason for the difference in pointing mechanism is that at high elevation angles, large joint velocities are required for small changes in pointing motion for the azimuth-elevation mechanism. The opposite is true at low elevation angles. Additionally, the x-y pointing mechanism requires much higher joint torques at low elevation angles.

It is not clear why the rocker-bogie chassis consistently best for low elevation angles, or why the Nomad chassis performs better for high elevations angles. For both the high- and low-elevation cases, manually changing the base type of the best configuration found by the synthesis program results in slightly lower performance. Thus, it appears that the synthesis process has optimally chosen the base type, although we are not sure why one is better than the other. This is one drawback of automated synthesis--it can sometimes be difficult to understand why one design is better than another. When a human designs a system, each modification is usually a deliberate attempt to remedy a specific shortcoming. With an automated synthesis program, we may not know how a particular design was arrived at; we can only judge its performance.

ANALYSIS

Generality

The two optimization tasks described in the previous section are significantly different from each other, but both are easily accommodated by the object-oriented architecture we have chosen. It may seem that we are belaboring a minor implementation detail, but we believe that this architecture is crucial in enabling a flexible tool to be developed. The reason for this is simple: genetic optimization techniques treat the system being optimized as a black box, which is one of the fundamental principles of object oriented programming.

The modules, trajectory generators, metrics, and core evaluation procedures in the examples are all software objects. All of the metrics share a common interface, as do all of the trajectory generators, modules, and evaluators. The interface allows the internals of each object to change without impacting the rest of the system. For example, the genetic optimization process is identical in the two problems: configurations are sent to evaluators, which return the results of the evaluation in a standardized form. Without the separation provided by these interfaces, the system would require much more extensive modification for each new optimization problem.

In some sense, our system is a toolkit: designers can build a synthesis system to meet their needs from a set of reusable components, adding new components when necessary. The end result is a tool for synthesis. As more and more tasks are addressed, more components are added and the toolkit grows, expanding the range of problems for which an appropriate synthesis tool is quickly available.

Genetic approaches are very flexible in theory; in practice, their flexibility is limited by the representation and evaluation used in the optimization process. We are confident that our framework for synthesis is general and extensible enough to encompass most, if not all, of the automated configuration synthesis tasks addressed in previous research.

Optimizing Multiple Metrics

Perhaps the biggest deficiency in our current system is the way multiple metrics are handled. Real design problems often have multiple conflicting requirements, making trade-offs necessary. At the minimum, there is usually a trade-off between cost and performance. A simple average or weighted sum of different factors is probably not the best way to capture a design's requirements. This is demonstrated by the problems we encountered in each of the examples: for the material handler configuration, we had to scale all of the metrics by the square of the path completion, and for the antenna pointing system we had to select weights which emphasized achieving low pointing error. In both cases, we simply desired to make one metric more important than all of the others. For example, if one material

piece telescoping booms, and an inline revolute joint. As with the wheel diameter parameters, we forced some parameters of the joints to be constant. The cross-section for each joint (and for the single link module) was set to 0.4m x 0.4m. We did this because this simulator doesn't currently model the strength of mechanical components. Thus, the only effect of varying a link or joint's cross section would be to change its weight. This in turn would change the torque requirements for each actuator. The end result would be that the smallest possible cross section would be considered optimal, because the decreased strength of a smaller cross section is not accounted for.

We chose a population size of 500; the initial population was generated from an embryo configuration consisting of a set of pallet forks attached directly to a mobile base. We used the degree-of-freedom filter to allow only configurations with 3 or 4 degrees of freedom in the manipulator, and we limited the maximum mass of a configuration to 15000kg. The best configuration from one run is shown in figure 5. This particular run was halted after generating about 70,000 configurations. This may seem like a lot of configurations, but it is tiny compared to the size of the search space (which contains over 10^{13} configurations in this example). This run occupied approximately 15 Silicon Graphics workstations for 9 hours. Out of 6 runs with slightly varying run time and populations, 4 produced a configuration which could stably reach the entire path. All of these were very similar to the configuration shown in figure 5; the only differences were small variations in some parameter values.

The remaining two runs failed to produce configurations which could complete the task. We believe this is due to way multiple metrics were combined. We used four metrics in all experiments: task completion, maximum joint torque, task execution time, and energy stability (how close a robot comes to tipping over; see (Messuri and Klein 1985) for details). To combine metrics into a single value when selecting configurations for reproduction or deletion, we initially took the average of the 4 metrics' normalized fitness values. This proved inadequate; for example, a configuration which couldn't even reach the first point on the path might have excellent performance in terms of joint torque. To emphasize the importance of task completion, we tried scaling all of the other metrics' adjusted values by the square of the fraction of the task that was completed. This was an ad hoc solution which seemed to help a bit, but it still did not give the desired results all the time. In the runs that failed, the population converged on solutions which could complete about 70% of the task, and which had very good values for joint torque and stability. When the occasional configuration was created which could complete the task, its stability and torque were substantially worse than the rest of the population, and so it was eliminated.

While the evaluation process we used in this example had several limitations, we feel it was useful as a proof of concept and as an initial testbed for experimentation.

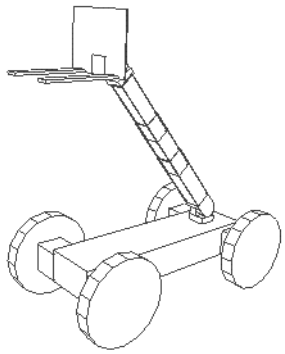


Figure 5. The best configuration from one of the experiments. This configuration was able to complete the task while maximizing stability.

Example 2: An Antenna Pointing System

Another problem to which we have applied the system is the configuration of a rover with an antenna pointing mechanism. In the evaluation, the rover's pitch and roll are computed by a simple sinusoidal terrain model, and the rover's heading rate is constant so that after 30 seconds, the rover has completed a circle. A path object generates commands which try to keep the antenna at a constant azimuth and elevation; these commands are used by the SRI controller to generate joint velocities.

We allowed the optimization process to vary the rover's base type (4-wheel with no suspension, 4-wheel with suspension similar to the Nomad rover (ref?), or 6-wheel rocker-bogie suspension) and dimensions (wheelbase and wheel diameter), and the pointing mechanism. We restricted the pointing mechanism to 2 degrees of freedom. With these constraints, the search space contains about 17 million configurations, though many of these are functionally identical (for example, an elbow joint can be rotated 180 degrees and still have the same function). Each experiment ran until 20,000 configurations had been evaluated; this represents about 0.1% of the search space. We ran the experiments on approximately 20 SGI workstations; run time was approximately 15 minutes. This is much shorter than in the previous example for two reasons: fewer iterations were required to reach an acceptable solution (since the space of possible designs is much smaller), and the time required to evaluate a configuration is less (in the material handler problem, the evaluation time was dominated by path planning computations).

We used five metrics for this problem: path completion (the ability to achieve the desired pointing direction), peak joint velocity, peak joint torque, maximum pointing error, and power consumption for the pointing mechanism. A weighted sum was used to combine multiple metrics into a single value used for selection configurations for reproduction and deletion. Some experimentation was required to determine the best set of weights to use to ensure convergence on a design with low

each metric, updates the sum of the adjusted fitness over the entire population. The population manager then computes the *normalized fitness* for every metric of each configuration. The normalized fitness is just the adjusted fitness divided by the sum of adjusted fitness over the entire population. Probabilistically selecting configurations in proportion to their fitness is called *fitness-proportionate selection*, and one common method of selection in genetic techniques.

When only one metric is used, we use a configuration's adjusted fitness as the probability that the configuration is selected for reproduction. When more than one metric is used (as is the case in most real design problems), we use a weighted sum of each metric's normalized fitness. This is where the constant *c* in the equation above is used--it allows us to bring the standardized fitness for different metrics into a similar range, so that the adjusted fitness for each metric will be similar. For example, if one metric has standardized values ranging from 0 to 10000, the adjusted values will usually be smaller than those for a metric with standardized valued between 0 and 10. If *c* for the latter metric is set to 1000, then the adjusted values will be in the same general range.

We have just presented a high-level view of how the synthesis system works. Next, we will look at two example synthesis tasks which illustrate the operation of the system, and point out some advantages and shortcomings.

EXAMPLES

Example 1: A Material Handling Robot

Material handlers are frequently used in construction sites for transporting heavy loads over moderate terrain. A typical material handler is shown in figure 3. To demonstrate the feasibility of our approach, we applied the synthesis process to the task of configuring a robot for a typical material handling task. (This is discussed in more detail in (Leger 1997))

The task was to approach and pick up a 2500kg payload, drive to another location, and raise the payload to a height of 6m. There are two small obstacles to negotiate. The robot is controlled using the SRI controller while following each of the two paths, and is controlled by a motion planner using numerical potential fields when moving between paths (Barraquand, Langlois, and Latombe 1992). The motion planner is two dimensional: the obstacles are assumed to present a hazard only to the base of the robot, not the manipulator. The payload is modeled simply as a force acting on the end effector; it has no geometric representation. The scenario and a manually-generated configuration are shown in figure 4. The system used nine modules to create configurations: 3 base modules, 4 joint modules, one link, and one tool. The base modules were geometrically identical, but had differing motion capabilities. The Ackerman-steer base uses the front wheels to steer; the four-

Figure 3. A typical material handler

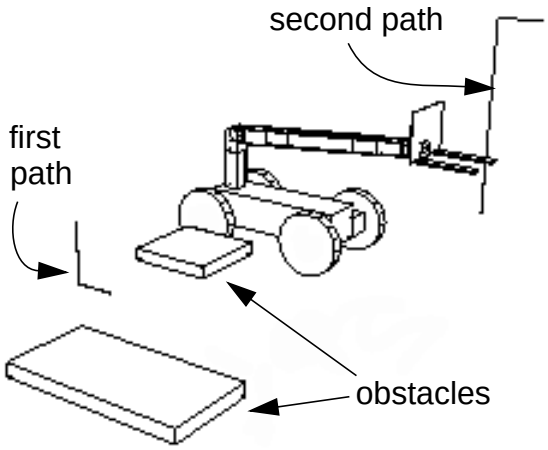


Figure 4. A manually generated configuration executing the material handling task.

wheel-steer base can articulate front and rear wheels independently; and the Mecanum base is an omnidirectional, holonomic base (Muir 1988). Each type of base presents an identical interface to the motion planner, so that the same motion planning code can be used for all three base types. We set the *const-flags* for the base height and wheel diameter parameters, since the evaluation process can't make meaningful decisions about these parameters using the current simulation. For example, the bases are assumed to be moving in a flat plane in the simulation, so varying wheel diameter has no meaningful effect on the outcome of the evaluation.

The joint modules consisted of an elbow joint, 2- and 3-

computers. These tasks evaluate configurations and send the results back to the main process (called the *population manager*), which adds the configurations to the population. The population that is kept during optimization consists entirely of configurations that have been evaluated; the initial population is kept separate, and configurations are moved from the initial population into the "working" population as the results of their evaluation are returned. Whenever a configuration is added to the population, one or two configurations are probabilistically selected for reproduction (based on their performance). The offspring are added to a queue of configurations to be evaluated. To keep the population at a fixed size, configurations are probabilistically removed from the population, with poorly-performing configurations being removed more often than configurations that perform well.

This architecture is significantly different from that used in a standard GA, but is similar to the one used by Paredis (Paredis 1996). A normal GA evaluates all members of a population and then generates a new population all at once. This is inefficient when evaluation is computationally intensive and thus must be distributed across several computers, since many machines will sit idle while the last members of the population are generated. The architecture we use can continually generate and evaluate configurations, thus increasing efficiency.

Evaluating Configurations

The architecture described above relies on evaluator processes to determine the performance of each configuration. Each evaluator task consists of two parts: the core evaluation code which measures a configuration's performance, and a shell which provides the interface between the core and the population manager. This separation allows the core to be modified to suit the task at hand. For example, a rover and an industrial manipulator would be evaluated in vastly different ways. An object-oriented approach allows us to define a standard interface between different parts of the system, so that various pieces can be replaced with problem-specific code without impacting the rest of the system. In the case of the evaluator, the interface is simple: the evaluator takes a configuration and set of metrics as input, and produces performance measures for each of the metrics as output.

In our system, the evaluation is based on simulation. While it is possible to incorporate heuristic evaluations into our system, we feel that simulation is a much more accurate method of evaluation and the benefits of simulation outweigh the increased computational cost. In order to simulate a robot, the robot must be controlled. To make this possible, we have implemented a set of primitives that can be used by an evaluator to control a robot in simulation.

Each configuration can compute a Jacobian for its base and serial chains (Leger 1997). Each configuration is a software object and contains not only the list of modules comprising the robot, but also algorithms to support various operations such as computing the Jacobian.) The Jacobian can be used in conjunction with the Singularity Robust Inverse, or SRI (Kelmar and Khosla 1990) to generate joint velocities which cause the robot's endpoint(s) to follow a desired path in space. Each endpoint of the robot can have a path object associated with it, which generates cartesian-space velocity commands. Again, the path object has a standard interface allowing task-specific paths to be substituted for more general ones. For example, the tip of each leg of a walker, a manipulator doing assembly operations, and a manipulator pointing an antenna all follow paths generated by completely different means, yet each of these paths can be followed by using the SRI controller. Like other objects, the SRI controller can be easily replaced if a particular task requires a different control algorithm.

We are currently using a kinematic simulation: joint accelerations and velocities may be set to arbitrary values. It would be more accurate (and computationally expensive) to use a dynamic simulation in which the robot is controlled by specifying joint torques and forces, but this has not yet been implemented. Even though the simulation is kinematic, joint torques can be estimated using the computed torque method (Craig 1989). These estimates are useful for comparing different configurations, but are not physically accurate for small joint torques since friction is not currently accounted for.

The designer can specify one or more metrics to measure the performance of different configurations. Some metrics, such as task execution time or the fraction of a task completed, are only used at the end of the evaluation; others, such as power or stability (for a mobile robot) are state-dependent and are measured at each time step of the simulation. Each metric can convert its raw value into *standardized fitness*, which is a value ranging from 0 to positive infinity, with zero being best. State-dependent metrics generate a vector of values; to collapse these into a single measurement, the designer can specify that the minimum, maximum, integral, average, or root-mean-square value of the measurements is used. This value is then converted to standardized fitness. In our system, *adjusted fitness* is computed from standardized fitness as follows:

$$a = \frac{1}{1 + c \times s}$$

where a is the adjusted fitness, s is the standardized fitness, and c is a scale factor, which is used to normalize fitness values between different metrics (we will discuss this later).

The adjusted fitness ranges from 0 to 1, with 1 being best. We use the adjusted fitness for two reasons: it is bounded, and it magnifies differences between good configurations. (See (Koza 1994) for a more detailed explanation.)

After evaluating a configuration, each metric computes its standardized and adjusted fitness values and the evaluator sends the results back to the population manager. The population manager enters the configuration into the population and, for

rations, similarly to the way a human embryo receives genetic information from two parents at conception. In our system, there are actually two types of crossover. The first type is called the *module crossover* operator, and works by exchanging subgraphs between two parent configurations to create two new child configurations. A crossover point is chosen in each parent, and two children are created by exchanging modules that occur after the crossover point in each parent. This process is shown graphically for a simple example in figure 2. The other type of crossover operator is *parameter crossover*, which creates two offspring by exchanging parameter information between similar modules in two parent configurations. Two modules of the same type are chosen, one in each parent. A bit string for each parent module is formed; there is an entry in the bit strings for each parameter that is variable (i.e. the parameter's *const-flag* is not set) in *both* modules. A crossover operation is then performed on the bit strings so that parameter information is exchanged between the two configurations.

The other general class of genetic operations is mutation. Mutation plays a small but important role in genetic methods: it prevents the population from becoming completely uniform. As the optimization process progresses, the population improves by converging on a relatively small area of the search space. This necessarily means that the population becomes less diverse; some parts of different configurations may be identical throughout most or all of the population. Mutation introduces small changes which allow the optimization to explore slightly different, and possibly better, configurations. When the population is highly fit (or well adapted to the task at hand), most mutations are likely to be detrimental. However, they can expand the search space beyond that which would be explored by crossover alone, and this can occasionally be beneficial. Mutation is not very useful in the early stages of optimization, since the population is often very diverse.

We have implemented several mutation operators. The *insertion* operator inserts a link or joint module into the configuration. The *deletion* operator deletes a non-terminal module (i.e any module other than the base or end of a serial chain). The *replacement* operator replaces a module with another of similar type. The *parameter mutation* operator randomizes a single parameter value in a configuration. Finally, the *attachment mutation* operator randomizes the twist parameter of an attachment between modules. Both the mutation and crossover preserve parameters and attachments which have their *const-flags* set. We include the *const-flag* in the representation to allow the designer to specify some known properties of the design. For example, if a certain task requires a particular sequence of joint modules for a wrist and a specific end effector, but the rest of the manipulator is undetermined, the designer can create an embryo configuration (from which the initial population is generated) and set the *const-flags* for the wrist modules' parameters and connectors. This will ensure that all configurations that are generated contain the desired wrist and end effector.

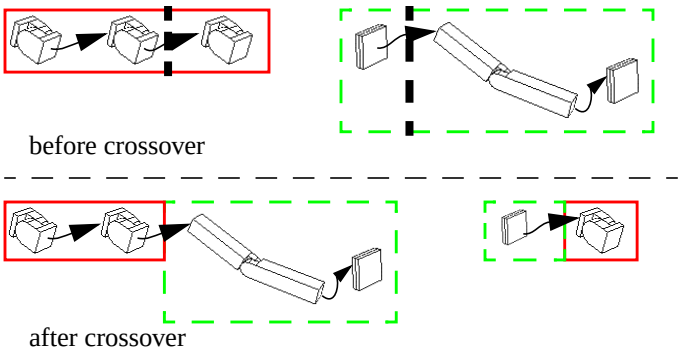


Figure 2. The module crossover operation. Two parent configurations are shown at top, with the crossover points shown as bold dashed lines. After the crossover, the child on the left contains the first two modules of the left parent and the last two of the right. The child on the right contains the first module from the right parent and the last module from the left parent.

The Optimization Process

Using the representation and operators described above, we can generate new configurations from old ones. This allows us to take the first step in the optimization process: generating the initial population. The designer specifies one or more embryo configurations, as mentioned above. These configurations can be trivial: in the case that the designer does not wish to specify any particular modules or parameters, the embryo can simply be an end effector attached directly to a base. In the scenario described in the previous section, the designer would create an embryo having the wrist and tool modules attached to the base.

The initial population starts with the embryo(s). A configuration is selected at random, and one or more genetic operators are applied to generate a new configuration, which is then added to the population. This process continues until the initial population reaches a desired size. ..

Several filters can be used to limit the breadth of the initial population. The designer can specify a minimum and maximum number of degrees of freedom to limit the complexity of the configurations, or a mass filter to limit the size. If the embryos have branching serial chains, an endpoint filter can be used to ensure that each configuration has a certain number of endpoints. (While no possible application of the genetic operators to a single serial chain can create a branching chain, if a configuration has multiple serial chains it is possible for the offspring to have more serial chains, and thus more endpoints, than the parent.) These filters are applied during the entire optimization process, not just while generating the initial population.

After creating the initial population, the optimization process begins. Configurations from the initial population are sent over a network to *evaluator tasks* running on a number other

Representation

The genetic optimization process requires a suitable representation for the objects being optimized--robot configurations, in this case. We represent a robot configuration as a set of connected modules. Each module represents a part of the robot; a module might be a link, a joint, an entire arm, a mobile base, or any other component of a robot. It is important to note that the modules are used only for synthesis. The final robot is not necessarily built from modular parts, though this can be accommodated with no modifications to our system.

Each module may have an arbitrary number of parameters which describe properties of the module. Most parameters describe the various dimensions of a particular module type, but parameters may also be used to describe non-kinematic properties such as the size of an actuator or the thickness of a link's structural members. This allows arbitrary properties of a module to be modified and evaluated. Each parameter has a number of components. The minimum and maximum value can be specified by the designer to limit the variation in a parameter. A *const-flag* can be set to indicate that the parameter's value should not be modified by any genetic operations. The resolution of each parameter's possible values can also be set. Finally, each parameter has a bit-string value (used by genetic operators) and a floating-point value (the actual value of the property represented by the parameter).

Modules may have any number of connectors. These allow multiple modules to be attached to each other to create configurations. Each connector is a coordinate frame attached to the module's geometry. When connecting two modules via their connectors, the two connector's coordinate frames are aligned in a regular way to determine the spatial relationship between the modules. Each connection between modules has two parts: a twist parameter determining the angle of rotation about the connector, and a *const-flag*, indicating whether the connection may be modified by genetic operations.

As mentioned above, configurations are composed of a set of connected modules. Each configuration is represented as a list of modules, with the base module first. Each module can specify a connection to a module that comes later in the list. In more specific terms, the configuration is stored as a topologically sorted, directed acyclic graph. Each node in the graph is a module, and each edge is a connection between modules.

This representation allows a variety of configurations to be represented. A single fixed-base manipulator can be represented as a series of link and joint modules. A rover might be represented by a single base module with parameters for various dimensions. More complex robots, such as a free-flying robot with multiple, branching manipulators or a walking machine with numerous legs, can also be represented. In these cases, symmetry between different arms and legs can be preserved in a straightforward manner: instead of storing a copy of each appendage in the list of modules, a single copy is stored and is

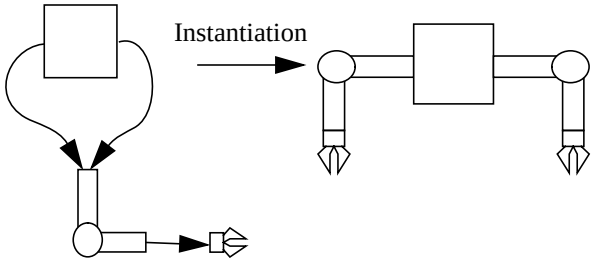


Figure 1. During creation of a configuration's geometry, multiple references to a module result in multiple copies of the subgraph rooted at the module.

referenced multiple times. Thus, a robot with two identical manipulators would have a base and a series of several modules representing one arm. The base module would have two connections to the *same* module--the first module in the arm. When the robot's geometric representation is created for evaluation purposes, two copies of the arm are generated and attached to the two connectors on the base. This process is shown in figure 1.

What sort of configuration can't be represented in this scheme? Robots with closed kinematic chains spanning multiple modules. For example, we cannot represent a robot with two manipulators that are permanently attached at the tips. However, a closed kinematic chain within a single module, such as the linearly-actuated revolute joint commonly found in hydraulic machines, can be represented. In this case, the module must "know" how to ensure its internal consistency.

We make extensive use of object-oriented programming in our system. (Briefly, object-oriented programming is a paradigm wherein software objects consist of both instructions and data, and present an interface to the rest of the system. Objects can thus be treated as black boxes--other parts of the system don't need to know what goes on inside an object.) This allows each module to handle its own special needs while providing a uniform interface to the rest of the system. This means that software for specialized modules, such as a mobile base or the linearly-actuated rotary joint mentioned above, can be incorporated without impacting other parts of the system. The combination of object-oriented programming with a genetic approach to optimization is quite powerful and allows the system to be easily extended in many ways, as we will discuss in a later section.

.Genetic Operators

We have just described the representation of configurations; now we will discuss how this representation is used. The genetic optimization process creates new configurations by using a set of genetic operators, which are analogous to the actions that create new genetic material in nature. The crossover operator creates a new configuration from two parent configu-

RELATED WORK

Existing work in automated synthesis for robots has focused on configuration synthesis rather than detailed electromechanical design. Much of this research addresses the problem of configuration a robot from a set of self-contained modules. Paredis (Paredis 1996) uses a distributed, agent-based genetic algorithm (GA) to create fault tolerant serial chain manipulators from a small inventory of link and revolute joint modules. Each configuration is evaluated kinematically on a satellite retrieval task.

Farritor et al (Farritor, Dubowsky, and Rutman 1996) propose a methodology for modular mobile robots. Much of their work focuses on quickly pruning the space of robot configurations to a manageable number through the use of "kits" (groups of types of parts) and fast heuristic evaluations.

Chen and Burdick (Chen and Burdick 1995) propose a method for determining an optimal configuration of modular links and joints. An Assembly Incidence Matrix (AIM) is used to represent serial chain manipulators; it is a matrix representation of the mechanism graph. A GA is used to determine the optimal assembly based on how many of a small number of task points are reachable by each assembly. Chedmail and Ramstein (Chedmail and Ramstein 1996) use a genetic algorithm to determine the base position and type (one of several manipulators) of a robot to optimize workspace reachability.

McCrea (McCrea 1997) discusses the application of genetic algorithms to the selection of several parameters of a manipulator used in bridge restoration. Direct calculation of the robot's inverse kinematics is possible due to the limited search space: two possible shoulder joints and two possible wrist joints.

Kim and Khosla (Kim 1992), (Kim and Khosla 1993) use a genetic algorithm to synthesize the kinematic parameters of a spatial manipulator with revolute joints and links modeled by line segments. A multi-stage optimization process first chooses the number and orientation of joints, then link lengths and joint angels for a small number of points along a trajectory. Constraints are gradually enforced to ensure continuity between task points, and then a kinematic controller generates joint angles for intermediate task points.

Roston (Roston 1994) uses a GA to create a 2D generalized frame walker, which is evaluated on simulated terrain. Motion plans are evolved concurrently with each mechanism, and a second GA is used to set the parameters for the first GA.

Sims (Sims 1994) uses a genetic approach to evolve virtual creatures and their behaviors. The creatures are represented genetically by directed, cyclic graphs and each link of the creature contains sensors and artificial neurons which create behaviors. The creatures are tested in a dynamic (as opposed to kinematic) environment in which they must swim, hop, and walk towards various goals (such as a light source).

Koza et al (Koza, Bennet, Andre, and Keane 1996) use genetic programming to evolve programs which modify a simple analog circuit to create a variety of filters. Although computationally expensive, the process creates filters with performance superior to those designed by humans. This work is important because it is a practical application of genetic techniques to a complex engineering task.

SYSTEM OVERVIEW

As mentioned above, our system is based on Genetic Programming (GP). Genetic Programming and its simper ancestor the Genetic Algorithm (GA) are based on two biological phenomena: natural selection and sexual recombination. In a GA, possible solutions to a problem are represented as bit strings. The GA maintains a population of bit strings, and reproduces them preferentially based on the quality of the solutions they represent; this is analogous to natural selection. Reproduction is performed by the crossover operator, which produces an offspring solution by combining parts of two parent solutions--this simulates sexual recombination. The quality of solutions in the population improves over time, since good solutions are reproduced more frequently than bad solutions. GP is similar, but represents possible solutions as program trees instead of bit strings. Each program can be directly evaluated simply by executing it.

Genetic approaches do not need to rely on problem-specific features. They have two basic requirements: possible solutions should be represented in a way such that two parents can be combined to yield an offspring, and a method of evaluating solutions must be supplied. The actual *meaning* of a bit string is or program tree is irrelevant, as are the internals of the evaluation method. This makes them more flexible than analytic optimization methods. Genetic approaches have been shown to effectively deal with large, high-dimensional, and non-linear search spaces.

These properties make genetic approaches attractive for synthesis. As long as we can represent configurations, combine them to create new configurations, and somehow evaluate them, a genetic approach should be able to generate a configuration that is suited for a task.

Unfortunately, we must say "should" rather than "will" in the preceding sentence. One drawback to genetic methods is that there is no guarantee that an acceptable solution will be found (assuming one exists). For example, genetic methods are not very good at "finding a needle in a haystack" If there are only a few acceptable solutions in a very large space of possible solutions, and the unacceptable solutions do not improve much as they become similar to an acceptable solution, then the genetic algorithm will be unlikely to find a good solution. Despite the lack of any guarantee of success, the advantages of genetic approaches have enabled them to be successfully used for a variety of applications, and we believe they are well suited for synthesis.

Proceedings of DETC'98
1998 ASME Design Engineering Technical Conference

Automated Synthesis and Optimization of Robot Configurations

Chris Leger
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: blah@cmu.edu

John Bares
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Email: bares@rec.ri.cmu.edu

ABSTRACT

We present an extensible system for synthesizing and optimizing robot configurations. The system uses a flexible representation for robot configurations based on parameterized modules; this allows us to synthesize mobile and fixed-base robots, including robots with multiple or branching manipulators and free-flying robots. We use an optimization algorithm based on genetic programming. A distributed architecture is used to spread heavy computational loads across multiple workstations. We take a task-oriented approach to synthesis in which robots are evaluated on a designer-specified task in simulation; flexible planning and control algorithms are thus required so that a wide variety of robots can be evaluated. Our system's extensibility stems from an object-oriented software architecture that allows new modules, metrics, controllers, and tasks to be easily added. We present two example synthesis tasks: synthesis of a robotic material handler, and synthesis of an antenna pointing system for a mobile robot. We analyze several key issues raised by the experiment and show several important ways in which the system can be extended and improved.

INTRODUCTION

Computer-Aided Design (CAD) tools are now in widespread use in many areas of engineering. This simplest CAD programs function as electronic drafting tools; more capable programs exist for simulating and even synthesizing complex devices. Synthesis tools of varying scope have been available for years to circuit designers; we hope to develop similar tools for the design of robots. Specifically, we are pursuing a methodology for synthesizing and optimizing a robot configuration for a given task. The configuration of a robot can be roughly described as the properties of the robot that are discernible by an observer: the topology and dimensions of the robot's links and joints. In the case of a mobile robot, the configuration includes the robot's locomotion system. Choosing a robot's configuration is the first step in designing a robot; the detailed electromechanical design cannot be done until the configuration is known. We feel that configuration synthesis and optimization is amenable to automation, and that a capable synthesis tool will be of use to the robot designer.

Automated synthesis tools for configuring robots offer many potential benefits. They can:

- quickly generate feasible designs
- explore many more alternative designs than a human designer
- allow rapid design and testing of robot configurations, and
- optimize configurations for complex tasks

A practical synthesis tool for configurations should have several general properties. It should be flexible, so that a wide range of design problems can be addressed. It should be computationally tractable, allowing synthesis of non-trivial configurations. Finally, the tool should produce feasible configurations.

The system we present attempts to address these requirements. Briefly, we use an approach based on Genetic Programming (Koza 1992) to generate and optimize robot configurations. The performance of each configuration is evaluated through simulation or by analytic methods. Flexible planning and control algorithms allow many different configurations to be evaluated in simulation. The approach is extensible in many ways, enabling new tasks, robot types, and methods of evaluation to be incorporated.