Automated Synthesis and Optimization of Robot Configurations: An Evolutionary Approach

Chris Leger

CMU-RI-TR-99-43

Submitted in partial fulfillment of the
requirements for the degree of
Doctorate of Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
December 9, 1999

Abstract

Robot configuration design is hampered by the lack of established, well-known design rules, and designers cannot easily grasp the space of possible designs and the impact of all design variables on a robot's performance. Realistically, a human can only design and evaluate several candidate configurations, though there may be thousands of competitive designs that should be investigated. In contrast, an automated approach to configuration synthesis can create tens of thousands of designs and measure the performance of each one without relying on previous experience or design rules.

This thesis creates Darwin2K, an extensible, automated system for robot configuration synthesis. This research focuses on the development of synthesis capabilities required for many robot design problems: a flexible and effective synthesis algorithm, useful simulation capabilities, appropriate representation of robots and their properties, and the ability to accomodate application-specific synthesis needs. Darwin2K can synthesize and optimize kinematics, dynamics, structural geometry, actuator selection, and task and control parameters for a wide range of robots.

Darwin2K uses an evolutionary algorithm to synthesize robots, and utilizes two new multi-objective selection procedures that are applicable to other evolutionary design domains. The evolutionary algorithm can effectively optimize multiple performance objectives while satisfying multiple performance constraints, and can generate a range of solutions representing different trade-offs between objectives. Darwin2K uses a novel representation for robot configurations called the parameterized module configuration graph, enabling efficient and extensible synthesis of mobile robots, of single, multiple and bifurcating manipulators, and of robots with either modular or monolithic construction.

Task-specific simulation is used to provide the synthesis algorithm with performance measurements for each robot. Darwin2K can automatically derive dynamic equations for each robot it simulates, enabling dynamic simulation to be used during synthesis for the first time. Darwin2K also includes a variety of simulation components, including Jacobian and PID controllers, algorithms for estimating link deflection and for detecting collisions; modules for robot links, joints (including actuator models), tools, and bases (fixed and mobile); and metrics such as task coverage, task completion time, end effector error, actuator saturation, and link deflection. A significant component of the system is its extensible object-oriented software architecture, which allows new simulation capabilities and robot modules to be added without impacting the synthesis algorithm. The architecture also encourages re-use of existing toolkit components, allowing task-specific simulators to be quickly constructed.

Darwin2K's synthesis algorithm, simulation capabilities, and extensible architecture combine to allow synthesis of robots for a wide range of tasks. Results are presented for nearly 150 synthesis experiments for six different applications, including synthesis of a free-flying 22-DOF robot with multiple manipulators and a walking machine for zero-gravity truss walking. The synthesis system and results represent a significant advance in the state-of-the-art in automated synthesis for robotics.

Acknowledgments

Numerous people helped me complete this work, in a variety of direct and indirect ways. My family has provided constant encouragement throughout my academic career; I don't recall any of them ever asking questions like "So, when are you finally going to get a job?" John Bares provided an optimal amount of guidance for my research and was a good source of motivation -- sometimes he seemed more excited about my work than I was. My thesis committee (Manuela Veloso, Rob Ambrose, Pradeep Khosla, and Chris Paredis) kept me on track in the early stages of my thesis and challenged me to think of the bigger picture during my defense; some of their comments have been incorporated into Chapters 5 and 6. Whether he intended to or not, Andrew Johnson acted as a big brother of sorts and is partly to blame for my decision to apply to grad school in the first place.

I'm a big proponent of finding balance in life, and I have my friends to thank for helping me achieve that. My band-mates -- Kevin Lenzo, Matt Siegler, Garth Zeglin, Scott Baugh, Scratchy, and Martin Martin (all of whom have had the Grad Student Experience) -- gave me an much-needed outlet for creative musical energies. Jorgen Pedersen introduced me to rock climbing, which has since become an obsession in my life. Chances are I would have gone nuts without having climbing and my climbing partners to keep me sane, though I suppose most people think climbers are nuts anyway. Jorgen also wrote the RTC communications package used in Darwin2K. John and Lee Bares kindly let me use their woodshop for other stress-relieving extracurricular activities. Jeff Smith, Parag Batavia, and my other classmates also helped me in the pursuit of slack. Finally, I'd like to thank Rosie Jones for providing balance of another sort, for taking care of real-life needs and creature comforts during the final stages of thesis-mania, for proof-reading this document, and for coming up with the name "Darwin2000".

Contents

List of Figures

List of Tables

1 Introduction

Robot configuration design is often performed in an ad hoc manner. The configuration process is hampered by the lack of established, well-known design rules for translating the requirements of a task into a robot configuration, and human designers cannot easily grasp the space of possible designs and the impact of all design variables on a robot's performance. Instead, they must rely on intuition and their own experience with related design problems, and on engineering rules applicable only to parts of the robot such as individual motors or links. Even when modifying an existing design, small changes to one part of the robot can drastically alter the performance of the whole: changing a single Denavit-Hartenberg parameter for a manipulator may reduce the dimensionality of the robot's workspace; increasing the size of one actuator may cause other actuators' torque limits to be exceeded; increasing the mass of one link may require other links to be strengthened; and so on. It is also difficult to determine if a candidate design is suitable for a specific task: Can it perform the required motions? Does it do so without colliding with objects in its environment, or with parts of itself? Can the robot's actuators provide the necessary forces and torques? How much energy does the robot use during the task? How accurately can the robot perform the task? Each of these questions (and usually others) must be answered to determine if a robot can satisfactorily meet a task's needs. Realistically, a human can only design and evaluate several candidate configurations, though there may be thousands of competitive designs that should be investigated. In contrast, a comprehensive and flexible automated approach to configuration synthesis can create tens of thousands of designs and measure the performance of each one without relying on previous experience or design rules.

When designing a robot for a task with many new characteristics, relevant experience in the design team may be limited and may restrict the range of designs that are explored. Frequently, a person or team investigates a small number of concepts based on previous design experiences and selects a few that look promising. This initial brainstorming often consists of qualitative thought experiments and back-of-the-envelope calculations to predict how well each design meets the major requirements of the task: Can each robot perform the basic motions required? Will the robot's kinematics necessitate large actuation forces or be prone to collisions and link interference? Based on the answers to these questions, one of the candidate configurations is selected for simulation and further design. Only after detailed simulation of the robot and its controller do some problems become apparent. Significant effort has now been invested in the design, such as deriving inverse kinematics for the robot, devising an appropriate controller, and searching through catalogs for motors and gearboxes. Much of this effort is robot-specific and is lost if a different design is chosen; therefore, from this point forward the preferred method of addressing design shortcomings is to modify the design rather than to start over with a different configuration. Once the robot is built, further changes may be required due to unforeseen problems or interactions. Significant design iterations are often not practical since much of a project's schedule and resources may be dedicated to creating a single robot; building a second or third robot to remedy design flaws is beyond the scope of many projects. Thus, it is crucial to perform as much analysis and simulation as possible before the robot is built, and it is highly desirable to get it right the first time --

Manual design	Automated synthesis
Few designs are created or evaluated	Generates and evaluates many designs
Designs limited by relevant experience	Uses search instead of experience
Few general design rules	Uses search and directly measures robot performance
Difficult to predict robot performance	Uses simulation to evaluate each design
Hard for a human to grasp design space and impact of variables	Explores the design space by sampling, directly measures effects of variables

Table 1.1: Comparison of manual and automated design

Many of the limitations of manual design can be addressed by an automated synthesis method that uses extensive search and simulation. Automated synthesis can be used to replace or augment the manual configuration design process.

---

since the first time may be the only time. Given the opportunity to build a second robot, the designers can think of things they would have done differently; examples can be found in [Bares93], [Bares99], and [Arocena98]. Many of the changes would require making different decisions in the early stages of the design process, when the focus is on limiting the number of designs as quickly as possible. However, detailed analysis of multiple designs is discouraged given the analysis tools currently available and the tight schedule of many robot design projects. If many different designs could be readily simulated with reasonable fidelity then the decision to commit to a particular design would be better-informed, leading to fewer surprises down the road and providing more confidence that the chosen design will be successful.

The challenges of robot configuration design make automated synthesis methods attractive. Table 1.1 summarizes the limitations of manual design and how they can be addressed by automated synthesis. Synthesis programs frequently evaluate 10^4 to 10^5 different solutions during the search for a feasible or optimal solutions; clearly, evaluating this many designs is impractical for a human designer. Thus, the design space can be more thoroughly searched by automated synthesis tools than by a human designer alone. If accurate simulation and analysis programs are used to evaluate designs during the synthesis process, then the designer can have high confidence that the automatically generated designs will have satisfactory performance for the desired application.

Because synthesis tools can use simulation and search in place of design rules and experience, they can create solutions to complex or poorly understood design problems where relevant experience is lacking in the design team. Similarly, an automated synthesis tool can produce unintuitive yet well-optimized designs that human designers -- with biases towards familiar or easily-understood designs -- would not create. A side effect of evaluating robots in simulation is that it can be easy for a human designer to manually modify solutions and then simulate them with the simulator used by the synthesis program, thereby reducing the cycle time of design iterations. Finally, a synthesis tool can reduce the time and cost required to generate new designs by performing a large part of the design and analysis process for engineered artifacts.

1.1 Problem Statement

There exists a need for a widely-applicable automated synthesis tool for robot configuration design. Compared to current design practices, such a tool would enable more thorough exploration of the design space, and more accurate assessment of candidate configuration performance.

1.1.1 Scope

This thesis addresses the development of a capable and extensible software system for robot configuration synthesis. The goal is to create a synthesis system that is applicable to a wide range of robot configuration design problems due to its extensibility and its core synthesis and analysis capabilities. The system should be expansive in the properties that can be synthesized: kinematics, dynamics, non-kinematic geometry, actuators, and other component selections should be generated by the system, and it should be possible to add new properties to be synthesized as demanded by specific tasks. The synthesis system should be capable of creating a wide range of robots, including mobile robots and manipulators (or combinations thereof) with modular or monolithic construction. The robots created by the systems should be well suited for the task at hand, meeting multiple design requirements while optimizing one or more performance objectives such as speed or system mass. This demands an appropriate synthesis algorithm, one that is capable of efficiently searching a large design space while optimizing multiple objective functions in a manner relevant to the task at hand.

Equally important is the ability of the system to accurately predict the performance of each robot, since inaccurate or incomplete performance predictions give the designer little confidence that the robot will behave as indicated by the synthesizer. In this thesis, task-specific simulation will be used as the primary means of predicting a robot's performance. To accurately assess robot performance with respect to a task's requirements, the system should include a library of commonly required simulation capabilities such as kinematic and dynamic simulation, actuator modeling, collision detection, and controllers for Cartesian and joint-space trajectory following. At the same time, the system should allow new robot components, analysis tools, robot controllers, performance metrics, and task representations to be added and to interact with existing capabilities so that task-relevant evaluation methods can be easily created. The addition of new simulation capabilities or robot components should not affect the synthesis algorithm, as this will limit the applicability of the system.

This thesis will not address detailed electromechanical design and issues such as cable routing, sensor selection and placement, nor will synthesis of control programs be addressed. Since the focus of this thesis is the creation of a broadly-applicable synthesis tool, in-depth investigation of task representations or performance characterization for specific types of robots (e.g. 6-DOF manipulators) will not be performed. The main issues addressed will be the development of core capabilities that are required for many robot design problems: a flexible and effective synthesis algorithm, useful simulation capabilities, appropriate representation of robots and their properties, and the ability to accomodate application-specific synthesis needs. Successfully addressing these issues will result in a practical, widely-applicable synthesis system for robot configuration design.

1.1.2 Research Issues

Optimizing multiple metrics. Real-world design problems inevitably have multiple requirements. The better an optimization method can capture the requirements, the more efficiently it can generate results that are appropriate for the application being addressed. For design, the optimization algorithm should account for the relative importance of different metrics as perceived by the designer (e.g. completing the task is always more important than how much power is used). Optimization of multiple metrics is an open area of research in both evolutionary optimization and in other domains. Finding an efficient method for optimizing multiple metrics is especially important when assessing the performance of each solution is expensive, as it is for robot configuration synthesis.

Architecture for extensibility. There is no single way of describing the requirements for every application, or for measuring the performance of every robot. A rover for planetary exploration has drastically different requirements than a manipulator for dextrous manipulation, which in turn has different requirements than a manipulator used for excavating soil. If a synthesis systems is to be useful for a range of applications, it must allow the use of appropriate representations for tasks and requirements, and must make it easy for these new representations to be incorporated. Any dependencies between the synthesis algorithm and the details of a particular application will require the synthesis algorithm to be modified when addressing new applications. At the same time, application-specific needs must be addressed by the system as a whole if the system is to be capable of generating relevant results. Balancing these two needs has been a driving force for the system architecture.

Equally important to achieving wide applicability is the robot representation used by the system. Allowing new robot topologies and components is necessary to enable the system to address new design problems, as the restriction to a narrow domain (e.g. single serial chain manipulators with fixed bases) limits the class of design problems for which the system can be used. Some synthesis tasks may require mobile robots; some may require manipulators; some may require that a robot be assembled from prefabricated modules. Some tasks may require optimization of actuator selection; some may require optimization of structural geometry; some may require optimization of the robot's controller. Choosing a representation that allows for a diversity of topologies and properties is thus important for a robot synthesis system that will not be limited to a small range of design problems.

Robustness vs. computation time. Creating an effective synthesis algorithm entails finding an appropriate trade-off between robustness and execution time. In machine learning circles, this is known as the exploration versus exploitation trade-off: How should an algorithm balance exploration of new (and potentially better) areas of the search space against exploiting known, promising solutions? Narrowing the search to focus around a promising configuration may quickly lead to a feasible solution, but may make the system more susceptible to local minima in the search space. On the other hand, a more conservative algorithm might unnecessarily explore many solutions because it does not commit to refining a promising design early on. Throughout this work, it has been important to find a balance between exploitation and exploration that gives reasonable robustness while remaining practical in terms of runtime.

Simulation fidelity vs. computation time. The performance of each configuration must be assessed in order to judge its suitability for the task at hand. In this thesis, simulation is used to evaluate each robot's performance so finding an appropriate balance between simulation fidelity and computation time is an important issue. The type of simulation (kinematic or dynamic) is one key factor that has significant impact on synthesis speed and fidelity; others include the approximations made by the simulation tools (such as those for computing link deflections), and the complexity of the representative task used to evaluate each robot's performance. Another basic manifestation of the fidelity vs. computation trade-off is the choice of control algorithms. Local algorithms require orders of magnitude less time to generate motion commands than global algorithms, but are not guaranteed to find optimal paths or avoid collisions. The synthesis algorithm will be biased towards configurations that perform well *with the controller used in simulation*; typically this will mean that robot performance is underestimated if globally-optimal control is not used. One alternative to incurring computational cost of global, deliberative planning is to co-evolve a motion plan along with each robot, though this just shifts the burden of optimal control onto the synthesis algorithm. To ensure a reasonable system runtime, this research has utilized local control algorithms while including some task and control parameters (such as via point location and velocity and acceleration profiles) in the synthesis process.

1.2 Related Work

There has been much prior work in the area of robot design. However, research in *automated* design--that is, the development of software systems which perform a significant part of the synthesis of a robot--has been limited to a handful of systems with widely varying scope and goals. When examining the scope of these, it is important to distinguish between configuration synthesis and configuration optimization. Configuration *synthesis* aims to generate kinematic type and dimensions (at the minimum) for a novel robot, while configuration *optimization* assumes a more limited range of solution types and refines their properties to improve performance. [Bentley99a] makes a distinction between synthesis and optimization for evolutionary methods of automated design in general. He divides evolutionary design approaches into "creative evolutionary design" and "evolutionary design optimization", and quotes [Rosenman97] to describe the continuum between them:

> *The lesser the knowledge about existing relationships between the requirement and the form to satisfy those requirements, the more a design problem tends towards creative design.* [Rosenman97]

Configuration synthesis corresponds to creative evolutionary design: the overall form of the configuration (solution) is not known *a priori* and must be created and optimized by the synthesizer. In contrast, configuration optimization (and evolutionary design optimization) assumes a particular solution form for which optimal parameter values must be chosen. Configuration synthesis and configuration optimization have significantly different scope and require different techniques. Synthesizing kinematic type -- rather than selecting from a small set of kinematic types or performing only parametric optimization -- requires more complex representations for robots and tasks and needs more flexible simulation, control, analysis, and numerical search tools than configuration optimization. When the solution form is limited (as in configuration optimization), analytic or closed-form methods can often be used; however, these methods are too limited in scope when synthesizing kinematic type, dynamics, actuator selection, and additional properties. For example, if an optimization problem consists of selecting one of several manipulators and computing an optimal location for the robot's base, then closed-form inverse kinematics can be used to control each robot in simulation. On the other hand, if the kinematic form of a solution is being synthesized then both synthesis and simulation become more challenging. In the latter case, the design space is larger and simulating each robot requires a more flexible controller that does not require a priori knowledge of a robot's inverse kinematics. As with creative evolutionary design and evolutionary design optimization, there is a continuum of complexity rather than a clear dividing line between robot synthesis and optimization. One rough measure of complexity of a synthesis or optimization method is the extent to which the solution process relies on analytic methods; methods which make use of closed-form equations for simulation (e.g. inverse kinematics) or for determining optimality can do so because the space of possible solutions is sufficiently restricted.

The remainder of this section presents a comprehensive review of research in the automated synthesis and optimization of robot configurations, and also gives a sampling of synthesis and optimization work for domains other than robot configuration.

1.2.1 Robot configuration synthesis and optimization

Previous approaches to robot configuration synthesis can be divided into two categories: modular synthesis, and non-modular or monolithic synthesis. The former group is characterized by constructing robots from fixed modules, mirroring a set of reconfigurable hardware modules from which the actual robot is built. The latter group has synthesized robots which consist of purely kinematic descriptions (i.e. Denavit-Hartenberg (DH) parameters or equivalent) and which are not built from fixed modules. Another interesting distinction is between systems which estimate performance through simulation, and those which use heuristics based on inherent properties of the robot as a measure of performance. Evaluation through simulation is typically much more expensive and often requires many parallel evaluation processes, while heuristic evaluation is faster but is less accurate in predicting robot performance. Nearly all approaches to configuration synthesis have employed evolutionary algorithms of some sort, due to the robustness of such algorithms to local minima and to search spaces that are highly nonlinear and of varying dimension.

Several systems addressed the assembly of a set of fixed modules into robotic systems. The goal of these systems is to take advantage of the inherent reconfigurability and rapid deployment potential of modular robot systems by synthesizing an assembly of modules for a specific task. Paredis synthesized fault-tolerant manipulators and trajectories from a set of fixed modules using a distributed genetic algorithm (GA) [Paredis96]. Each module was characterized by geometric, inertial, and actuator models, and a continuous kinematic simulation was used to evaluate each manipulator. Number of collisions, task completion, tolerance to single-joint failure, and actuator torque capacity were combined into a single objective function to guide optimization. This system used a fairly realistic kinematic simulator which simulated each robot over the entire task. The controller used in this work generated trajectories that were tolerant to any single joint failure, and as with most other systems, obstacle avoidance was not performed by the controller. Instead, the number of collisions during the task (both self-collisions, and collisions with the payload or obstacles) was used as a metric to be minimized by the synthesis algorithm. Because of the high computational cost of simulation, many evaluation tasks were executed in parallel to give reasonable system runtimes. The completeness of module representation (including geometric, inertial, and actuator properties) and fidelity of simulation and control are this work's main contributions to robot synthesis.

[Ambrose94] presented an approach to modular design of planar manipulators. Significantly, this system accounted for manipulator mass, actuator speed and torque capabilities, joint friction and backlash, and link stiffness in addition to other inherent properties of planar modular manipulators. Arm performance was measured as a weighted sum of these metrics, with feasibility criteria for each metric determined by estimates of task requirements. A comparison of predicted arm performance and measurements from the actual physical assembly were also presented. While this approach worked well for planar manipulators and optimized many important non-kinematic robot properties, the effectiveness of using only inherent robot properties for evaluation is questionable for more complex, three-dimensional tasks in which obstacle avoidance, gravity compensation, and dextrous manipulation are important.

Farritor, Rutman, and Cole demonstrated a system for synthesizing modular walking robots for an inspection task [Farritor96a], [Farritor96b], [Farritor98], [Rutman95], [Cole93]. Each module included information on cost, size, mass, and capability (maximum force or torque for actuators and energy capacity for power modules). Rutman developed hierarchical selection procedures to quickly reduce the search space of modules, and evaluated assemblies with several simple heuristics such as leg length and power consumption (based on the power supply and number of joint modules). Rutman's work was restricted to bilaterally-symmetric walkers and heuristic evaluation of each robot in 2D. Farritor replaced Rutman's final exhaustive search phase with a genetic algorithm and manually selected several promising designs from the output of the GA to be further evaluated. The candidate designs were then evaluated on the task at discrete time steps (corresponding to different phases in the robots' gaits) with a manually-generated, generic motion plan. Another genetic algorithm then generated motion plans for each robot from a set of motion control software modules, demonstrating improved performance over the initial generic plan (detailed in [Cole93] and [Farritor98]). The automated design procedure was much faster than other approaches that use simulation for evaluation, although the fitness values computed by the genetic algorithm were not good predictors of performance. Farritor's system required more manual intervention than other approaches when filtering synthesized designs and did not focus on accurately predicting performance during the automated design phase; however, the system required significantly less time to create configurations than approaches using simulation to measure candidate performance. Additionally, the evolutionary synthesis of motion plans was a significant part of the system and was demonstrated for both the synthesized robot and an actual mobile robot. On the whole, this work is distinguished in that it addressed synthesis of a mobile robot and synthesized effective motion plans for the robot and task.

Three systems demonstrated synthesis of assemblies of joint and link modules, using weighted sums of kinematic performance metrics for their objective functions [Chen95], [Chocron97], [Han97]. All three used reachability and manipulability as part of the objective function, and Chocron added a metric measuring the distance from each link to a number of spherical obstacles. Though presented in the context of design for modular robots, both Chocron and Han allowed the length of link modules to vary. Han presents a two-step method for modular synthesis of non-redundant manipulators with revolute joints: an analytic procedure for choosing the kinematic type based on the desired endpoint trajectory, and a genetic algorithm to determine link lengths that maximize manipulability during simulation along the trajectory. The actual physical link modules can be manually adjusted so that their lengths match the computed values. Chocron uses two genetic algorithms: the top-level GA synthesizes an assembly of link and joint modules, and the lower-level GA optimizes joint positions for several discrete task points for each robot generated by the top-level GA. This system thus evolved discrete poses for each manipulator, rather than continuously simulating the robot as it performed a task. All three of these systems only synthesized kinematic properties (Denavit-Hartenberg parameters), though the modules in Chen's system contained representations of non-kinematic module geometry (i.e. 3-D polygonal representations, instead of simply modeling links with line segments).

Other work has focused on the synthesis of non-modular manipulators, though all of it has been concerned only with kinematic properties and does not address actuator selection or synthesis of structural and dynamic properties. Kim explored task-based kinematic synthesis of manipulators [Kim93]. This system generated Denavit-Hartenberg parameters, joint limits, joint positions, and the base location for fixed-base manipulators. This system used a genetic algorithm with multiple subpopulations for each task point (up to 7 task points were used in design problems) and gradually enforced design and task constraints so that the multiple populations converged on a single solution. As with other systems, a weighted sum of metrics was used with manipulability as the sole objective function and other metrics (e.g. task reachability and observance of joint limits) as constraints. A kinematic type generation procedure iterated over the different kinematic types for a given number of degrees of freedom, with the multipopulation GA performing optimization of link lengths and other variables within each kinematic type. The number of DOFs was fixed by the designer for each synthesis task, and the kinematic type was restricted for more complex design problems: the orientation of the first joint was fixed, as was the kinematic type of the final 3 DOFs. Significant contributions of this work include the use of continuous kinematic simulation in evaluation, the synthesis of non-modular manipulators, and the synthesis of base pose and joint limits for manipulators.

One application-specific approach is described in [McCrea97]. This system addressed the kinematic synthesis of a 6-DOF manipulator for a bridge restoration task using a genetic algorithm. The algorithm operated in two phases. The first phase selected one of two kinematic types for the first three DOFs (3 revolute joints [RRR], or 2 revolute joints followed by a prismatic joint [RRP]), as well as the location of the elbow along the arm for the RRR configuration and the optimal movement sectors for each joint. Total arm length was fixed for each configuration, as a "unit workspace" was used. In this phase, number of collisions and percent coverage were used as metrics. The second phase selected one of two wrist types and generated velocity and acceleration parameters for each joint, using productivity and dexterity as metrics. As no actuators or dynamic properties were synthesized, the acceleration and velocity parameters determined by the system did not effectively predict the productivity of the robot. Closed-form inverse kinematics were used due to the restricted kinematic types.

Chedmail and Ramstein used a GA to synthesize overall length and one of two kinematic types (two rotary joints, or one rotary and one prismatic) for a 2DOF planar manipulator, based on path completion and number of collisions with obstacles [Chedmail96]. They also applied the GA to the selection of one of four commercial manipulators and determination of location for the manipulator's fixed base which allowed a path to be followed without collisions.

Paredis and Khosla also explored kinematic synthesis of non-modular robots using numerical optimization to explore a subspace of *kinematic space* (D-H parameters, joint angles, and task points) [Paredis93]. In this approach, manipulators are represented as D-H parameters, with links modeled as line segments. Simulated annealing, random line search, and random sampling were used to find D-H parameters. Generalized inverse kinematics determined joint angles for each task point and set of D-H parameters. Reachability, collision-free motion, and joint limits constrained the search, as did the requirement that the last 3 joint axes intersect at a point (which also allowed the use of a simpler form of inverse kinematics).

Roston's thesis explored the use of genetic algorithms in several design problems: designing planar linkages for a desired motion; 1-dimensional frame walkers for traversing 1-D stepping stone terrain; and generating control programs for the frame walkers.

Katragadda developed Synergy, a general software framework for robot design [Katragadda97]. Synergy is built around a design spreadsheet that allows the designer and several optimization algorithms to modify designs, while using multiple simultaneous simulations to evaluate different dimensions of performance (such as mechanical, electrical, and thermal). This approach was able to perform parametric optimization and component selection, as well as generate simple controllers. Because of Synergy's framework and its ability to use external modeling programs, Synergy was able to simulate a wide range of properties. Additionally, Synergy provided the designer with insightful causal relationships between components--for example, using a faster CPU on a lunar rover would require more mass to be dedicated to power and heat dissipation, which in turn increase the payload requirements on launch vehicle and rockets used for descent to the lunar surface. However, due to its breadth in systemic optimization, Synergy's abilities to synthesize robot geometry and generate novel configurations were limited compared to other approaches to robot synthesis.

In addition to the systems outlined above that focus on configuration synthesis, there have been several systems or methods for robot optimization in narrow problem domains. Rastegar, Liu, and Mattice developed an approach to determine the link lengths, diameters, and control gains for a two-link planar manipulator with revolute joints to optimize accuracy for high-speed trajectory following [Rastegar98]. Moon and Kota present a kinematic synthesis method for reconfigurable machine tools, in which the kinematic requirements of a machining task are matched by an assembly of machine tool modules [Moon98]. Other robot design approaches for specific problems can be found in [Au92], [Park89], and [Tsai85]. The systematic design of wheeled rovers for specific terrain characteristics has also been addressed, in [Apostolopoulus96].

1.2.2 Synthesis and optimization in other fields

Much work has been done in synthesis tools for fields other than robotics. Synthesis tools have been particularly effective in digital VLSI circuit design, for example ([Brayton90], [Kuh90]). Several overviews of synthesis and analysis tools are available; see [Katragadda97] for a broad discussion of design and synthesis tools. [Bentley99a] provides an overview of the use of evolutionary algorithms for a broad range of design problems. Among these are GADES [Bentley99b] which creates non-articulated polyhedral objects from three-dimensional primitives. This system was applied to the synthesis of optical prisms, coffee tables, hospital floor plans, boat hulls, and car bodies. [Funes99] presents a system that evolves Lego structures to support masses or withstand forces, such as simple bridges, scaffolds, and cranes. The use of a genetic algorithm to design flywheels is presented in [Eby99].

Sims has applied genetic techniques to create, among other things, virtual creatures which interact in a dynamic simulation [Sims94]. These creatures are represented as directed graphs, which are instantiated as articulated connections of rigid bodies with embedded control laws and actuators at each connection. The creatures also included joint angle, contact, and light sensors, as well as a brain that contained control laws evolved along with the physical description. Sims used the algorithm to generate creatures for several scenarios: swimming, walking, jumping, following or fleeing light, and competing with another creature to grasp a block. This work generated impressive (and entertaining) results, though the constraints on performance and form of solution were much more relaxed than in robot synthesis problems.

Koza pioneered Genetic Programming (GP), the use of evolutionary techniques to synthesize arbitrary computer programs [Koza92], [Koza94]. Much like robot synthesis approaches where a desired task is specified, a desired output for a program is specified and an evolutionary algorithm generates program trees (in the form of LISP expressions) which are optimized to produce a desired output or behavior. GP uses a population as in a GA, but uses special operators which work with program trees. Examples of problems solved by GP include function approximation, control of a simple planar manipulator, and navigation algorithms for planar mobile robots. GP has also been applied to the synthesis of high-performance analog filter circuits [Koza96]. In this case, GP is used to evolve programs which modify an initial kernel solution, rather than evolving the solution directly. The resulting circuits are competitive with human-generated designs, and include evolved subcircuits that are well known in circuit design.

Numerous analysis, optimization, and synthesis tools have been developed for digital circuit design. These tools range from the simplification of boolean logic circuits, to behavioral simulation of digital circuits [Cadence94], placement of transistors in VLSI layout [Kuh90], automatic netlist generation for VLSI layouts [Synopsis94] and their subsequent simulation, and generation of test cases to verify correct operation [Sunrise94]. The layout of chips and connections on printed circuit boards has also been addressed [Pads99]. There has also been significant work in analog circuit synthesis. Examples include [Ochotta96] and [Krasnicki99]. The work by Krasnicki et al uses a hybrid genetic/simulated annealing algorithm for optimization, and is similar to Paredis' system and to the system described in this thesis in that it uses multiple distributed processes for evaluation. It uses high-fidelity simulation for evaluating candidate designs and distributes the cost of evaluation over a network of workstations.

1.2.3 Limitations of Previous Work

Previous work has demonstrated the feasibility of automated configuration synthesis for various restricted classes of robots, and has shown that evolutionary algorithms can effectively perform configuration synthesis. However, several limitations are evident in the existing body of research in robot synthesis. Each system to date has specifically addressed the synthesis of either manipulators or mobile robots. The only systems to perform any non-kinematic synthesis were those using fixed modules. The systems for non-modular robots are purely kinematic and have all modeled manipulators as joints connected by zero-thickness line segments. The strictly modular systems, while able to represent non-kinematic properties, are by definition not able to independently vary non-kinematic properties such as actuators or link structure, thus leading to suboptimal solutions. Previous systems have not addressed analysis and simulation needs such as dynamic simulation and estimation of link deformation due to loads, thus inhibiting the ability of these systems to meaningfully optimize actuator selection, link structural geometry, and inertial properties. Most synthesis systems have addressed fairly specific classes of design problems (e.g. kinematic synthesis with coverage and collision-free motion requirements), and none have been aimed at creating an extensible, general-purpose robot configuration synthesis system. Many approaches have made gross simplifications to the analysis problem in order to reduce computation time, while not taking advantage of the fact that genetic approaches are inherently parallelizable, and that the availability of CPU power will most likely continue to increase in the future. These simplifications affect the quality of the synthesized result and the accuracy of the prediction of the robot's performance, thus limiting the designer's confidence in the design. Though the time/quality trade-off must be made somewhere, it seems likely that giving precedence to synthesis quality rather than time will yield results that are more useful to the robot designer. In summary, these shortcomings have limited the practicality and applicability of previous systems, and must be remedied if an effective, broadly-relevant synthesis system is to be created.

1.3 Approach

This thesis creates Darwin2K, an extensible automated system for robot configuration synthesis. The synthesis process is based on an evolutionary algorithm: the current set of solutions are represented as a *population* of designs, and new solutions are created by selecting prior solutions based on their performance and combining or modifying them via several genetic operators. The performance of each solution is assessed through simulation on the prescribed task. Since the evaluation of each candidate design is independent of the evaluation of any other design, many evaluations are carried out in parallel over a network of heterogeneous computers to reduce the time required for synthesis (Figure 1.1).

Each robot configuration is represented as an assembly of parameterized modules. Each module represents part of a robot and is a self-contained software object (Figure 1.2). Modules have parameters describing arbitrary properties--for example, a parameter might represent dimension (kinematic or structural), a discrete component selection, or a task or control parameter. Modules may vary in complexity from a single link, to a joint module, to a manipulator, to an entire mobile robot. Each type of module interprets its parameters internally, rather than having a centralized function to interpret parameters for every type of module. Because of this, it is possible to add new modules with new types of properties without requiring the synthesizer to be changed. On the other hand, if a single centralized function were used, then this function would have to be modified whenever a new type of module was needed.

Robot configurations are constructed by connecting modules together in a directed, acyclic graph. This *parameterized module configuration graph* (PMCG) allows the synthesis algorithm to generate diverse robot topologies and combine parts of different configurations. The PMCG also allows subassemblies of the configuration to be duplicated, so that a walking robot could contain a single leg representation that is used multiple times to preserve symmetry. Figure 1.3 shows a simple example of how multiple references to a subgraph are duplicated when the PMCG is instantiated into a description of the robot's links and joints. The primary limitation of this representation is that it restricts the ways in which parallel mechanisms can be created; this will be discussed in Chapter 2. The use of the parameterized module configuration graph as a representation is the main way in which Darwin2K is specialized for robot synthesis; this representation is not

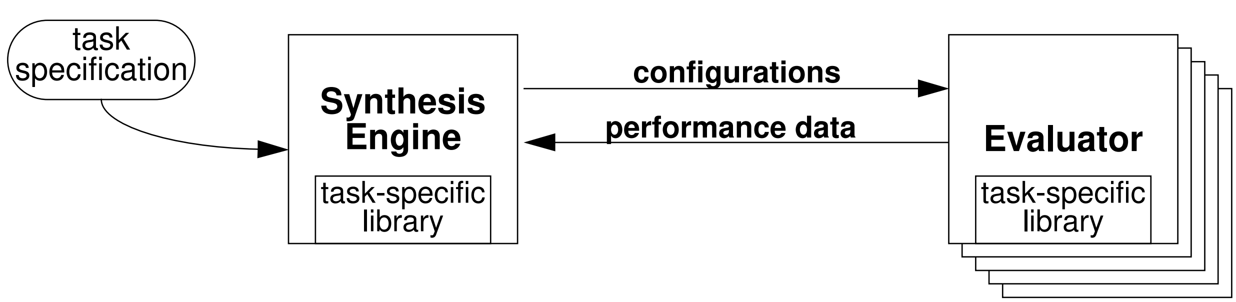
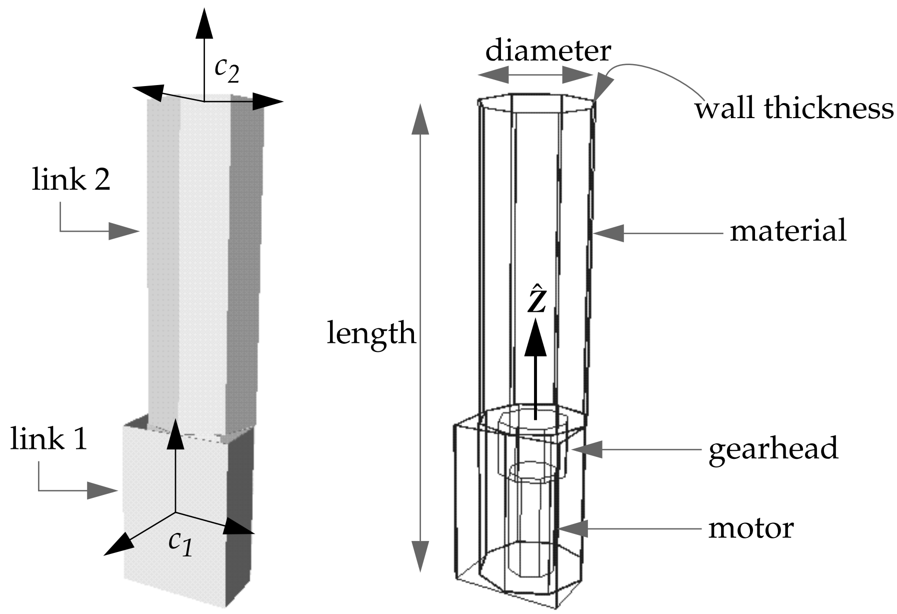


Figure 1.1: Distributed synthesis architecture

Since the evaluation of each robot is independent and evaluation is the main bottleneck in the synthesis process, many evaluation tasks are distributed over a network of workstations to provide performance measurements to the synthesis algorithm.

appropriate for other types of design synthesis such as circuit synthesis or program generation.

The synthesis process must measure the performance of candidate designs; these measurements are computed by evaluating each configuration in simulation. The simulation is task-specific, with performance metrics dictated by the task's requirements. Darwin2K contains a variety of simulation components, including kinematic and dynamic simulators, several controllers, algorithms for computing joint torque and link deflec-



Parameters:
- motor selection
- gearhead selection
- material selection
- tube outer diameter
- tube wall thickness
- overall length

Figure 1.2: Sample parameterized module

A `rightAngleJoint` module is shown here with a list of its parameters. The first two parameters select a motor and gearbox for the module's single degree of freedom (whose joint axis is labelled $\hat{Z}$); one parameter chooses a material (e.g. a particular aluminum alloy) for the module, and the remaining parameters determine the module's size. Each of the parameters can be individually varied by the synthesizer. The module's two connectors are labelled c_1 and c_2, respectively.

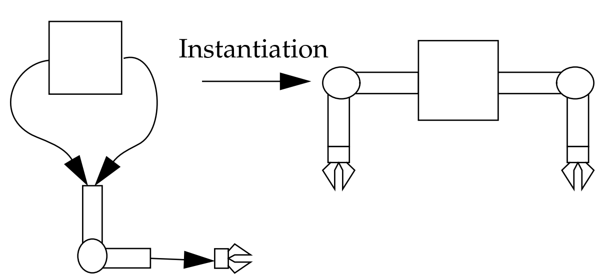


Figure 1.3: Schematic view of configuration graph showing symmetry

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

tion and for detecting collisions; modules for robot links, joints (including motors and gearboxes), tools, and bases (fixed and mobile); and metrics such as task coverage, task completion time, end effector error, actuator saturation, and link deflection. The software objects used in the evaluation process are implemented in a common framework that allows new types of simulation and analysis objects to be added while maximizing reuse of existing components. Entirely new methods of simulation (for example, an underwater robot might require a dynamic simulator which accounts for hydrodynamic forces) can be added, as can new simulation components such as controllers.

The extensible nature of the synthesis framework is a significant contribution of this thesis. Extensibility is dependent to a large degree on the isolation of the details of task and robot from the synthesizer: the synthesizer tells each evaluation task the metrics to use and the robot to evaluate, and the evaluation task returns performance data for the robot. The synthesizer does not depend on the internals of the evaluation process; only the results of evaluation are relevant to the synthesizer. Similarly, the genetic operators used by the synthesizer do not need to know the details of each module, such as what each parameter means. This simple interface allows new module types to be added without requiring the synthesizer to be changed in any way. Combined with the synthesizer and the toolkit of simulation capabilities, Darwin2K's extensible architecture makes it possible to effectively synthesize task-specific robots for a wide range of applications.

1.4 Contributions

The framework and implementation developed in this thesis addresses the limitations outlined in Section 1.2.3, resulting in a synthesis system significantly more capable than previous approaches (as summarized in Table 1.2). Briefly, the contributions of this thesis are:

- An extensible framework for robot configuration synthesis and optimization, allowing the addition of new robot properties and components, task representations and requirements, and simulation and analysis capabilities without impacting the synthesis process or architecture;
- a practical software toolkit for synthesis, including a library of robot components and modules, and simulation, control, and analysis algorithms;
- a new representation for robot configurations, allowing representation and synthesis of a wide range of robots and properties, and allowing new robot components and properties to be optimized without requiring modification of the synthesizer;
- new methods for optimizing multiple metrics in a task-relevant way, including two algorithms that are applicable to other evolutionary design domains;
- novel analysis and optimization capabilities in automated robot synthesis, such as dynamic simulation and estimation of link deflection;

- a demonstration of the solution of configuration synthesis problems of a complexity and scope exceeding previous work. This includes synthesis and optimization of robot kinematics, dynamics, link structure, actuators, task parameters, and base pose for manipulators and mobile robots.

These contributions constitute a significant advancement of the state of the art in robot synthesis tools.

Table 1.2: Comparison of Darwin2K to previous robot synthesis systems. Perhaps most significant is the fact that the only prior systems to have optimized (or even represented) components or non-kinematic geometry were those with fixed modules.

	modular design (fixed modules)	non-modular design	mobile robots	manipulators	non-planar	prismatic joints	inertial & structural properties	actuators & components	multiple/branching manipulators	arbitrary, independent component properties	dynamic simulation	task or control optimization
Darwin2K	●	●	●	●	●	●	●	●	●	●	●	●
Paredis[96]	●			●	●		❍	●				
Chen	●			●	●				❍			
Farritor	●		●		●	●	❍	●	❍			●
Ambrose	●			●			●	●				
Chocron	●			●	●	●						●
Han	●			●	●		❍	❍				
Kim		●		●	●							●
McCrea		●		●	●							
Paredis[93]		●		●	●							
Chedmail		●		●				❍				
Roston		●	●									●

❍ - included in representation

● - included in representation and optimized by system

1.5 Dissertation Overview

This chapter presented relevant background in automated synthesis and provided an overview of the thesis. Chapter 2 discusses representation and architecture issues, giving details of the robot representation, interfaces between the synthesizer and evaluation algorithms, and software architecture. The system's extensibility is a direct result of the topics discussed in Chapter 2.

The synthesis algorithm is described in Chapter 3. This includes the genetic operators, criteria for selecting configurations for reproduction and deletion, the use of performance metrics by the synthesizer, synthesis for multiple task constraints and objective functions, and an 'elitist' strategy for multiple prioritized metrics. Chapter 4 details the evaluation process and describes the simulation and analysis algorithms and metrics in the synthesis toolkit. Chapter 5 presents several synthesis experiments, demonstrating the capabilities of the system and providing some insight into factors that affect the synthesis process. Chapter 6 presents the conclusions and contributions of the thesis. The appendices present some implementation details of various aspects of the system. The reader is encouraged to make use of the Glossary as well.

2 Representation and Architecture

The goal of this thesis is to build a practical, widely-applicable synthesis tool for robotics. The applicability of any robot synthesis method is determined primarily by the representations used for robots and tasks. Previous synthesis work has not focused on the development of extensible, general-purpose systems, and has thus been limited in applicability by representation and extensibility. In previous systems, significant aspects of the robot or task were hard-coded into the synthesis algorithm, making it difficult to change those aspects as required by different tasks. To achieve significant applicability and generality, a robot synthesis system should have few, if any, dependencies between the synthesis algorithm and the robot and task representations. Specifically, a synthesizer should not rely on rules, representation, or operations that are specific to a particular type of robot, or particular components, because this limits the synthesizer to those robot types or components. Similarly, the synthesizer should not rely on knowledge specific to a particular task, as this requires the synthesizer to be changed should a different task be desired.

This chapter describes two aspects of Darwin2K which provide the system's applicability and extensibility: the Parameterized Module Configuration Graph representation, from which robots are built; and the extensible software architecture, which allows new tasks and analysis tools to be added without impacting the synthesis algorithm. These features make Darwin2K significantly more flexible than previous synthesis systems.

2.1 Robot representation

Darwin2K uses a representation for robots called the Parameterized Module Configuration Graph (PMCG), first presented in [Leger97]. As the name implies, robots are composed of modules, each of which may contain parameters describing properties of the module. These modules are connected to each other in a *configuration graph*, which determines the topology of the robot. This section motivates the use of parameterized modules, discusses their implementation, and then describes the configuration graph. Finally, the PMCG is compared to previous representations used in robot synthesis, and its limitations are discussed.

Like previous robot synthesis systems, Darwin2K uses an evolutionary algorithm (EA). Evolutionary algorithms are a class of optimization algorithms based on the principles of natural selection (or survival of the fittest) and sexual reproduction (creating a new artifact by combining descriptions of two other artifacts). Genetic Algorithms (GAs) [Holland75] are the most common type of EAs; Genetic Programming (GP) [Koza94] is another oft-used class. EAs require a way of representing a solution (in this case, a robot configuration) such that parts and properties of different solutions may be interchanged; this symbolic representation is called a *genotype*. At the highest level, EAs operate by measuring the *fitness* (performance) of different genotypes, and preferentially selecting high-fitness genotypes for reproduction. One feature of EAs is that they can be *blind*: they do not require any knowledge of how the genotype is interpreted, or how the fitness of a genotype is measured [Goldberg89]. When properly exploited, this blindness is extremely powerful: it allows the EA to be independent of the fitness computation or details of genotype interpretation, so that these can be arbitrarily changed without requiring any modification to the EA. For example, the standard genetic algorithm represents solutions as a fixed-length string of bits, and does not depend on how the bits are interpreted to represent a specific solution when measuring the fitness of the solution. A GA views a solution simply as strings of bits and a fitness measurement; the actual meaning of the bit string and the method of determining the fitness of the bit string are irrelevant to the GA. The interpretation and fitness measurement can thus change as new problems are addressed, but the solution representation and fitness measurements themselves do not appear any different to the GA. On the other hand, if a particular GA depends on knowing how parts of the bit string are interpreted or how they relate to each other, then the GA will have to be changed if the interpretation is modified. Thus, if we wish to maximize the ability of a robot synthesis system to address new problems and synthesize arbitrary properties, then we should choose a genotype representation that minimizes what the EA must know about the genotype's interpretation.

So far, we know that the genotype representation should hide the details of interpretation from the EA, and the EA needs to be able to exchange properties between different genotypes. These are very general requirements; to determine an appropriate genotype representation, we must consider the types of robots and properties that will be synthesized. It is desirable to be able to synthesize both kinematic and non-kinematic properties for both modular and non-modular robots. It is also desirable to address synthesis of manipulators (including multiple or bifurcated manipulators) and mobile robots. Manipulators require some way to represent serial chains; mobile robots may require multiple serial chains (in the case of walkers), wheeled, or even free-flying bodies. Different types of robots require different properties to be synthesized: manipulators require parameters describing kinematics, dynamics, and actuators, while a wheeled rover needs descriptions of suspension, wheel diameter, and perhaps even sensor placement--and the rover may also have a manipulator on it. One possible genotype (as in [McCrea97]) would be to represent each type of robot as a fixed set of parameters (with each new design task requiring a specific parameter set), so that the evolutionary algorithm operates on the variables without knowing what they represent. This encapsulates the interpretation of the genotype in task-specific code, allowing new properties to be represented and synthesized, and new tasks to be addressed. However, this representation has some limitations: each task or robot type requires a new way of representing solutions, and the fixed set of parameters makes it difficult to vary the size (such as number of degrees of freedom) of solutions.

Another representation is that used by modular robot synthesis methods: robots are assembled from a set of fixed modules, and the evolutionary algorithm replaces or reorders modules, and also swaps modules between solutions. This makes it possible for each module to be self-contained so that the evolutionary algorithm does not need to know many of the details of each module, apart from how the module connects to other modules. Additionally, the evolutionary algorithm can exchange information between robots of varying size and topology by swapping modules or sets of modules between robots. However, using a purely modular approach has a significant drawbacks. Properties of the robot cannot be varied except by exchanging modules, which has one of two implications: either the synthesized designs will be poorly optimized due to limited selection of module properties; or a large number of similar modules (such as elbow joint modules with varying dimensions and actuators) are required, and the synthesizer must know how the modules are similar.

2.1.1 Parameterized Modules

By combining purely modular and purely parametric representations, the limitations of both approaches can be eliminated. This *parameterized module* representation is similar to a fixed module representation, but allows each type of module to have an arbitrary number of parameters. A module's parameters represent arbitrary properties of the module, such as geometric dimensions, actuator type, or even controller parameters. Each module is a self-contained software object and can include kinematic, dynamic, structural, and other representations in addition to special-purpose routines for simulation, control, and analysis. The complexity of a module can vary: at one extreme, a module with no parameters might represent a link of a robot with fixed properties (i.e. a fixed module), and at the other extreme a module might be an entire mobile robot with parameters for each property (i.e. a parameter set describing a robot). The synthesis algorithm can vary the parameters of modules, and the way the modules are connected (Figure 2.1); when coupled with appropriate analysis capabilities, this allows synthesis and optimization of properties such as actuators, structural dimensions, kinematics, and dynamics.

As with purely modular synthesis, each module type specifies the location of its *connectors*, which are used to attach modules to each other. These connectors do not need to have a physical embodiment in the robot's hardware; they are simply for specifying how modules can be connected together. Naturally, if Darwin2K is being applied to a

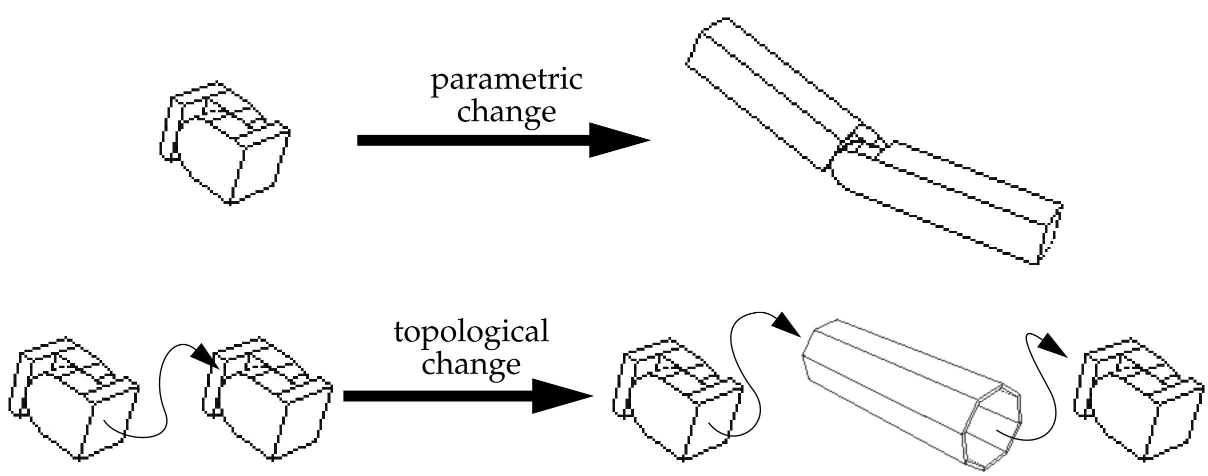


Figure 2.1: Parametric and topological modification
Darwin2K represents robots as connected assemblies of parameterized modules. This allows the synthesis algorithm to vary both robot parameters (top) and topology (bottom).

purely modular design problem, the modules can contain geometric and other representations for the connectors of the actual hardware modules, but this is not required when synthesizing non-modular robots.

Parameterized modules have two important and related advantages over fixed modules. The first is that the synthesizer can independently vary a parameterized module's properties, thus allowing changes to be made with minimal accidental disruption of well-optimized features. In contrast, if fixed modules were used then the synthesizer would have a difficult time independently varying a single parameter: it would have to know how each of the modules differ from every other module, or it would have to blindly select another module which could lose information about properties that are already well-optimized. The second advantage of parameterized modules is that they can efficiently represent a wide range of module properties. For example, a typical joint module in Darwin2K might have parameters which specify the motor, gearhead, overall length, outer diameter, and inner diameter. If each of these variables can have 8 discrete values, there are $8^5 = 32768$ different variations of the joint module. If purely modular design was being used, 32768 different fixed modules would be required to represent the same set of module properties. Clearly, it is impractical for a human to specify the properties of each different fixed module, and automatically specifying the properties for the entire range of modules is effectively the same as using parameterized modules. Storage management and memory issues also become more important if such a large number of modules is required. Thus, the ability of parameterized modules to represent a large space of module properties, coupled with their ability to allow independent variation of properties, enables efficient and independent optimization of robot attributes.

Parameterized modules also have significant advantages over purely parametric representations. First, they do not assume any fixed, global interpretation of the parameters. Two robots might be identical except that one has a joint with a brake on it. Using parameterized modules, the robots would have a different module type for that joint, one with an extra parameter describing the brake. This difference does not have a global impact on the representation of the robot because the brake parameter is encapsulated in the module. A purely parametric representation would require a new procedure for interpreting the set of parameters; the addition of an extra parameter causes a global change in the parameter interpretation.

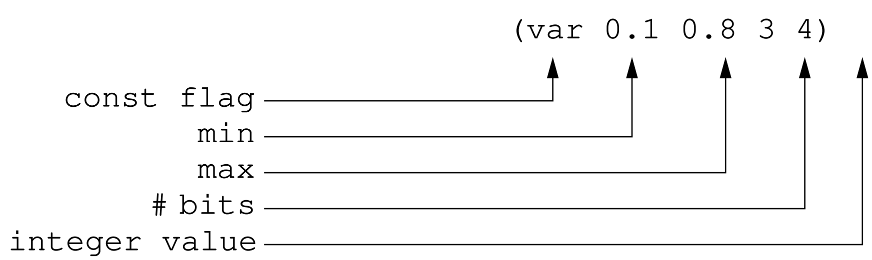
Second, parameterized modules allow information to be exchanged between solutions of varying size. For example, the parameters describing the wrist structure of a 6-DOF manipulator may also be useful for a 7-DOF manipulator. A purely parametric approach does not tag the parameters with any sort of identifier, so there would be no way of ensuring that the wrist parameters for the 6-DOF robot end up in the right place for the 7-DOF robot. Parameterized modules effectively tag each parameter with an identifier that specifies how it is interpreted; thus, when the wrist modules from the 6-DOF robot are exchanged with those of the 7-DOF robot, the parameters retain their meaning because they remain associated with the same modules. Modules also provide the synthesizer with building blocks of an appropriate level of detail: a lower-level representation might require the synthesizer to figure out geometric details that are obvious to a human designer (e.g. a motor and gearbox should be located near each other and near the joint), while a higher-level representation (e.g. selecting from a number of existing manipulators) would not allow much optimization to be performed.

Darwin2K's modules present a simple, consistent interface to the synthesis algorithm: each module has a type identifier, a *component context* label, a list of parameters, and a list of attachments to other modules. The module type identifier allows the synthesizer to exchange information between modules of similar type. The component context label is reference to a list of components (e.g. motors and gearheads) to be used when interpreting the module's parameters; the component context (list) itself, and the properties of the individual components, are specified in a component database file. (See Section 4.7.6 for descriptions of the component models used in Darwin2K.) The attachment list describes how the module is connected to other modules, and the parameter list determines the properties of the module. The synthesizer does not need to know how each parameter is interpreted, thus allowing parameters to represent arbitrary properties. Since modules are self-contained, there is no single global function which interprets parameters; thus, new module types can be added with out requiring modification of existing code.

Each parameter has several attributes: minimum and maximum values, a number of bits, an integer value, the actual parameter value, and a *const flag*. The minimum and maximum values determine the range of actual values; the number of bits determines the resolution (or discretization) of the actual value, and the integer value is the binary representation of the actual value. The actual value a is obtained from the number of bits b, minimum value m, maximum value M, and integer value i by a simple linear interpolation:

$$a = m + \frac{i}{2^b - 1}(M - m) \tag{2.1}$$

Values of i range from 0 to 2^b-1; thus, an integer value of 0 gives an actual value of m, and an integer value of 2^b-1 gives an actual value of M. The *const flag* may take one of two values: `var`, indicating that i (and thus a) may be changed by the synthesizer; or `const`, indicating that i and a are fixed. Thus, the designer can specify values for some parameters if their optimal values are known a priori (or if they should be fixed for any reason). The LISP-like text format used to specify parameters in Darwin2K (and in the text of this thesis) is as follows:



While some parameters represent continuous values such as dimensions, others may specify discrete properties such as motor, material, or gearhead selections. In these cases, only the number of bits, integer value, and `const` flag are used. The component context mentioned earlier specifies which components are determined by each of a module's parameters. This component context consists of a list of permissible components for each selection parameter, and is described in a component database file. In the component database file, the component context for a particular type of joint module might look like this:

```
context revoluteJoint {
        parameter 0 "motors";
        parameter 1 "gearHeads";
}

componentList "motors" {
        rotaryActuator "maxonRE25.118755";
        rotaryActuator "maxon2260.815";
        rotaryActuator "maxonRE36.118800";
        rotaryActuator "maxon2260.889";
}

componentList "gearHeads" {
        gearBox "maxon16.118188";
        gearBox "maxon26.110396";
        gearBox "maxon32.110464";
        gearBox "maxon42.110404";
}
```

In this case, parameter 0 would select a motor (from the `componentList` labeled "motors"), and parameter 1 would select a gearhead (from the list labeled "gearHeads"); other parameters of the module do not select components and so are not mentioned in the component context. In addition to the component contexts, the component database file contains descriptions of each component, such as the density and modulus of elasticity for materials, or the gear ratio, efficiency, and torque and velocity limits for gearheads. Using actual component descriptions from manufacturers' catalogs can help reduce design time, since a human engineer will not have to search through catalogs to find components that meet all design constraints.

Each type of component (`rotaryActuator` and `gearBox` in the example above) is a software class, with code for parsing entries in the component file. In this way, new component types can be added (complete with parsing procedures); since there is no global, monolithic function for parsing the whole component database file, new component types can be added without requiring modification to the existing parsing code.

Some selection parameters may have dependencies on other selection parameters: for example, one parameter can specify a motor, and another can specify a gearhead. However, not all gearheads are compatible with all motors. To handle compatibility constraints such as this, Darwin2K allows components to specify dependencies on other components in the component database file. The order of a module's parameters determines how the component dependencies are resolved: a component specified by the j^{th} parameter can depend only on the components selected by parameters 0 through j-1, thus preventing cyclic dependencies. The procedure for determining a module's components from its parameters and component context is shown in Figure 2.2.

Internally, each module type in Darwin2K is a C++ class, and specific modules are

C++ objects. (If the reader is not familiar with object-oriented programing, see Appendix A for brief description of object-oriented programming and its use in Darwin2K.) All modules in Darwin2K are derived from the `module` class, so they share a common minimum interface. Each module has a (possibly empty) list of parameters that the synthesizer may change. Since modules are self-contained software objects, they also contain several functions (called *methods*) that are used when measuring the performance of the robot. Some modules have specialized methods, but at the bare minimum each module must specify a `createGeometry` method, which creates a polyhedral representation of the module's physical geometry. This polyhedral representation may be as simple or as complex as the designer wishes, and is used to generate the kinematic, geometric, and inertial properties of the module. Each module's geometry consists of one or more `part` ob-

---

```
procedure chooseComponents(parameterList PL,
                           componentContext CC
                           module M) {
  parameter p; /* parameter being used to select component */
  list l;      /* list of components for one parameter */
  component c; /* specific component choice */

  for j = 0 to (|PL|-1) {
    p = PL[j];

    if p is a component selection parameter then {
      /* we need to select a component for p */
      l = list of allowable components for p from CC;

      /* next, filter components from l that are not */
      /* compatible with previous components         */
      for k = 0 to j-1 {
        if PL[k] is a selection parameter then {
          c = component for PL[k];
          remove components from l that are incompatible
            with c;
        }
      }

      /* now, l contains only components that are   */
      /* compatible with previous components.       */
      /* p's component is the ith entry in l, where */
      /* i is p.i, the integer value for p.         */

      /* add the selected component to M's list of components */
      addComponenentToModule(M, l[p.i]);
    }
  }
}
```

Figure 2.2: Pseudocode for selecting components

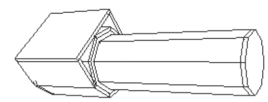
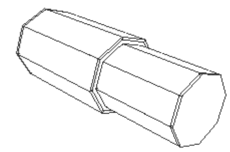
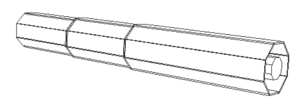
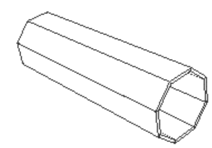
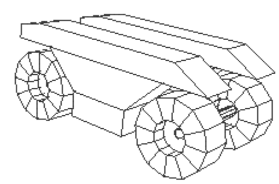
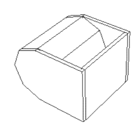
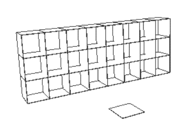
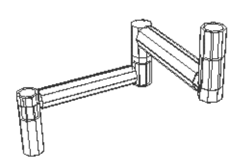
jects, each of which contains one or more polyhedra. A module's `parts` are connected together by fixed, rotating, or translating connections, allowing joints to be created. Darwin2K provides geometric primitives for parts and polyhedra so that the `createGeometry` method for new module types can be easily implemented. `createGeometry` is also tasked with specifying the location of the coordinate frames of the module's connectors, which are used to attach modules together.

Darwin2K has several abstract classes derived from the module class: `dofModules` (those with degrees of freedom), `linkModules` (those without degrees of freedom), and `toolModules` (those representing end effectors). Additionally, `dofModules` are subdivided into `jointModules` and `baseModules`. The synthesizer differentiates between these broad classes of modules, so that nonsensical robots are avoided: for example, replacing a robot's base with a tool would result in an infeasible robot. Each of these subclasses defines some new methods and/or members; for example, `toolModule` defines a tool control point (TCP) data member and a method for converting the TCP to the module's link coordinate system, and a `dofModule` contains methods for querying about the torque, position, and velocity limits of the module's degree(s) of freedom. These methods are used not by the synthesizer, but by the evaluation algorithms described in Chapter 4. New module types can be derived from these generic classes, and since they present a consistent interface to the evaluation algorithms, the algorithms do not require modification to work with the new modules: for example, different types of joint modules may have different numbers of parameters, and may interpret their parameters differently, but they all have the same interface to the synthesizer and simulation algorithms, so the algorithms can be independent of the details of the modules' internals. Darwin2K currently contains approximately 40 module classes for various bases, links, tools, and joints. Some are general-purpose and are useful for many problems, while others are task-specific (that is, they were implemented for a particular design problem). Table 2.1 shows a selection of these modules and their properties. Appendix B also lists a number of parameterized modules and gives a description and parameter list for each.

2.1.2 Connecting modules: the configuration graph

Parameterized modules describe only parts of a robot; an additional representation is needed to complete the robot description. The configuration graph is just that: it describes the way the modules are connected to each other. The configuration graph is a directed acyclic graph (DAG) in which nodes are modules and edges are physical connections between modules. The edges of the graph are *directed*: modules specify connections to their children via outgoing edges, but do not refer to their parent connections (incoming edges). The configuration graph is stored as a list of topologically sorted modules, and cycles in the graph are prevented by only allowing modules to specify attachments to modules which occur later in this list. Each module in a configuration can specify an attachment to another module for each connector. Attachments contain several fields: the ID of the connector on the parent module, the index of the child module, the ID of the con-

Table 2.1: Sample of Darwin2K's modules

module type	**parameters**	
`rightAngleJoint`	components: motor, gearbox, material length between connectors tube diameter wall thickness	
`inlineRevolute2`	components: motor, gearbox, material length between connectors tube diameter wall thickness	
`prismaticTube`	components: motor, gearbox, lead screw, material outer diameter wall thickness segment length	
`hollowTube`	material selection length outer diameter wall thickness	
`oclChassis`	wheelbase, engine location (front to back), front-to-back position of connector (to which other modules can be attached) connector height	
`offsetElbow`	components: motor, gearbox, material distance between actuator housing and plate initial joint angle wall thickness	
`stackerBase`	total number of bins, number of bins vertically, x and y location of connector, x and y location of payload entry point	
`scaraElbow`	components: motor, gearbox for each of 3 joints; material for all links length, wall thickness, diameter for links joint angle offsets for each joint	

nector on the child module, a `const`-flag, handedness, and a twist parameter

```
(var 0 (0 1 left (var 0 270 2 1)))
```

```
const flag
parent connector ID
index of child module
child connector ID
handedness
twist parameter
```

The connector IDs simply identify which connectors are being attached; the twist parameter indicates the angle of rotation about the z-axis of the child's connector with respect to the parent's connector. The handedness can be either `left`, `right`, or `inherit`, and indicates whether the module's geometry should be normal (`left`) or a mirror image (`right`) A handedness of `inherit` means that the module should have the same handedness as its parent module. As with parameters, the `const`-flag can be used to indicate that a property (in this case, the module and connector references) should not be changed by the synthesizer. This allows the designer to specify that a particular sequence of modules should be untouched by genetic manipulations. For example, if the designer knew that a particular wrist configuration and end effector are required, then she would set the `const`-flag for the attachment between the wrist module and end effector. Thus, the designer can easily add significant constraints on the final form of the synthesis results, effectively incorporating task-specific knowledge and human expertise into the synthesis process. Figure 2.3 shows the text representation of a simple robot, and its physical instantiation.

For the purposes of simulation and analysis, the configuration graph is instantiated into a mechanism consisting of links (rigid bodies) connected by joints. This process is performed recursively, starting with the base and proceeding in a depth-first manner. The `createGeometry` method is called for each module, and the `parts` on either side of inter-module connections are then attached to each other by a rigid connection. (All connections between modules are rigid; the only non-rigid connections are those forming joints between parts within the same module.) After all modules have created their geometric representation, the parts that are rigidly connected (as opposed to those connected by translating or rotation connections) are grouped into rigid bodies, and the inertial properties of each rigid body are computed using Coriolis [Baraff96]. The mechanism is thus represented by a tree, with nodes representing rigid bodies (links) and edges representing joints between bodies. The root of the tree is the one of the base's links. Figure 2.4 shows the configuration graph and mechanism (links and joints) representations for a simple, non-branching manipulator. After creating the mechanism representing the robot, the mechanism tree is traversed to identify serial chains. The mechanism and serial chain representations are used by numerous algorithms during the evaluation process, such as computing the robot's Jacobian or dynamic model.

While the mechanism graph is a tree, the configuration graph may not be: when the graph is interpreted as mechanism, multiple connections referring to the same module will result in multiple copies of that module and its children. This allows pieces of a mechanism to be duplicated to preserve symmetry. For example, a hexapod robot would

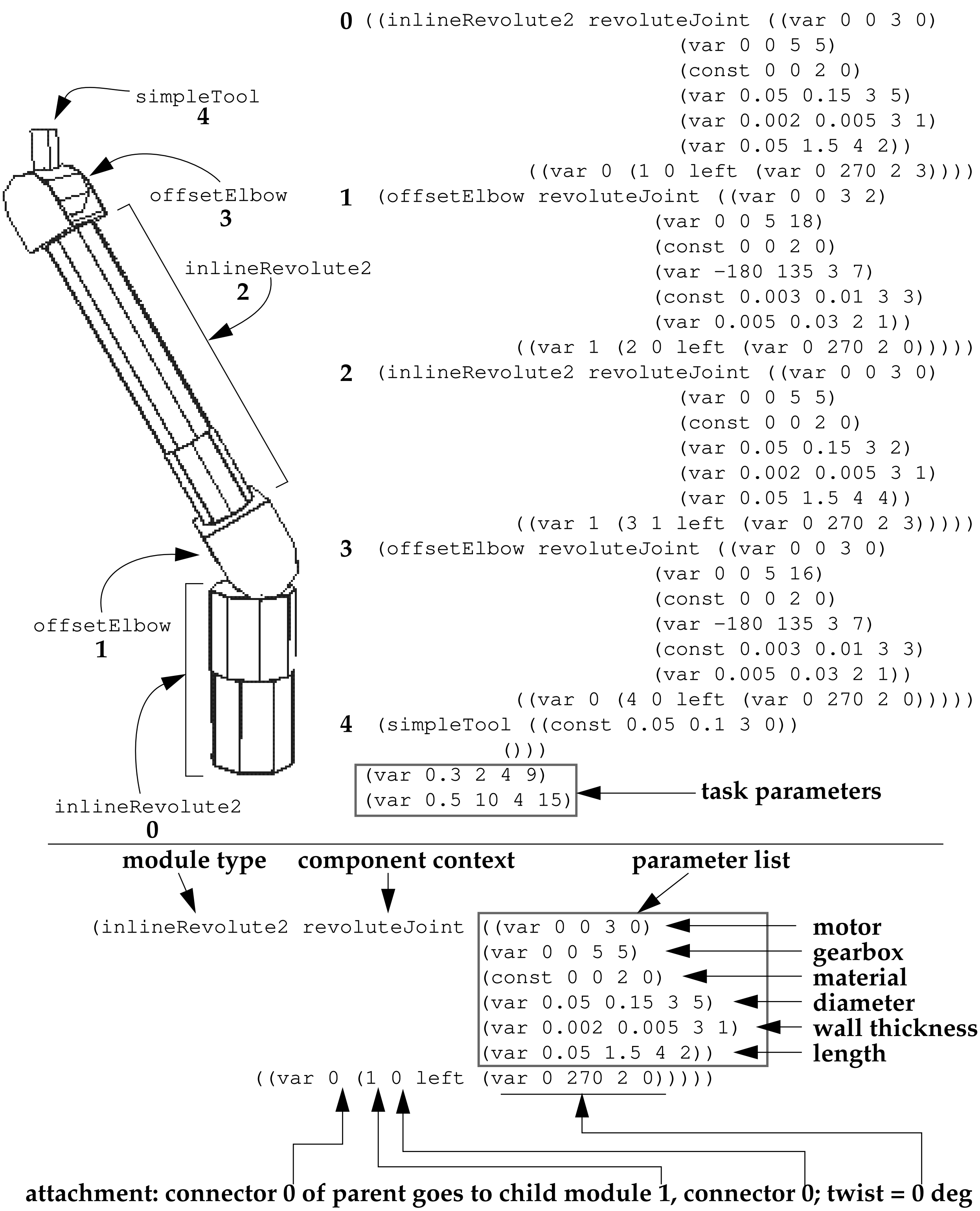


Figure 2.3: Sample configuration and text description

A 4-DOF manipulator constructed from five parameterized modules and with two task parameters is shown at top left, along with its text description (top right). Each module is labeled with a bold number to show the correspondence between the robot and text. At bottom is module **0**, with labels detailing parts of the module description.

have one base with 6 connectors, each of which is attached to the same module (the base of a leg) in the configuration graph. Figure 2.5 shows an example of a free-flying robot with two identical arms. Note that there are two connections from the second module to the third; thus, the third module (and all its child modules) are duplicated, creating two identical arms.

Since it may be desirable to optimize some properties that describe the task (such as via point location or path-following velocity for a trajectory), configurations may optionally contain one or more *task parameters*. This is simply a list of parameters that are interpreted by the evaluation and simulation algorithms. Whether task parameters are included in a configuration depends on the task being addressed, and the number of task parameters must be the same for all configurations in a given synthesis run. The synthesizer only needs to know how many task parameters are being used, not what they mean or which parts of the task they correspond to; each software object used in the simulator parses its corresponding task parameters, thus isolating the interpretation of the task parameters from the synthesis algorithm.

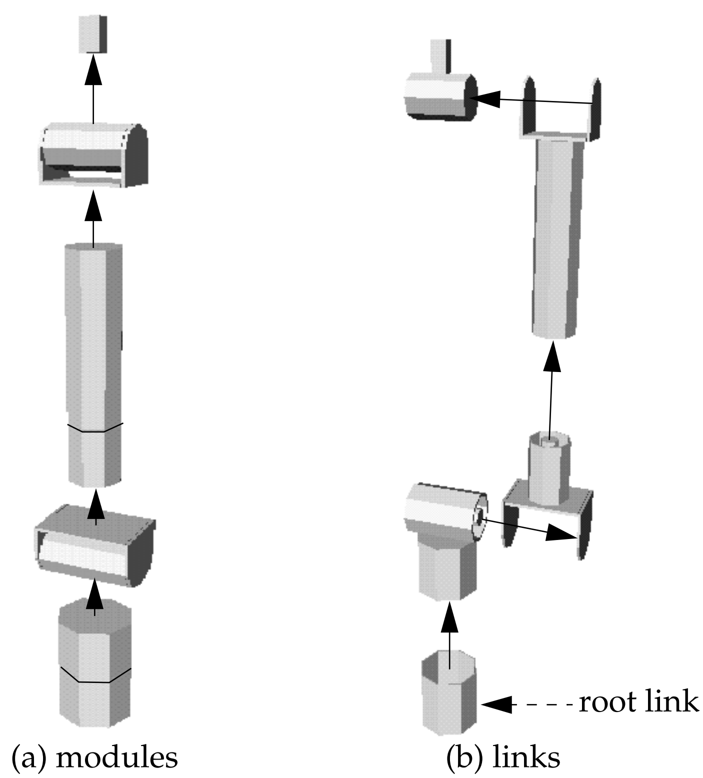


Figure 2.4: Configuration shown as modules and links

(a) shows the configuration from Figure 2.3 as a configuration graph, with modules connected by attachments (arrows).

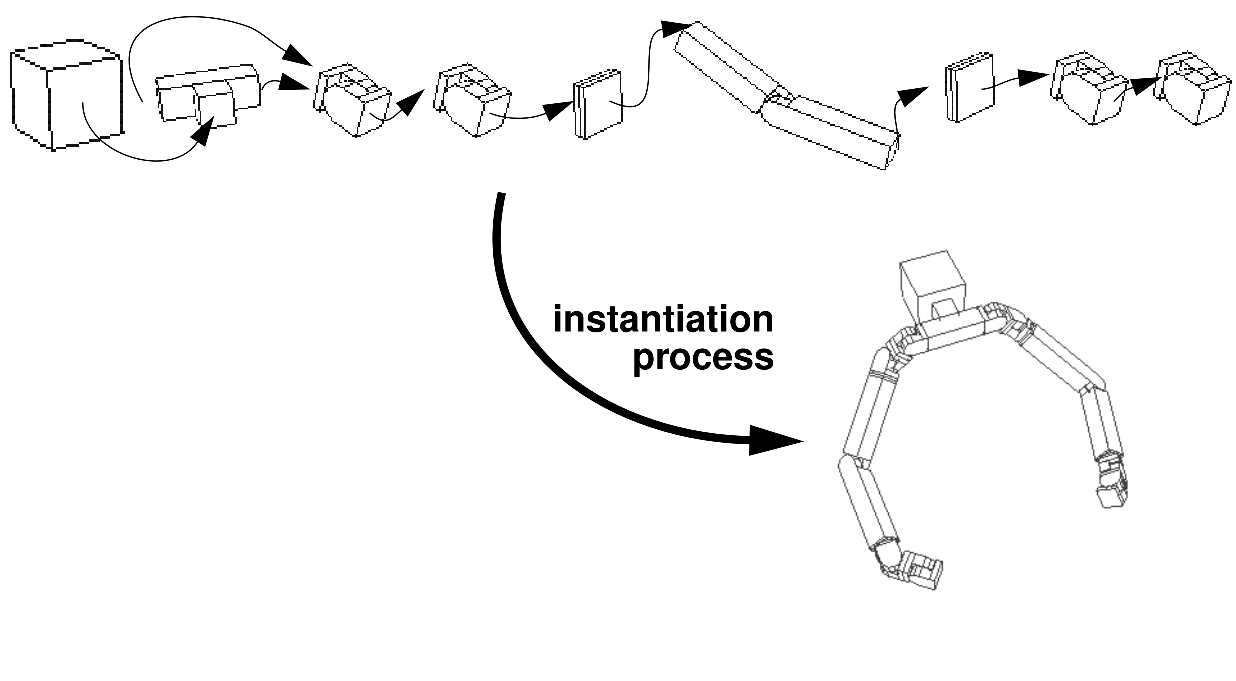
(b) shows the robot represented as a mechanism, with links connected by joints. In this figure, the joint axes are represented by arrows, and the links have been displaced along the joint axes for clarity. The root link of the mechanism is indicated by the dashed arrow.

The Parameterized Module Configuration Graph is significantly more flexible than representations used in previous robot synthesis work. This representation is able to express purely modular, purely parametric, and hybrid robots, and thus is useful for modular and non-modular design. In prior systems, representations for non-modular robots were purely parametric and did not include non-kinematic features of the robot such as actuators, inertial properties, and link geometry. The only systems to include representations of these properties used purely modular representations, so non-kinematic properties could not be independently varied. The PMCG also allows synthesis of mobile robots, multiple and branching manipulators, and allows the inclusion of task-specific knowledge through the use of `const`-flags for both parameters and robot topology.

2.1.3 Limitations of the Parameterized Module Configuration Graph

The primary limitation of the PMCG is the restriction on the form of parallel mechanisms: multiple connections to a single module in the PMCG result in duplication of the module, rather than multiple connections to the same physical module (Figure 2.6a). Because of this, multiple modules cannot lie in parallel in the configuration graph. However, parallel mechanisms can be created within a single module (Figure 2.6b) by specifying all but one of the connections between the `parts` forming a loop. The module can then use its own internal procedures for ensuring correct positioning of the links and joints in the loop. It should be noted that Darwin2K's does not include any algorithms for parallel

Figure 2.5: Configuration graph for a two-armed robot
At top is a symbolic view of a configuration graph. Each node is a module, and each edge is a connection between modules. The second module from the left has two outgoing connections to the third module; when the graph is parsed to create a description of the robot's links and joints, the subgraph rooted at the third module is duplicated, thus preserving symmetry.

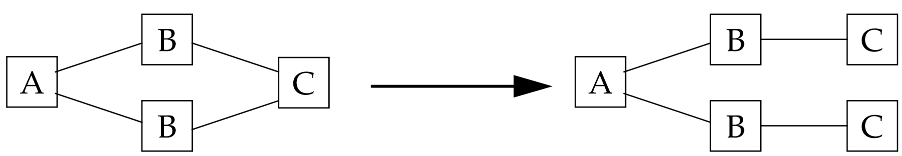


kinematics, dynamics, or other types of analysis, though these can be added through the extensible architecture discussed in Section 2.2.

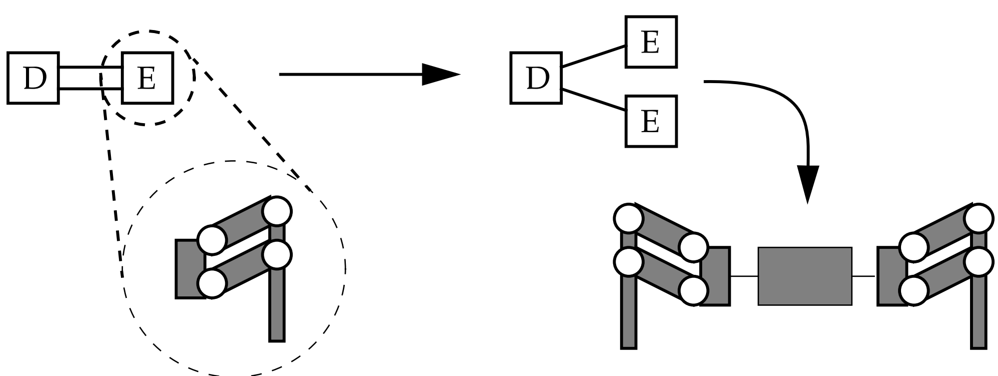
Another limitation of the PMCG stems from the fact that each module has a fixed number of parameters. This makes it difficult to represent variable mechanism size and topology within a module while retaining independent parameters for each of the module's constituent parts. For example, a module could represent a serpentine manipulator with a parameter for the number of links in the arm; however, since the number of parameters is fixed, the properties of each link in the serpentine manipulator could not be varied independently by the synthesizer. Perhaps the most significant impact of this restriction is in synthesis of mobile robots, where the ability to synthesize complex linkages such as those used in some suspensions is limited to parametric variations on fixed topologies. Still, the ability to choose between different topologies (e.g. multiple mobile base types) and optimize their parameters is an important capability.

2.2 Extensible Software Architecture

Each application has its own task description and requirements, which drive the properties of the robot being designed -- whether it is being designed manually, or by an



(a) loops in configuration graph get split before creating mechanism



(b) parallel linkages can be created within a single module

Figure 2.6: Parallel configuration graphs and mechanisms in Darwin2K

Loops in the configuration graph (multiple references to a single module) result in duplication of those modules that are referenced more than once, e.g. the module labeled "C" in (a). However, parallel mechanisms can exist within a single module in Darwin2K, as in the circled module in (b).

automated synthesis method. No single task representation short of a fully-featured language can represent the range of tasks for all robots: a planetary rover may have to navigate over rough terrain and around obstacles, while a manipulator may have to spray paint an object of known shape and position. Similarly, the requirements of each application are different: a planetary rover should have minimum mass and power requirements while maximizing the ability to cross rugged terrain, but the spray painting robot might be concerned with painting the entire object as fast as possible, minimizing speed changes and not colliding with objects in the workcell. A flexible robot synthesis system should not be dependent on a specific task representation, as this will limit the system's applicability.

The Parameterized Module Configuration Graph allows a wide variety of robot types, topologies, and properties to be represented; an equally flexible means of describing applications is also needed. To achieve this, Darwin2K employs an extensible, object-oriented software architecture that allows new task representations, simulation and analysis algorithms, and performance metrics to be added. The reason for this is simple: there is no one task representation that is suitable for all robots. Several robot programming and scripting languages have been developed for manipulators, precisely because different tasks require different plans, control schemes, and data structures. Most robots, particularly field or mobile robots, are controlled by programs written in a high-level language such as C. For this reason, it makes sense for an extensible synthesis system to allow special-purpose code (written in a high-level language) to be used for task representation, control, and simulation. With that said, there are certain primitives (such as trajectories in Cartesian space, PID controllers, and dynamic simulation algorithms) that are broadly applicable in robotics. Darwin2K includes a set of these primitives (described in Chapter 4) and allows new tasks, control algorithms, and simulation methods to be added (written in C++) without requiring modification of the existing code or architecture.

Another factor that affects the applicability of a robot synthesis system is the degree of coupling between the synthesis algorithm and the task. In some systems, there are explicit dependencies between the task and synthesizer. For example, in [Kim92], there is a separate population of robots for each via point in a trajectory; if the number of via points in the trajectory is changed, then the synthesizer's properties (number of sub-populations) must change as well, and it seems likely that the synthesizer's performance would be affected by the number of via points. Ideally, the synthesis algorithm used by a synthesis tool should not depend on the task being addressed, so that the task can be changed without affecting the synthesis algorithm. Thus, if wide applicability is a goal, then the system should allow new task representations to be added and the synthesizer should be completely independent of task.

Darwin2K's software architecture has been designed so that the synthesizer is insulated from the details of task and robot primitives. This is enforced to a degree such that new task-specific capabilities are added to Darwin2K through dynamic libraries, so that the synthesis and the evaluation programs do not even need to be recompiled. (Any parameterized modules or simulation algorithms that are specific to the task and hand can be compiled into libraries, which are then linked at runtime by Darwin2K.) The synthesizer operates on configuration graphs, and requires fitness measurements for each configuration it generates. However, the synthesizer does not depend on the internals of the modules, task requirements, or evaluation process; thus, it makes sense to define an interface between the synthesis and evaluation procedures so that all task-specific information (including the internals of modules) are contained entirely in the evaluation library, which is not used by the synthesizer. This separation allows application-specific task representations and simulations to be used when evaluating the performance of configurations. Additionally, since the synthesizer is an evolutionary algorithm, it can be easily parallelized. Darwin2K takes advantage of this by having one process containing the synthesizer, and many identical processes for evaluation. The synthesis process (called the Evolutionary Synthesis Engine, or ESE) sends configuration graphs to the evaluation processes (called Evaluators), and Evaluators send performance measurements back to the ESE. A shared-memory and TCP/IP-based communications package called Real Time Communications (RTC) is used for communication [Pedersen98], allowing processes to be distributed over a network of heterogeneous workstations. RTC provides efficient and robust communication between processes; the entire system can tolerate the failure of individual modules.

Using many identical evaluation processes lends easy scalability to the system: new processes can be dynamically added as more computers are available, and a process can sleep indefinitely if another user requires use of the computer. Each evaluation process periodically checks the machine it is running on to see if the CPU load is high or if any users are active; if so, the process will sleep for several minutes to avoid interfering with others' work. Another benefit of having identical distributed processes is that the failure of any evaluator has no impact other than to make the system run a bit slower. In contrast, other distributed approaches have used heterogeneous distributed processes that had limitations to scalability and robustness to failure. The system in [Kim93] used one process per trajectory point, thus tying the number of processes to the number of points in the trajectory, and in [Paredis96] each evaluation process (rather than the central population database) applied a specific genetic operator, so adding or removing processes altered the relative frequency with which different genetic operators were applied. In both of these systems the synthesis algorithms were affected by the addition or subtraction of distributed processes, thus making them more sensitive to machine availability and program crashes.

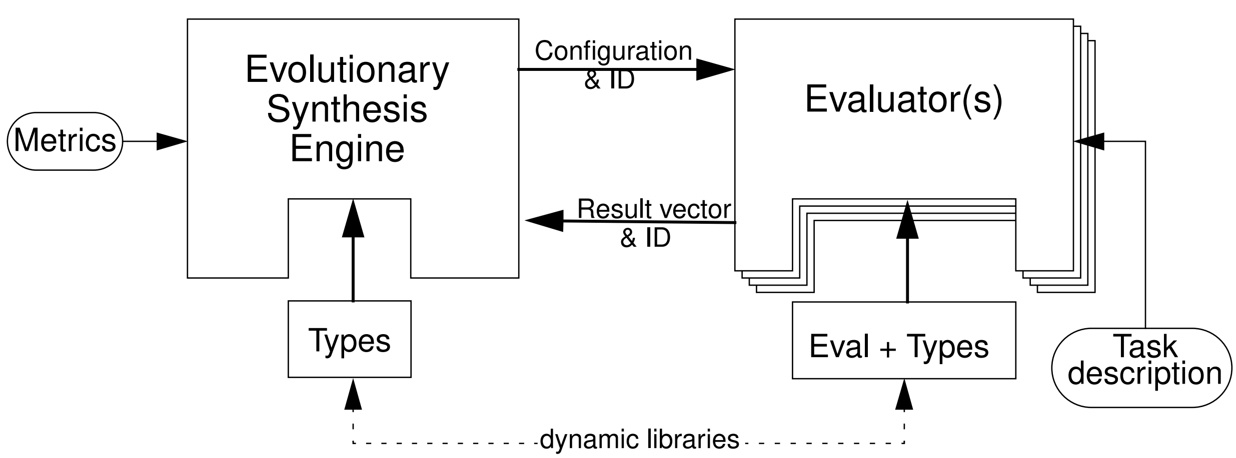
Figure 2.7 shows a schematic view of the synthesis architecture. Both the ESE and Evaluator programs use dynamic libraries for application-specific code. The ESE's library only needs to contain limited descriptions of modules and performance metrics: for modules, it contains the module name, type, and number of connectors and parameters; and for metrics, the metric name and the range of values that the metric produces. (Metrics are described in detail in Chapter 4.) This allows the synthesizer to manipulate configuration graphs, and use metrics for selecting robots to be reproduced or modified as the design space is explored.

The Evaluator's library contains much more information, such as module-specific analysis and simulation code, task-specific simulation algorithms, and code that enables metrics to measure performance. Many of these simulation capabilities are implemented as self-contained `evComponents`; examples include collision detection, PID and Jacobian-based controllers, a motion planner, and trajectory representations. The `evComponents` are used by an `evaluator`, which is a task-specific C++ object for performing simulation initialization and high-level control. `evComponents` provide a standard interface to the `evaluator` class, so that if (for example) a new metric requires a new analysis capability, a new `evComponent` can be created which performs the analysis while working with existing `evaluators`. `evComponents` also provide their own functions for parsing parameter files and for obtaining task parameters from a configuration. Darwin2K includes a set of `evComponents` (described in Chapter 4) that are widely applicable, so that task-specific simulators for new synthesis problems can be quickly constructed.

2.3 Summary

Previous approaches to automated robot synthesis have not focused on the creation of general and extensible systems, and have been limited by robot and task representations. Darwin2K's Parameterized Module Configuration Graph is significantly more flexible than the representations used in previous work, allowing representation (and synthesis, with appropriate analysis tools) of fixed-base and mobile robots, multiple and branching manipulators, modular and non-modular robots, as well as arbitrary parametric properties. To take advantage of the flexibility of the PMCG representation, Darwin2K's software architecture allows new task representations, simulation methods, performance metrics, and other software components to be easily integrated. Additionally, Darwin2K's synthesis algorithm is independent of the internals of robots and tasks, relying only on small, well-defined interface to metrics and modules. This combination of flexible robot representation, extensible architecture, and a task- and robot-independent synthesizer enable Darwin2K to address a wide range of robot synthesis problems.

Figure 2.7: Schematic view of synthesis architecture



3 Synthesis Methodology

3.1 Introduction

What is desirable in a general-purpose synthesis methodology for robots? To answer this question, we must consider the input to, and output from, the configuration process. Design problems usually have multiple, often-conflicting performance requirements; the synthesis method should be able to capture these requirements and generate designs which meet them. It is also desirable to be able to incorporate human design expertise, since the designer may know properties that are useful or required of the solution, thus reducing the computational cost of synthesis and increasing the quality of the generated design. A general-purpose synthesis method for robots should not rely on details of the artifact to be generated, or of the requirements, as this limits the applicability of the method. Substantial synthesis of any type of artifact is often computationally expensive; anything that reduces the computational cost (either in CPU time or the amount of time a person must wait for the result) is beneficial. In particular, it is beneficial to be able to take advantage of multiple computers and to change the scope of the synthesis process based on available computing power.

These requirements have driven the development of Darwin2K's synthesis process. Like previous work in automated synthesis of robots, Darwin2K uses an evolutionary algorithm to synthesize robot configurations. Evolutionary algorithms are appropriate for configuration synthesis for several reasons:

- they are more flexible than analytical approaches since they do not have to make assumptions about the solution form, or the relationship between solution form and performance;
- they can effectively deal with discrete, continuous, and mixed search spaces and solutions of varying size;
- they offer reasonable robustness to local minima in the search space; and
- they are easily parallelizable, reducing system runtime

The first point above has significant impact on the applicability of Darwin2K: it allows the synthesizer to determine arbitrary robot properties and to use arbitrary means of calculating robot performance. In contrast, analytic approaches to synthesis often have significant restrictions on the number and type of properties being synthesized, and can be limited to optimizing metrics whose values can be computed by formulae, rather than through simulation. The second and third points are relevant to the quality of the synthesis result: the search space in design problems is often high-dimensional, non-linear, and can have a mixture of discrete and continuous properties, so the ability to operate over the entire design space (not just a small region of it) is crucial in determining the synthesizer's applicability. Additionally, the design space invariably has many local optima;

while it may not be reasonable to expect an algorithm to consistently reach the global optimum, it should ideally be able to reach well-optimized or near-optimal solutions. Finally, the parallelizability of evolutionary algorithms is of great importance: the quality of the synthesis result generally improves with the amount of computation, and some problems cannot be addressed in a reasonable amount of time on a single computer. For these reasons, evolutionary algorithms are a good match for automated synthesis.

A synthesis method needs to be able to create designs which satisfy multiple performance requirements. Typically, some of these requirements are conflicting (e.g. speed versus power, or capability versus mass and cost), and there are often non-linear dependencies between requirements that affect the relative merits of solutions. As a trivial example, a robot that remains motionless will use much less power than a robot that can complete the task at hand, yet the motionless robot is, to a human, clearly the inferior of the two. Ideally, the synthesis method should consider the nonlinear and discontinuous nature of the interactions between requirements during the synthesis process.

3.1.1 System Overview

Darwin2K consists of an Evolutionary Synthesis Engine (ESE, also referred to as the *synthesizer*) which creates configurations, and one or more evaluation processes which provide fitness measurements to the synthesizer. The synthesizer uses *genetic operators* to create new solutions from existing parent solutions, with parent solutions selected for reproduction based on fitness. Because fitness evaluations can be relatively expensive (typically ranging from 1 to 60 seconds of CPU time) yet are independent from each other, they are best performed in a distributed manner--hence the multiple evaluation processes. In the usual formulation of a genetic algorithm (GA), the GA creates an entire population of solutions at once, and then measures the fitness of each solution; this is called a *generational* approach. When fitness computations must be distributed to reduce the algorithm's runtime, the generational approach can be wasteful: the faster evaluation processes will idle after finishing their assigned computations as the GA waits for the fitness results from the slowest evaluation processes. The *steady-state genetic algorithm* (SSGA) [Whitley90] remedies this by continually generating new solutions and adding them to the population, while removing less-fit solutions to make room. Darwin2K uses the steady-state approach to make efficient use of distributed computing resources.

To select configurations for reproduction in a way that can optimize multiple objective functions, the synthesizer can use one of two methods: Requirement Prioritization or a Configuration Decision Function. Requirement Prioritization captures the relative importance of task requirements in an intuitive way and efficiently guides the synthesizer as it explores the design space. The CDF encodes the designer's decision process for determining which of two configurations is better, and is used by the synthesizer to choose configurations for reproduction and deletion. To prevent well-optimized designs from being lost due to the probabilistic nature of evolutionary algorithms, the synthesizer uses an *elitist* method that is appropriate for multi-objective optimization. This method combines Pareto-optimality (the notion that a solution is not inferior to any other solution) and solution feasibility (relative to a set of task requirements) when deciding whether a solution can be removed from the population to make room for new solutions.

Finally, because of the way Darwin2K uses the Parameterized Module Configuration Graph (PMCG) representation (described in Chapter 2), existing knowledge of useful robot properties (from both human designers and from previous synthesis runs) can be directly incorporated. This has several beneficial effects: synthesis time can reduced (or synthesis quality improved) by including partial solutions known to be useful; the scope of synthesis can be increased or reduced depending on the computing available; and the synthesis tool can be used to iteratively refine solutions.

3.2 Genetic Operators

One defining feature of evolutionary algorithms is that they create new solutions from existing, tested ones. This creation is done by *genetic operators*, which are procedures that modify or combine solutions to create new solutions. Three broad classes of genetic operators are *crossover* operators, mutation operators, and duplication. Crossover operators typically create two copies of parent solutions and then exchange parts between the copies; they are the primary source of progress in many evolutionary algorithms. Duplication simply creates a copy of a solution, though it can be counterproductive in steady-state GAs and is not used by itself Darwin2K (see Section 3.4.3). Mutation operators randomly alter properties of solutions, and are useful for introducing new genetic material into the population and for recovering material that has been lost due to limited population size. Mutation is most effective when the population has reached a local minima. In general, mutation operators are used infrequently and are applied to new solutions generated by crossover or duplication.

3.2.1 Crossover Operators

In a Simple Genetic Algorithm [Goldberg89], solutions are represented as a string of symbols. Most commonly, the symbols are bits, and the crossover operator works by exchanging bits between solutions (Figure 3.1). 1-point crossover consists of exchanging all of the bits before or after a randomly-chosen *crossover point* at the same location in both strings; for example, an offspring of two parents of length n would have its first m bits from one parent, and the remaining n-m from the other. 2-point crossover entails choosing two crossover points, and exchanging information in the segment between, or outside of, the points. Both 1- and 2-point crossover are limited in the possible ways information can be exchanged: for two parent strings of length n (generating two offspring of length n), there are $2n$ ways a 1-point crossover operator can exchange information between strings, and $n^2 - n$ ways for a 2-point crossover operator to exchange information. A more capable crossover operator is the uniform crossover operator, which can exchange information independently at each bit, thus allowing 2^n different outcomes. Experimental results have suggested that uniform crossover is more effective than 1- and 2-point crossover [Syswerda89].

In Genetic Programming, solutions are represented by tree structures rather than bit-strings, thus requiring different crossover operators. However, the basic concept is the same as the 1-point crossover operator for strings: select a crossover point (in each of two parent trees), and swap the subtrees beneath the crossover point. Since Darwin2K contains both graph and bitstring components (the modules are connected in a graph, and each module has parameters that are represented as bitstrings), the crossover operators are slightly different. Additionally, Darwin2K's crossover operators have to work with constant parameters and constant attachments that the designer has indicated should not be changed. At one extreme, every attachment between modules might be designated as `const`; at the other, a purely modular design problem might have modules with no parameters, or with every parameter's `const` flag set. To handle these cases (and those in between), Darwin2K has both a *parameter crossover operator* and a *module crossover operator*. Briefly, the parameter crossover operator looks for modules of the same type and performs a uniform crossover on the parameters that are not marked `const` in either module, while the module crossover operator searches for non-`const` attachments and exchanges the subgraphs after the attachment points.

The parameter crossover operator searches monotonically through two configurations' module lists. Beginning with the first module in each configuration, the operator checks to see if the modules' types and component contexts are the same. If so, it performs uniform crossover on the modules' non-const parameters (Figure 3.2). If corresponding parameters have different ranges and resolutions in the different modules, the minimum and maximum values for the parameters are adjusted to cover the union of the two parameters' original ranges, and the numbers of bits for the parameters are adjusted so that the resolution is equal to or greater than the parameter with higher resolution. Each parameter's floating-point value is then converted into an integer value based on the new range and resolution, and finally uniform crossover is performed. In practice, adjustment

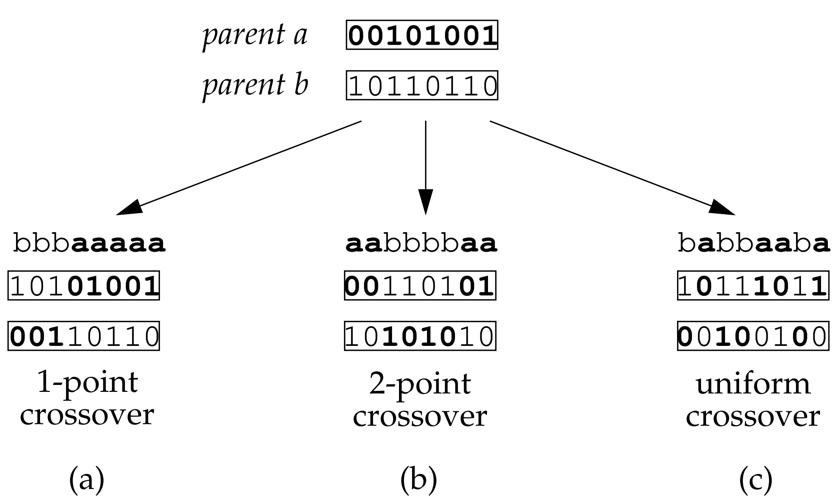


Figure 3.1: Crossover operators for bit strings
Three common methods of crossover for bit strings are shown here: (a) 1-point crossover, (b) 2-point crossover, and (c) uniform crossover. Two parent strings are shown at top, and a template is given for each crossover operator (string of **a**'s and b's) showing which parent each bit comes from in the offspring.

of the range and resolution of parameters can be avoided by specifying the same range and resolution for corresponding parameters if multiple instances of a module are present in the list of modules used for a synthesis problem. After performing the crossover for the parameters of the two compatible modules, uniform crossover is performed on the non-`const` twist parameters of attachments that connect to corresponding connectors on both modules (e.g. an attachment to connector 1 of a `rightAngleJoint` module will only be crossed with an attachment to connector 1 of another `rightAngleJoint` module). The crossover operation for the attachments' twist parameters is identical to the crossover used for the modules' parameters.

Once parameter and attachment crossover have been performed for one pair of modules, the parameter crossover operator steps to the next modules in the configurations' module lists. If the modules are not compatible, the operator continues to step through one of the configuration's modules (starting from the last module that was modified) until a compatible module is found. Uniform crossover is performed on the modules' parameters and attachments, and then the operator moves to the next module in both configurations. This process is repeated until the end of one configuration's module lists is reached. Finally, if there are any task parameters associated with the configurations (see Section 2.1.2), uniform crossover is performed on them. The parameter crossover operator has the property that if two configurations are topologically identical (i.e. have the same sequence modules connected in the same way), then the resulting crossover is equivalent to a uniform crossover on the bit string formed by concatenating the configurations' variable parameters and attachments. When the topologies are not identical, the parameter crossover operate still does a thorough job of exchanging relevant genetic material. Figure 3.3a illustrates how parameter and attachment information is exchanged between configurations for several cases.

Module crossover (Figure 3.3b and Figure 3.4) is more complex than parameter crossover. It begins by finding a `var` attachment in each configuration cfg_a and cfg_b; these are the two crossover points cp_a and cp_b which point to subgraphs g_a and g_b, respectively. g_a is copied into cfg_b, and any `var` attachments in cfg_b that point to g_b are changed to point

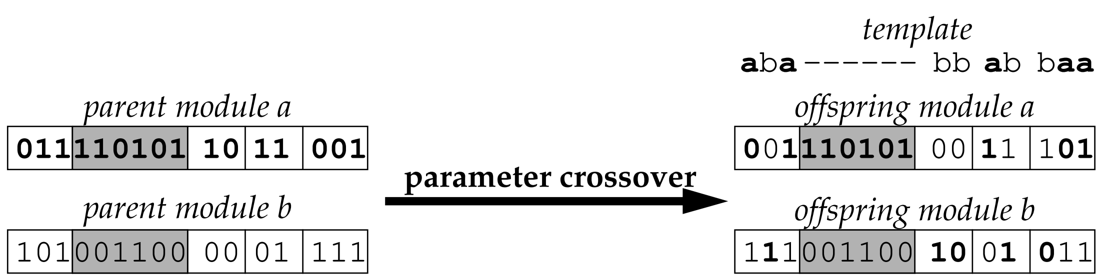


Figure 3.2: Parameter crossover within a module

For each pair of compatible modules in two parent configurations (parent modules *a* and *b* above), the parameter crossover operator performs uniform crossover on the non-const parameters (bit strings in white boxes) to create new modules for the offspring configurations. Boldface digits in the offspring come from *a*, while plain digits come from *b*. Note that the `const` parameters (the shaded boxes) do not change in the offspring.

to g_a (`const` attachments to g_b remain unchanged). Finally, any subgraphs within g_b that do not have any incoming attachments are removed (such as module F in configuration cfg_b in Figure 3.4). This process is repeated with cfg_a and g_b. As with the parameter crossover operator, if the configurations have any task parameters then uniform crossover is performed on them.

The module crossover operator can create configurations whose topologies differ substantially from their parents. This ability allows useful configuration subgraphs to be exchanged between configurations of different size and topology: for example, a useful wrist assembly in a 6-DOF arm can be grafted onto part of a 7-DOF arm. While this can be beneficial in exploring new parts of the design space, it can also be very disruptive to well-optimized features: for example, two robots with 6 1-DOF joint modules each may generate one robot with 0 degrees of freedom and another with 12, and neither may perform as well as the originals. In the later stages of synthesis, where topology is likely to

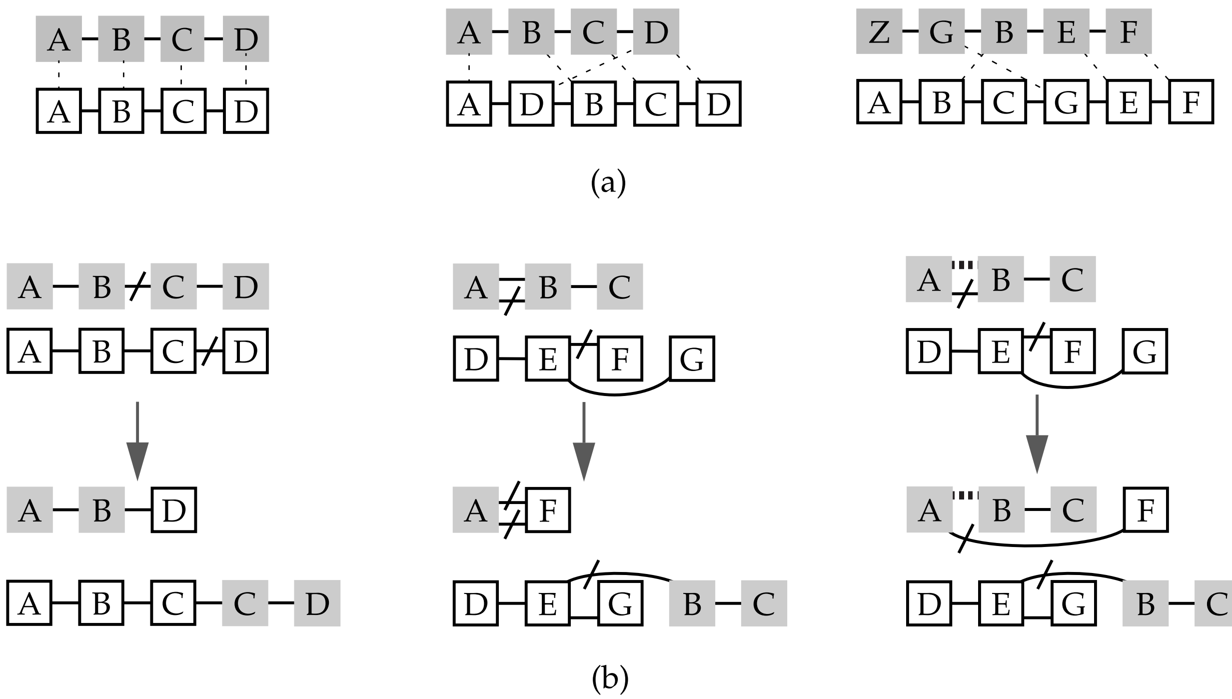


Figure 3.3: Parameter and Module Crossover Operators

(a) The *parameter crossover operator* performs uniform crossover on the parameters of the modules connected by dashed lines. The figure shows three different examples; in each case, parameter crossover is performed on corresponding modules of the same type in both configurations (e.g. the modules connected by dashed lines).

(b) The *module crossover operator* exchanges subgraphs occurring after the crossover points (bold dashed lines). In the middle example, all attachments to module B get modified to point to F in the new configuration; in the example on the right, the upper (grey) configuration has one `const` attachment to B (denoted by the striped line), which is preserved in the new configuration.

be well-optimized, module crossover will rarely be productive. In these cases, it may be more desirable to use a crossover operator which preserves common topology between the parent configurations: after all, if two solutions are selected because they both perform well, then it stands to reason that the features they have in common (rather than the ones they do not) are the likely cause for the solutions' good performances. Thus, preserving common features of two fit parents in the offspring they generate can lead to improved performance [Chen99]. In a normal genetic algorithm operating on fixed-length bitstrings, the crossover operator (be it 1-point, 2-point, or uniform) will preserve any features that two parents have in common: since bits in corresponding locations in the two configurations are swapped, any bits that are the same in both configurations do not change.

This is not the case for configuration graphs: explicit action is required to preserve common features during crossover. The *commonality-preserving crossover operator* (CPCO) preserves the largest common subgraph in two PMCGs and ensures that it is present in the offspring. Commonality-preserving crossover uses the module crossover operator to do the actual crossover; the only differences are in how the crossover points are chosen, and that a parameter crossover is performed within the common subgraphs shared by the configurations. The first step is to identify the largest common subgraph shared by the two configurations. This process relies on a procedure called `growSubgraph`, which takes two modules as input and finds the largest subgraph that begins at the given modules and is topologically identical in both configurations. By *topologically identical*, we mean that corresponding modules in the two subgraphs have identical types and component contexts (if any), and that corresponding attachments between modules within one subgraph are identical (same connectors, modules, and twist angles) in the other subgraph. `growSubgraph` starts with two modules of identical type and examines the modules' attachments; if any attachments are identical, then `growSubgraph` is called recursively on the modules to which the attachments lead. `growSubgraph` computes the

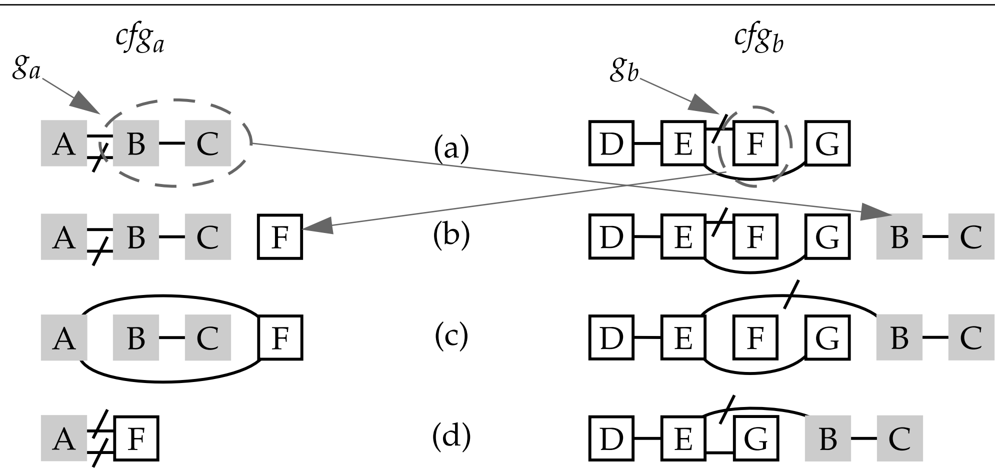


Figure 3.4: Steps of module crossover

(a) original configurations (b) subgraph after crossover point is copied into the other configuration. (c) *all* `var` attachments to the old subgraph (not just the attachment at the crossover point) are switched to the newly-copied one (d) unconnected subgraphs are deleted.

size of the common subgraphs (number of modules) and a mapping between corresponding modules in the subgraphs. (The mapping is simply an array that contains the indices of corresponding modules in the two subgraphs.) The CPCO calls `growSubgraph` for every pair of corresponding modules with the same type and component context in the two configurations, and records the largest common subgraph and its associated starting points and mapping.

At this point, the CPCO has identified the largest common subgraph; this will be preserved during the crossover operation. The next step is to perform a parameter crossover between corresponding modules in the subgraphs. The mapping computed by `growSubgraph` is used to select the modules for parameter crossover. After the parameter crossover, the CPCO chooses module crossover points in the configurations that lie *outside* the common subgraph, thus ensuring that the crossover will not disrupt the topology shared by the two configurations. Module crossover is then performed using the same method as the module crossover operator.

3.2.2 Mutation Operators

While crossover operators can generate a vast range of solutions, they are unable to introduce attributes that do not already exist in the population. For example, if all of the `elbowJoint` modules in the configurations of a population have a '0' in a particular bit of a particular parameter, then no crossover or duplication operator will be able to generate a configuration with a '1' in that location. Similarly, if no configurations in the population contain an `elbowJoint` module at all, then neither will the result of any crossover or duplication. This becomes significant as the population converges towards a local minima. At some point, the only way out of a local minima will be through a mutation, since a finite population size and decreasing diversity imply that some attributes will be lost. Mutation operators can introduce new attributes that have either been lost or that were never in the population to start with.

Darwin2K contains several different mutation operators, corresponding to different features in the configuration graph. The *parameter mutation* operator makes random changes to a module's parameters, while *attachment mutation* makes random changes to the twist parameter of an attachment. These operators are very straightforward: the parameter mutation operator randomly selects a `var` parameter and performs a *creep* mutation, which increases or decreases the parameter's integer value by a small, random amount (in this case, up to +/- 2). Attachment mutation randomly selects an attachment with a `var` twist parameter and randomly selects a new value for the parameter. While a creep mutation could be used instead of selecting a random value, the twist parameter for attachments is often coarsely-discretized (90 degree increments), with several basically equivalent orientations. Because of these factors, a creep mutation on twist parameters effectively ends up being a randomization anyway.

The other mutation operators change the topology of the configuration graph. The *insertion* and *deletion* operators add and delete modules, respectively, while the *module replacement* operator replaces a module and the *permutation* operator exchanges two modules within a configuration. These operators are more complex than the parameter and attachment mutation operators, since they must change the configuration's topology while ensuring that they do not disrupt any `const` attachments. The insertion operator randomly selects a `var` attachment in the configuration graph. If *m* is the module pointed to by the attachment (i.e. the attachment is an incoming edge to *m*), the insertion operator inserts a new link or joint module (with random parameter values) before *m* in the configuration graph. Any `var` attachments that were connected to *m* are changed to connect to the new module, while `const` attachments to *m* remain unchanged. The deletion operator finds a module with at least one `var` attachment to it and only one child, and any `var` attachments to the module are redirected to the module's child. If there are no `const` attachments to the module, then the module is removed from the configuration graph; otherwise, it remains and the `const` attachments are unchanged.

The module replacement operator selects a module with no `const` attachments to or from it, and replaces it with a module of similar general type (base, link, joint, or tool) that has at least as many connectors as were used on the module being replaced (e.g. a module which has attachments to three of its connectors will only be replaced by a module with three or more connectors, though a module that has three connectors, only two of which are used, can be replaced with a module with only two connectors). As with the insertion operator, the parameters of the module being inserted are randomized. The permutation operator finds two modules that are internal nodes in the configuration graph (i.e. have incoming and outgoing attachments), have only `var` attachments to and from them, and have the same number of attachments. The modules are exchanged in the graph, as are their outgoing attachments.

The module replacement and insertion operators both create new modules and insert them into configurations. These modules come from the module database, which is a list of modules specified by the designer. The ranges, numbers of bits, `const` flags, and values for the parameters of the modules in the database are all set by the designer though a database file, thus allowing module properties to be specified in advance if beneficial properties are known. The component context of each module is also set in the database file. When a module from the database is selected for insertion, a copy is made and the copy's `var` parameters are randomized. However, any `const` parameters retain their original values.

3.3 Selecting Configurations for Reproduction and Deletion

Just as the use of genetic operators is a defining property of evolutionary algorithms, so is the idea of fitness-based selection. This is based on the principle of natural selection: organisms that are somehow more 'fit' than other, competing organisms will be able to reproduce more, and thus will have a greater influence on the gene pool of the species. In nature, the selection process is automatic: those organisms that reproduce most before dying will have the greatest influence. In an evolutionary algorithm, an artificial means of rewarding 'fit' solutions is required: when selecting solutions to be reproduced, those that perform better are more likely to be selected. Additionally, steady-state evolutionary algorithms like Darwin2K require a method of selecting solutions for deletion to make room in the population for new solutions. The method used for selection (for both reproduction and deletion) plays a large role in determining the performance of the evolutionary algorithm, particularly when multiple metrics are used to measure the fitness of solutions.

The final selection methodology used in Darwin2K is, appropriately enough, the result of a number of evolutionary steps. Each step was a decision made in response to an observed shortcoming of the method being used at the time. Certainly, at many junctions different options could have been explored, which would likely have changed the methodology now being used. In retrospect, and in light of evolutionary algorithms in fields other than robot design, the decisions were reasonable and have indeed resulted in an effective system. The organization of this section reflects the design decisions, both in ordering and in content. Briefly, four methods of selection were investigated. Initially, a simple weighted sum was used to create a single selection probability from multiple performance metrics. This method often favored unacceptable solutions over acceptable ones; to overcome this problem, two alternate methods were formulated: the Configuration Decision Function, which allows the designer to specify a method of comparing two solutions to decide which is better; and the Metric Decision Function, which the designer specifies to indicate which metrics should be used for selection based on properties of the population. The Metric Decision Function was the more promising of the two methods, but was not intuitive for the designer. Requirement Prioritization was a response to the non-intuitive interface of the Metric Decision Function and is the method now used in Darwin2K. Requirement Prioritization is based on specifying a priority and acceptability criteria for each metric, which the synthesizer uses to decide which metric to use for selecting configurations. This method is easy for the designer to use and can efficiently guide the synthesizer in optimizing multiple metrics.

3.3.1 Basic formulation of selection

A common method for performing selection in evolutionary algorithms is *fitness-proportionate selection*. This entails computing the fitness (performance) of each solution and then probabilistically selecting a solution from the population, where a solution's chance of being selected is directly proportional to its fitness. This is the primary method used in *Adaptation in Natural and Artificial Systems* ([Holland75]) and *Genetic Programming* ([Koza92]), two important texts on evolutionary algorithms; a modified version of it is used in Darwin2K.

In fitness-proportionate selection as formulated in [Koza92], a solution's performance (called the *raw fitness*) is translated into a scalar called *standardized fitness.* This is a number greater than zero, with zero being the best possible value. In Darwin2K, each type of fitness measurement (performance metric) performs this conversion internally (See Section 4.8 for details on how standardized fitness is computed for various metrics), though the general idea is:

$$s(i) = scale \times \begin{cases} max - raw\ fitness & \text{positive sense} \\ raw\ fitness - min & \text{negative sense} \end{cases} \tag{3.1}$$

where positive sense indicates that a larger raw fitness is better, and negative sense indicates that smaller raw fitness is better. Negative standardized fitness values are clipped to zero. The standardized fitness is then translated into *adjusted fitness* by the equation

$$a(i) = e^{-s(i)} \tag{3.2}$$

where $s(i)$ is the standardized fitness of configuration i and $a(i)$ is the adjusted fitness of the same configuration. Adjusted fitness ranges from 0 to 1, with 1 being best. For example, two configurations with standardized fitness values of 5 and 6, and another two configurations having standardized fitness 0 and 1, will have adjusted fitness values of 0.0067 and 0.0024, and 1 and 0.37 respectively. The difference in standardized fitness is 1 in both cases, and the ratio between adjusted fitness values is the same as well: $1/e$. This provides a consistent selection pressure for better-than-average configurations over the whole range of standardized fitness values, and assigns high selection probability to solutions with significantly better-than-average standardized fitness. The scale value for a particular metric and task can be chosen to provide a multiplication in selection probability proportional to the incremental change in the raw value of the metric. For example, if we want a 5kg reduction in mass to lead to a twofold increase in adjusted fitness, we would use the formula

$$scale = \frac{\ln(r)}{\Delta m} = \frac{\ln(2)}{5} = 0.138 \tag{3.3}$$

where r is the desired ratio of increase in adjusted fitness for an improvement of magnitude Δm in the raw value of the metric.

The selection probability of a configuration is directly proportional to its adjusted fitness; the actual probability of selection (also referred to as the *normalized fitness*) is the adjusted fitness normalized by the sum of the adjusted fitness values of all configurations in the population:

$$n(i) = \frac{a(i)}{\sum_{j=1}^{n} a(j)} \tag{3.4}$$

Though the normalized fitness is not stored explicitly for each configuration in Darwin2K, it is computed each time a selection is made. When selecting a configuration for deletion, the probability of selection is proportional to the inverse of the adjusted fitness:

$$d(i) = 1/a(i) = e^{s(i)} \tag{3.5}$$

and again, the actual probability of selection is normalized over the entire population:

$$r(i) = \frac{d(i)}{\sum_{j=1}^{n} d(j)} \tag{3.6}$$

In some work such as [Koza92], a different formulation of adjusted fitness is used:

$a(i) = 1/(1+s(i))$. However, this did not provide a consistent selection pressure throughout the synthesis process, especially when the best possible raw fitness was not known in advance. If it turns out not to be possible to achieve a standardized fitness of 0, then there may be little selection pressure using the *1/(1+s(i))* formulation. On the other hand, using the exponential formulation for adjusted fitness means that an increase in standardized fitness of constant value will always result in an increase in adjusted fitness of a constant multiple, which can consistently provide large selection pressure towards solutions that outperform the rest of the population. While increased selection pressure can cause an evolutionary algorithm to converge *too* quickly and settle into a part of the search space that is far from optimal, better results have been achieved using the exponential formulation for adjusted fitness.

Consider the case where the metric being optimized is the mass m of a robot. If we do not know the best (minimum) mass that a feasible configuration can achieve, we could make a reasonable but conservative guess--let's say it is 5kg for a certain problem. Thus, if we choose a scale factor of 1 then the standardized fitness for a mass m would be m-5. If one configuration has a mass of 15kg and another has a mass of 15.5kg, then with the exponential formulation the lighter one will be 64% more likely to be selected for reproduction. With the formulation in [Koza92], the lighter one will be 5% more likely to be selected. If it turns out that the best possible mass is 10kg, let us examine the case where the synthesizer succeeds in generating the optimal configuration. If we have a configuration with mass 10.5kg, and another with mass 10kg (which is optimal, though the synthesizer doesn't know it), then with the exponential formulation the lighter one will still have a 64% greater selection probability for the 0.5kg difference in mass. Using the formulation of adjusted fitness from [Koza92], the lighter configuration will only have a 10% greater chance of being selected--even though it has achieved the optimal value. The exponential formulation is thus able to consistently differentiate between configurations when the optimal value of a metric cannot be accurately specified in advance. Additionally, the exponential formulation of adjusted fitness provides a consistent selection pressure when selecting configurations for deletion.

It is worth noting that the scale factor used in converting raw fitness to standardized fitness directly controls the multiplicative increase in selection probability per linear decrease in standardize fitness. In the absence of better information, a rule of thumb for choosing a scale factor is that the range of variation in a metric's raw value should roughly map to the interval [0,5] in standardized fitness; in practice this has worked well enough to preclude a detailed study of the best way to select the scale factor.

The fitness-proportionate selection method described above is fine when the fitness of a configuration can be measured by a single metric. However, the vast majority design problems are characterized by multiple, often-conflicting performance metrics, some of which can be considered *task requirements*: there is a certain threshold a solution must meet to be considered *feasible.* General examples of conflicting metrics are cost versus performance, mass versus capability, and speed versus accuracy.

The easiest and most common way of handling multiple metrics is *scalarization*: collapsing a vector of metrics into a scalar using some sort of scalarizing function. All previous approaches for robot synthesis performed scalarization by taking some function of a weighted sum of metrics:

$$f_{combined}(i) = g\left(\sum_j w_j f_j(i)\right) \quad (3.7)$$

where w_j is the weight for the jth metric and $f_j(i)$ is the fitness for metric j of configuration i, and g is a function (often the identify function, or an exponential). Darwin2K initially used a weighted sum

$$f(i) = \sum_j w_j a_j(i) \quad (3.8)$$

but the usual problems were encountered: choosing appropriate values for w_i was nontrivial, and the synthesizer frequently assigned (based on the weighting scheme) the highest selection probability to designs that were inferior to others in the population.

A scalarization approach used for design synthesis will have shortcomings if it does not account for the nonlinear dependencies between metrics. The fundamental problem with simple scalarization is that it does not capture the relative merits of different designs as perceived by the designer -- and it is the designer's goals that the synthesizer is trying to achieve. As a trivial example, consider the following: show a person two cars, one that gets 40 miles to the gallon and can be driven for thousands of miles between repairs, and one that gets 300 miles to the gallon yet whose wheels can only rotate through one revolution. If you ask the person which one they would rather drive, they are likely to pick the first one. The person will not assign weights to mileage and maximum distance traveled and calculate a function from the weighted performance metrics. The thought process is probably more akin to asking the questions, "Does each car meet my minimum requirements? If so, which performance trade-offs are most desirable to me?" When using scalarization methods that do not explicitly account for the significance of metrics, there will always be cases where a design that is *in the designer's eyes* inferior to another design will appear superior according to scalarization. In practice, these cases occur quite frequently and can cause the synthesizer to concentrate on inferior designs. A better approach for optimizing multiple objective functions *for design* is to somehow encode the significance of each metric and any conditional dependencies between them, so that the synthesizer can make better decisions. Two possible ways of doing this are to tell the synthesizer how to meaningfully compare configurations, and to tell the synthesizer which metric(s) should be used for selection at a given time. Both approaches were investigated before focusing on the latter, as it is easier for the designer to specify and improves the effectiveness of the elitist approach outlined below.

The next section describes the development and implementation of both approaches, but before delving into them it is useful to introduce the concept of *elitism*. In a generational EA, elitism means always duplicating the best few solutions when the next generation is created; in a steady-state algorithm, elitism always preserves the best few solutions. Elitism prevents an EA from losing the best solutions found to date and can improve convergence toward good solutions by giving them more opportunities for reproduction.

When a single objective function is used, the notion of 'best' is obvious; when multiple objectives are being optimized the notion of "best" must be interpreted appropriately. The simplest way is to consider the "best" solutions to be those that have the best fitness in any single metric over the entire population; however, this is a very limited definition as it does not consider solutions which make trade-offs between metrics. A more useful definition of the set of "best" is the *Pareto-optimal* set. Pareto-optimality can be understood in terms of dominance:

- solution s_a is said to *dominate* solution s_b if s_a is better than s_b with respect to one or more metrics, and is equal to s_b in the remaining metrics.

The Pareto-optimal set is the set of solutions that are not dominated by any other solution; this set contains solutions which have varying trade-offs between metrics. This is the basis for Darwin2K's elitist strategy: configurations in the Pareto-optimal set should not be deleted. In practice, the size of the Pareto-optimal set can easily approach the size of the population, making it difficult to find configurations to delete (and ultimately requiring the population size to be increased if we are to avoid deleting Pareto-optimal configurations). This becomes more pronounced as the number of metrics increases. To alleviate this, feasibility is added as a requirement for elitism: if any configurations in the population are feasible according to the task requirements, then only those configurations that are in the Pareto-optimal set of the *feasible* solutions are prevented from being deleted. A configuration is said to be feasible if it meets the acceptance criteria for (or *satisfies*) each metric. We thus define the *elite set* as follows:

$$E = \begin{cases} PO(P) \text{ if } |\mathrm{F}| = 0 \\ PO(F) \text{ if } |\mathrm{F}| > 0 \end{cases} \tag{3.9}$$

where E is the elite set, P is the population, F is the set of feasible configurations from P, and $PO(x)$ is the Pareto-optimal set of x. E can also be understood in terms of a modified dominance relationship:

- solution s_a is said to *feasibly dominate* solution s_b if s_a is feasible and s_b is not, or if s_a dominates s_b and both solutions have equal feasibility.

The elite set is thus the set of solutions that are not feasibly dominated by any other solutions. Note that feasibility and domination are understood to be defined with respect to a set of metrics. Basing elitism on the elite set as defined here, rather than on the Pareto-set, significantly reduces but does not eliminate the need to increase the population size to accommodate an expanding set of elite solutions.

3.3.2 Configuration Decision Functions

One way to encapsulate the significance of, and dependencies between, metrics is for the synthesizer to ask the designer, "Which of these two configurations is better?" While it is not feasible to do this for every choice made by the synthesizer, we can ask the designer to specify a *configuration decision function* (CDF) that attempts to capture the designer's decision process for comparing two configurations. For the automobile example in the previous section, the CDF might be something like:

Is each car capable of taking me to work every day? If so, the one with better gas mileage is best. If not, then the one that takes me the farthest is best. If they have equal ranges, the one with better mileage is best.

While we could ask the designer to write a snippet of C++ code which would be compiled into the system, this would be inconvenient when making changes and in any case a fully-featured language is not required. Instead, the CDF is specified in a C-like format that is interpreted at runtime, which is more concise and convenient for the designer. Several primitives are required:

- comparison: $a > b$ (or <, =, >=, etc.)
- conditional: `if-then-else`
- boolean operations: `AND`, `OR`, `NOT`
- decision statement: cfg_a is better, or cfg_b is better
- values: metric i for cfg_a, numerical constants
- arithmetic operators: +, -, /, etc.

The UNIX tools *yacc* and *lex* [Levine92] were used to create an interpreter for the CDF with the primitives listed above. A sample CDF specification file is shown in Figure 3.5. This CDF sequentially compares the *raw* metric values of the two configurations, in an order that indicates the relative importance of each metric: if one configuration completes more of the task than the other, then that configuration is *always* considered better. The error metric is only used to decide between configurations when one or both has error greater than a threshold; differences in error below that threshold are not to be considered significant. Actuator saturation and link deflection are treated similarly.

The CDF tells the synthesizer how to decide which of two configurations is better, but does not compute an explicit fitness value that can be used for fitness-proportionate selection. Instead, *tournament selection* [Goldberg91] is used with the CDF. Tournament selection randomly chooses two configuration and uses the CDF to select the better of the two for reproduction (or the worse of the two, for deletion). When used with a steady-state EA, tournament selection provides an inherent, though small, level of elitism: clearly, the best solution will never be selected for deletion. One property of tournament selection is that it does not create as much selection pressure towards good configurations as fitness-proportionate selection does. This is a trade-off: on one hand, the EA is less susceptible to premature convergence, but on the other hand it may take longer to reach a good solution. Initial experiments indicated that using tournament selection with a CDF decreased the likelihood of the synthesizer getting trapped in a local minima as compared to the weighted sum approach.

The grammar used for the CDF can also be used to specify the *feasibility decision function*, which can be used to limit the elite set as discussed in Section 3.3.1. The simplest form of the Feasibility Decision Function compares one or more of a configuration's raw fitness values to a threshold to determine if it is feasible; however, more complex functions can be specified if desired.

Rank selection [Goldberg91] can also be used with the CDF, though it was not in-

```
// first, decide on percentage of task completed (metric 0)
if (cfg1->metric[0] > cfg2->metric[0]) return cfg1;
else if (cfg2->metric[0] > cfg1->metric[0]) return cfg2;

// choose cfg w/ fewer collisions (metric 1)
if (cfg1->metric[1] < cfg2->metric[1]) return cfg1;
else if (cfg2->metric[1] < cfg1->metric[1]) return cfg2;

// decide on error (metric 2) only if > 3cm
if (cfg1->metric[2] > 0.03 || cfg2->metric[2] > 0.03) {
 if (cfg1->metric[2] < cfg2->metric[2]) return cfg1;
 else if (cfg2->metric[2] < cfg1->metric[2]) return cfg2;
}

// decide on link deflection (metric 3) only if > 5mm
if (cfg1->metric[3] > 0.005 || cfg2->metric[3] > 0.005) {
 if (cfg1->metric[3] < cfg2->metric[3]) return cfg1;
 else if (cfg2->metric[3] < cfg1->metric[3]) return cfg2;
}

// decide on actuator saturation (metric 4) only if > 0.5
if (cfg1->metric[4] > 0.5 || cfg2->metric[4] > 0.5) {
 if (cfg1->metric[4] < cfg2->metric[4]) return cfg1;
 else if (cfg2->metric[4] < cfg1->metric[4]) return cfg2;
}

// decide on time (metric 5)
if (cfg1->metric[5] < cfg2->metric[5]) return cfg1;
return cfg2;
```

Figure 3.5: Example CDF file

This CDF essentially prioritizes different metrics: path completion is most important, followed by number of collisions, error, link deflection, joint torque, and task completion time. Note that error, link deflection, and joint torque all have acceptability criteria greater than zero, so comparison is only performed if one or both configurations do not meet the criteria. The raw fitness (rather than standardized) is used for each metric comparison.

---

vestigated. In rank selection, all solutions in the population are sorted based on fitness and are then assigned selection probabilities based on their rank. Since sorting relies on comparing two elements in a set, the CDF can be used for the sorting operation. However further investigation of the CDF did not seem as useful as exploring methods based on fitness-proportionate selection. Modifying Darwin2K to perform rank selection was costlier than extending fitness-proportionate selection, so rank selection was not explored.

3.3.3 Metric Decision Functions and Requirement Prioritization

If we wish to use fitness-proportionate selection for multiple metrics, we cannot use the CDF. While an improved scalarization approach that uses conditional logic may be appropriate, we can also try to guide the synthesizer by telling it which metrics it should be optimizing at a given point in time. The motivation for this was the observation that if there is little variation in a particular metric, then making selections based on that metric is not very meaningful: in the limit, if all solutions have the same value in one metric, then selecting based on that metric amounts to uniformly random selection. 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; however, initial experiments with this approach were not promising. One reason is that different metrics have different ranges--sometimes by orders of magnitude; 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. The *metric decision function* (MDF) thus arose in an attempt to answer the question, “What variation in a metric is meaningful in terms of the task’s requirements?”

As with the Configuration Decision Function, the MDF is specified by the designer in a text file that is interpreted at runtime; however the MDF has slightly different primitives since it deals with properties of the population as a whole, and selects metrics rather than configurations. One primitive is the *fraction of population comparison* (FPC). This computes the fraction of the population that has a specific metric less than, equal to, or greater than a threshold. For example, if 90% of the population has a task completion metric of 100%, then the synthesizer shouldn’t select based on task completion. The other key primitive for the MDF is the metric selection statement, of which there are two types: a fixed metric selection which returns a specific metric, and a weighted random metric selection, which chooses a metric based on a set of weights, in which case a metric’s chance of selection is directly proportional to the weight it is assigned. The initial formulation of MDFs used these primitives as well as the operators and conditional statements used for Configuration Decision Functions. While effective at optimizing multiple metrics, specifying a useful MDF in this way was not intuitive and often required the designer to specify many if-then statements, as shown in Figure 3.6. To make MDF specification easier for the designer, a weight assignment statement was added which allowed the selection probabilities for each metric to be set independently, thus reducing the complexity of the MDF file. Figure 3.7 shows an MDF equivalent to the one in Figure 3.6, but with the use of the weight assignment statement.

The structure of the MDF is more visible in Figure 3.7: path completion and collision avoidance are emphasized first, but once most of the population performs well for those metrics, the priorities of some other metrics (maximum error, link deflection, and peak joint torque) are increased. Finally, when error and link deflection reach acceptable levels for most of the population, peak joint torque and task completion time alone are

```
// first focus on path completion (metric 0)
if (fpc(metric[0] > 0.9) < 0.5) {
  // less than half the population has completion > 0.9; improve it
  if (fpc(metric[1] == 0.0) < 0.5) {
    // need to improve collisions (metric 1)
    // give weights of 1 to path completion (metric 0) and
    // collisions (metric 1), and give weights of 0.1 to all other
    // metrics
    return random(1, 1, 0.1, 0.1, 0.1, 0.1);
  } else {
    // need to improve completion, but not collisions
    return random(1, 0.1, 0.1, 0.1, 0.1, 0.1);
  }
} else if (fpc(metric[1] == 0.0) < 0.5) {
  // population is okay with respect to task completion, but
  // less than half can complete the task with no collisions.
  // give a weight of 1 to collisions and 0.1 to other metrics
  return random(0.1, 1, 0.1, 0.1, 0.1, 0.1);
}

// path completion and number of collisions are ok, so focus on
// other metrics
if (fpc(metric[2] > 0.03) < 0.5) {
  // optimize error (2)
  if (fpc(metric[3] < 0.005) < 0.5) {
    // link deflection (3) needs improvement
    if (fpc(metric[4] < 0.5) < 0.5) {
      // need to reduce actuator saturation (4), too
      return random(0.1, 0.1, 1, 1, 1, 0.1);
    }
    return random(0.1, 0.1, 1, 1, 0.1, 0.1);
  }

  if (fpc(metric[4] < 0.5) < 0.5) {
    // improve actuator saturation (4)
    return random(0.1, 0.1, 1, 0.1, 1, 0.1);
  }

  return random(0.1, 0.1, 1, 0.1, 0.1, 0.1);
}

// keep trying to reduce actuator saturation (4) and time (5)
return random(0.1, 0.1, 0.1, 0.1, 1, 1);
```

Figure 3.6: Sample MDF specification file

The `random` function returns a metric chosen randomly based on the weights specified in its arguments. For example, `random(0.9, 0.1)` will return metric 0 90% of the time and metric 1 10% of the time. Note that `random` normalizes the weights so that they do not have to sum to one as specified by the designer.

```
weights = 0.1;  // assign 0.1 to all weights

if (fpc(metric[0] > 0.9) < 0.5 ||
    fpc(metric[1] == 0.0) < 0.5) {
  // focus on path completion (metric 0) & collisions (metric 1)
  if (fpc(metric[0] > 0.9) < 0.5)  weights[0] = 1;
  if (fpc(metric[1] == 0.0) < 0.5)  weights[1] = 1;

  return random(weights);
}

if (fpc(metric[2] > 0.03) < 0.5) {
  // need to improve error (metric 2)
  weights[2] = 1;

  // see if link deflection needs improvement
  if (fpc(metric[3] < 0.005) < 0.5) weights[3] = 1;

  // reduce actuator saturation if necessary
  if (fpc(metric[4] < 0.5) < 0.2) weights[4] = 1;

  return random(weights);
}

// keep trying to reduce actuator saturation (4) and time (5)
weights[4] = 1;  // actuator saturation
weights[5] = 1;  // time

return random(weights);
```

Figure 3.7: MDF specification using weight assignment

assigned high weights. Soon after implementing this change, it became apparent that the weights could be adaptively set in a more continuous manner: for example, the selection probability for a metric could be proportional to the fraction of the population that did not satisfy the metric. This approach seemed to work well and made specifying the MDF more straightforward. Note that the Feasibility Decision Function mentioned in Section 3.3.2 is also used with the MDF to limit the size of the elite set based on feasibility.

While initial experiments with the MDF were promising, the interface was awkward and required the designer to make choices in specifying the MDF that in practice would likely be made arbitrarily, since there was no clear guide for how to translate task requirements into an MDF. At the same time, an idiom seemed prevalent in the MDFs that had been used successfully: high-priority metrics were optimized first, and lower-priority metrics were only considered after some configurations satisfied the high-priority metrics.

Requirement Prioritization thus arose as a way of formalizing this idiom to capture the relative importance of different metrics in a way compatible with fitness-proportionate selection. For each metric, the designer assigns a priority and an optional acceptance threshold in terms of raw fitness. The priority is simply an integer indicating the importance of the metric relative to the other metrics, and more than one metric may share the same priority. This is far simpler than specifying a full MDF: no rules must be specified, and in contrast to the many weights and thresholds required for the MDF, the only numerical constants that Requirement Prioritization needs are in terms of the task: the importance of each metric, and a feasibility threshold:

```
priority = 0   metric 0   threshold: == 1.0   // task completion
               metric 1   threshold: == 0.0   // # collisions

priority = 1   metric 2   threshold: < 0.03   // error
               metric 3   threshold: < 0.005 // deflection
               metric 4   threshold: < 0.5    // saturation

priority = 2   metric 5   threshold: none     // time
```

This is clearly less involved than specifying a MDF or a CDF and implicitly includes the FDF as well. (Note that a *lower* value of priority indicates *higher* importance.)

Darwin2K uses Requirement Prioritization to adaptively set metric selection weights. Starting with the metrics having priority 0 (i.e. most important), each group of equal-priority metrics (called a *requirement group*) is optimized in turn. Let us denote the group of metrics with priority p as g_p. The set G_p is then defined to be the set $\{g_0, g_1, \ldots, g_p\}$. We say that g_p is *satisfied* when at least one configuration meets the acceptance criteria for each of g_p's metrics, and that G_p is satisfied when its members are all satisfied by a single configuration. Once the population contains a certain number of configurations which satisfy G_p, Darwin2K moves to G_{p+1}. When optimizing for G_p, the probability of choosing a metric of priority > p is zero; for metrics in G_p, the selection probability is directly proportional to a weight w assigned to each metric by the rule

$$w = \begin{cases} 2 & \text{if metric } i \text{ is not satisfied} \\ 1 - f_i & \text{if metric } i \text{ is satisfied} \end{cases} \tag{3.10}$$

where f_i is the fraction of the population that does not satisfy metric i. The effect of this process is that less-important metrics are not addressed until the more-important ones are satisfied, and metrics that are satisfied by few configurations are given more importance than those that are satisfied by many. For example, it makes no sense to select configurations based on the time they spend on a task if none can complete it; thus, task completion is assigned a lower priority number (higher importance) than task execution time. Additionally, if most of the population can complete a task, but few can do it without any collisions, then it is more useful to select configurations based on number of collisions than on task completion.

As mentioned earlier, the size of the Pareto-optimal set can easily approach the size of the population if elitism is used, thus requiring that either Pareto-optimal configurations be deleted, or that the population size be increased. Also noted was that feasibility can be used to limit the number of configurations in the *elite set*. Requirement Prioritization can further reduce the size of the elite set in two ways: by reducing the number of metrics that are used to determine Pareto-optimality, and by incrementally adding feasibility requirements for those groups of metrics that have been satisfied. If the group of metrics currently being optimized has priority p, then the set of metrics M used to determine Pareto-optimality are determined as follows:

- if G_{p-1} is not satisfied, then M contains the metrics in G_p's members
- if G_{p-1} is satisfied then M contains only g_p's metrics

It may seem that G_{p-1} will always be satisfied when optimizing for G_p, but this may not be true if the metrics in g_p require a new simulation method that may more accurately compute values for metrics in G_{p-1}. An example of this is when G_p contains the `dynamicPathCompletion` metric, which requires dynamic simulation to measure task completion. In this case, the task completion as measured earlier in kinematic simulation may be inaccurate, and no existing configurations may meet G_{p-1}'s feasibility thresholds when re-evaluated with dynamic simulation. After a new configuration c has been evaluated, it is tested to see if it satisfies G_p. M is determined as above and the elite set E is recomputed using the *feasibly dominant* relationship with respect to M (as defined in Section 3.3.1) to determine membership in E. Figure 3.8 shows the algorithm used to update the elite set.

Darwin2K uses adjusted fitness rather than raw fitness to determine dominance. Raw fitness can take on any value, but it is clipped to a limit specified by the designer before computing the standardized and then adjusted fitness. This allows the designer to indicate that raw fitness values above or below a certain threshold are equivalent in terms of their relevance to the task, since any raw fitness values past the threshold will be as-

```
procedure updateOptimalSet(configuration c, set O)
   boolean wasDominated = 0;

   for c_old in O {
      if c feasibly-dominates c_old then
         remove c_old from O;
      else if c_old feasibly-dominates c then
         wasDominated = 1;
         exit loop;
      endif
   }

   if (wasDominated = 0) then
      add c to O;
   endif
end
```

Figure 3.8: Algorithm for updating the elite set

Each time a configuration is added to the population, this function updates the elite set according to the feasibly dominant relationship described in Section 3.3.1.

signed the same adjusted fitness.

Requirement Prioritization is effective for addressing robot design problems with multiple metrics, and it is the method used in most of the experiments documented in Chapter 5. The ability of Requirement Prioritization to effectively capture task-specific dependencies between metrics, and to do so in a manner that is very easy for the designer to specify, make it attractive for synthesis. Additionally, Requirement Prioritization incrementally adds feasibility conditions on the elite set, creating a good trade-off between preserving configurations that perform well and having to accomodate an unmanageably large set of elite solutions.

3.4 Synthesis Process

We have just seen how Darwin2K selects configurations for reproduction and deletion, and how it creates new configurations using the genetic operators. While these are the core operations performed by the Evolutionary Synthesis Engine, there are other auxiliary supporting procedures that round out the system. These include generating an initial population, deciding which genetic operators to use, controlling the size and diversity of the population, deciding when to move to the next group of metrics, and communicating with evaluation processes.

3.4.1 Generating configurations

The synthesis process begins with the initial population. This can be the final population of a previous synthesis run, but more frequently it is created using the genetic operators. However, the genetic operators can only combine and modify existing configurations; they do not create configurations from scratch. Thus, one or more initial *kernel configurations* are required for the operators to work on. These kernels are specified by the designer and can be as trivial or complex as desired. If the designer knows some parametric or topological features that will be useful to the synthesizer, he can provide them as kernel configurations and can optionally ensure that the features remain present in every configuration generated by the synthesizer through the use of `const` flags on parameters and attachments. On the other hand, if nothing is known about properties of configurations that will be useful for the task, then the kernel can be trivial, consisting of a tool module attached to a base module. Partial or complete solutions from previous synthesis runs can be included as well; for example, if a particularly well-optimized assembly of modules forming a wrist was discovered by Darwin2K in a previous run, then it can be included as part of a kernel.

Figure 3.9 shows a data flow diagram for the creation of the initial population. The process works by randomly applying genetic operators to configurations randomly selected from a pool. Initially this pool contains only the kernels, but each newly-generated configuration is added to the pool. The process of expanding the pool may have one or two passes, depending on whether the designer specifies any *configuration filters.* These filters provide an inexpensive way of limiting the search space by culling inappropriate designs before they are evaluated. As with other aspects of Darwin2K, the designer can create task-specific filters as part of a dynamic library; however Darwin2K includes three general purpose filters: the `dofFilter`, the `endPointFilter`, and the `moduleRedundancyFilter`. The `dofFilter` culls any configurations having a number of degrees of freedom outside a specified range; the `endPointFilter` is similar but checks for an allowable number of endpoints (tools). The `moduleRedundancyFilter` removes configurations having two forbidden pairs of modules connected together; the designer provides a list of pairs of modules (and their specific connectors, if desired) that should not be attached to together. The main purpose of the `moduleRedundancyFilter` is to prevent kinematically redundant modules from following each other: for example, two joint modules that perform a twist motion should not be connected to each other, since their joint axes will be colinear. This filter is a generalized version of the Module Assembly Preference in [Chen95].

During the first pass (which is the only one if no filters are used), every configuration that is generated is added to the pool. Half the time, a single parent configuration is selected at random from the pool, and is duplicated; the other half of the time, two parents are selected at random and the module crossover operator is used to create two new configurations. Mutation operators are then applied to the new configurations, with each operator having a 50% probability of being applied to each new configuration. The non-`const` module and task parameters of each new configuration are then randomized, and then each offspring is added to the pool. This continues until the pool reaches a desired size, or until more than two configurations have been generated and a time-out is reached (since it can be difficult to create new configurations if the filters are very restrictive or if the kernels have many `const`-flags set). If filters are being used, then any configurations in the pool that do not pass filtering are removed, and the generation process is repeated for a second pass. During the second pass, only those configurations that pass filtering are added to the pool.

Upon completing the generation of the initial population, all of the configurations are put into the *evaluation queue* and are in turn sent to the available evaluation processes; this begins the synthesizer's main loop, as shown in Figure 3.10. Many evaluation processes can be run simultaneously on different workstations to reduce system runtime; typically, between 10 and 40 computers were used in this work. The synthesizer's event

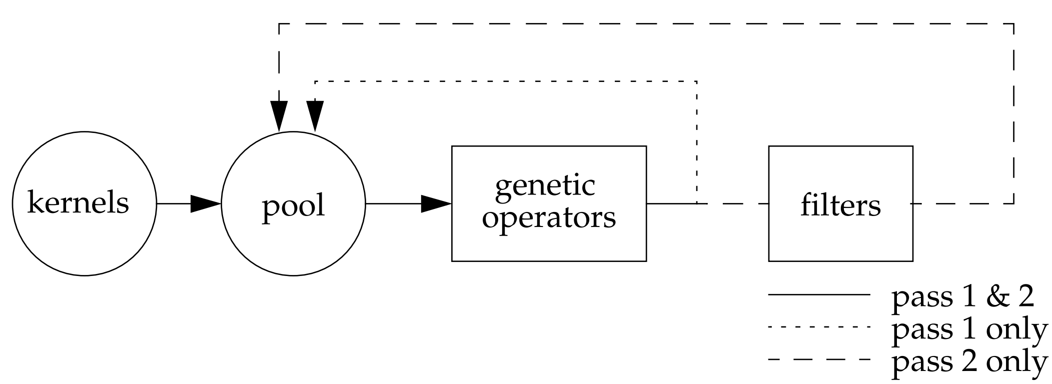
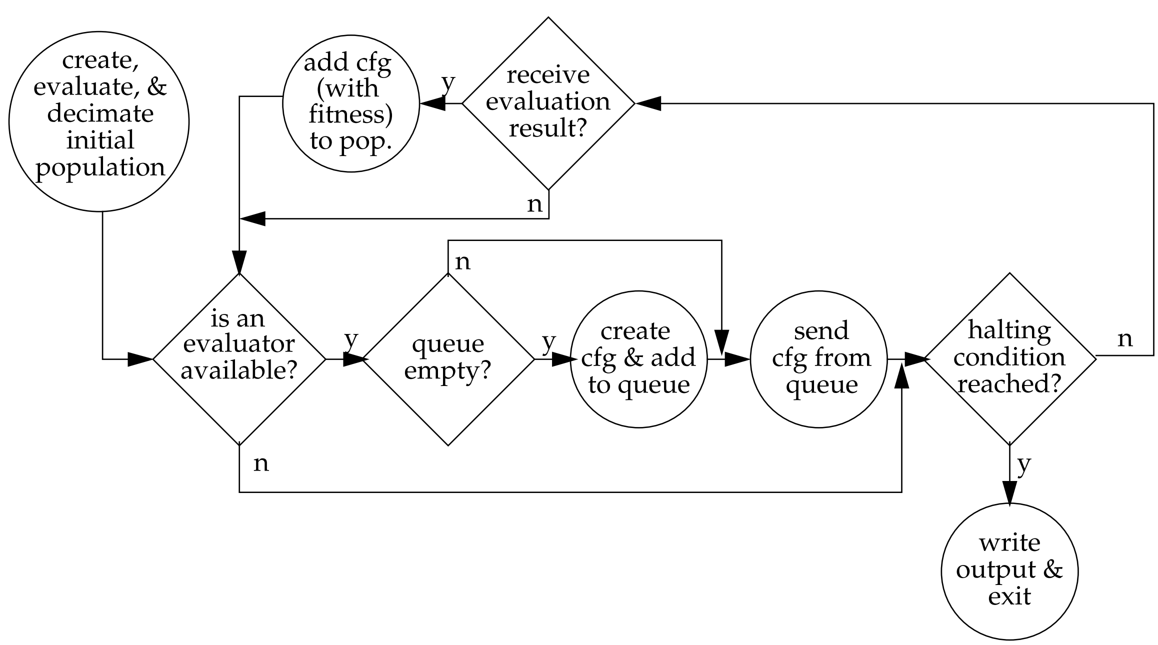


Figure 3.9: Data flow for generating initial population

loop monitors messages from the evaluation processes, sending them configurations when they are ready and later receiving the results. The evaluation queue contains configurations that have not yet been evaluated. These configurations can only be added to the population once their fitnesses are known, since fitness measurements are required when selecting configurations from the population. Whenever an evaluator sends a message indicating that it is free, the synthesizer removes several configuations from the queue and sends them to the waiting evaluator. Initially, the queue contains only the configurations generated from the kernels. To provide an adequate diversity of genetic material for the synthesizer, the number of initial configurations can be much larger than the nominal population size used during the bulk of the synthesis process. After the queue has been emptied of the initial configurations and all results have been received, configurations are selected for deletion and are removed from the population until the desired population size is reached. This is called *decimation* ([Koza92]) since most of the initial population is culled after it has been evaluated.

For the remainder of the synthesis process, the synthesizer creates configurations whenever the evaluation queue is empty, and deletes configurations from the population to make room for new configurations. When creating configurations, the synthesizer chooses which genetic operators to use based on probabilities set by the designer. The designer can independently set the probabilities for the crossover operators (module, parameter, and commonality-preserving), with the probability of duplication being computed as one minus the sum of the crossover probabilities. Mutation operators are applied independently from crossover and duplication, and have much smaller probabilities since their effects can be disruptive if used continuously. Additionally, the synthesizer adjusts the probability of mutation based on an estimate of how stagnant the population is. The reason for this is that mutation is most useful when the population has reached a local optima and requires new genetic material to escape it. In early experi-

Figure 3.10: Flow chart for ESE control flow



ments it was observed that the population could quickly escape a local optima if the mutation probabilities were temporarily increased via the synthesizer's user interface. To automate this process, the synthesizer increases the probability of mutation based on the time since a configuration was added to the elite set. The probability of application p_i for each mutation operator is computed from the operator's base probability $p_{i,base}$ and a time constant c specified by the designer such that the total probability of mutation (over all mutation operators) increases exponentially with the number of configurations generated since the last improvement in the elite set:

$$P(mutation) = 1 - \prod_i (1 - p_i) = \exp\left(\frac{t - t_{last}}{c}\right) \tag{3.11}$$

where t is the number of the current configuration being generated, t_{last} is the number of the last configuration to be added to the elite set, and $p_i = \alpha \times p_{i,\,base}$. α is precomputed for a range of total mutation probabilities, and the maximum mutation probability can be specified by the designer. While a more accurate measure of stagnation might directly measure the diversity of configurations, rather than a plateau in their performance, this simple approach seems to work adequately. One interesting property of the synthesizer is that since a large number of good configurations can be preserved via the elitist strategy mentioned previously, very high mutation rates can be tolerated without undoing the progress already achieved.

To avoid deleting configurations in the elite set, the synthesizer must occasionally increase the size of the population. It does this whenever the size of the elite set is 90% of the size of the population; in this case, it increases population size by 10%. This typically happens when optimizing the final group of metrics; surprisingly, the increased population size does not appear to adversely affect the rate of improvement in the final group's metrics: the best raw fitness (for each metric) over the elite set continues to improve without much apparent dependence on the growing population size. One possible explanation is that the optimal configurations that make trade-offs (instead of having one extremely good metric) have small selection probability compared to those feasible configurations at an extreme, due to the exponential nature of adjusted fitness. Thus, the configurations with significantly better-than-average performance in one or more metrics will continue to drive the search, as they will be selected much more frequently than any of the multitude of trade-off configurations. The trade-off configurations may thus just 'stick around' without significantly affecting the selection probability of the feasible solutions at the extremes. The final stage of the synthesis process is characterized by an ever-expanding set of feasibly-optimal solutions, providing a sampling of a trade-off surface between the different metrics of the final metric group.

3.4.2 Requirement Prioritization and control flow

When Requirement Prioritization is used for selecting configurations for reproduction and deletion, the synthesizer must decide when to move to the next group of metrics. Additionally, the synthesizer can take advantage of the incremental way in which metrics are addressed under Requirement Prioritization. The general idea is that the synthesizer should try to generate a reasonable number of configurations before moving to the next group of metrics, but should not spend too much time in the current group if it has generated at least a few feasible configurations. The synthesizer will advance to the next group of metrics when it has generated a certain number of configurations since creating the first feasible configuration, or when a certain fraction of the population is feasible. The thresholds involved do not appear to be critical. Upon moving to the next group of metrics, the synthesizer first clears the elite set and puts copies of each configuration in the population into the evaluation queue. It is necessary to re-evaluate the population because evaluators only record performance for the metrics that are in the previous priority groups; the fitness for metrics in the new requirement group will not have been measured. While requiring the population to be re-evaluated, this allows evaluators to perform only those computations necessary for the metrics being currently used. This is almost always a good trade-off since the number of re-evaluations is typically a fraction of a percent of the total number of evaluations during a synthesis run, while relatively expensive evaluation methods such as dynamic simulation can be avoided during the first stages of synthesis. In addition to re-evaluating the population when moving to the next group of metrics, the synthesizer also takes the opportunity to expand the search space a bit by creating copies of the configurations that were feasible and randomizing their task parameters. Finally, if the population had been increased past the desired size, configurations are deleted until the desired size is reached. While configurations that were in the elite set may now be deleted (since the elite set was cleared), their duplicates will be evaluated and will re-join the elite set if they remain fit in light of the new metrics.

The synthesizer continues to move through the requirement groups until the last one is reached or until the termination condition is true. The designer can specify three parameters for the termination condition: a time limit, a group limit, and a minimum limit. The synthesizer will never go beyond the time limit, but otherwise will always surpass the minimum limit. The group limit indicates how many configurations should be generated since the last metric group advancement, or since the first feasible configuration for the group was generated. The general idea of the group limit is to keep working towards the last requirement group as long as some progress has been made in the not-too-distant past. In practice, the synthesizer usually reaches the final requirement group without nearing the group or minimum limits and is instead halted by the time limit.

3.4.3 Preserving population diversity

Explicit measures must be taken to prevent a single configuration from dominating the population; otherwise, the population will contain many identical copies of a configuration, which clearly do not contribute any additional information about the search space. Because configurations can potentially persist in the population indefinitely, every time a configuration is duplicated its probability of selection is effectively increased further. This creates a positive feedback loop, frequently resulting in a population that consists mostly of identical configurations. In a generational EA (as opposed to Darwin2K's steady-state EA), this is not a problem: a duplication does not immediately affect the probability of a configuration being reproduced, since the duplicate is added to the next generation instead of to the current population. Darwin2K prevents duplicate configurations from being introduced into the population by searching the population for existing configurations that are identical to each new configuration If the new configuration is a copy, then mutation operators are applied until a novel configuration is generated (or until a time-out is reached). Obviously, comparing every new configuration against every configuration in the population would be quite expensive; to avoid this, a multidimensional hash table is used to drastically reduce the number of comparisons required to ascertain if a configuration already exists in the population.

3.4.4 Using commonality-preserving crossover for encapsulation

In Genetic Programming, the *encapsulation* operator creates a new node from a subtree in a solution so that the subtree is effectively protected from potentially disruptive operations. A related but more powerful concept in Genetic Programming is the *automatically defined function*; this is similar to encapsulation, but it creates a new function that may be treated like any built-in one. Analogous operators in Darwin2K would be collapsing a subgraph of the PMCG into a single module with fixed parameters, and collapsing a subgraph into a module with variable parameters and then adding it to the module database. The *const*-flags of the PMCG make a hybrid solution easy: by setting the const-flags of a subgraph's attachments, the topology of the subgraph becomes fixed (thus immune to disruptive genetic operators), but the parameters can still be varied. However, the subgraph is not actually collapsed into a module or added to the module database, so it is not used by the insertion and replacement operators. This operation is called *subgraph preservation* and is used in conjunction with the commonality-preserving crossover operator. Whenever the Commonality-Preserving Crossover Operator is applied to two feasible configurations and subgraph preservation has been enabled by the user, there is a 50% chance that the subgraph preserved by the CPCO will be fixed via the *const*-flags. When this happens, any *const*-flags of the attachments in the subgraph that are not already `const` are sent to a new value, `subp` (for subgraph-preserving). `subp` is identical in meaning to `const`, with two exceptions: it can be returned to `var` if a future application of the CPCO finds a larger subgraph that should be preserved, and a `subp` attachment is always reset to `var` when the synthesizer advances to the next group of metrics. Subgraph preservation simultaneously protects beneficial topology in the PMCG and helps limit the search space to parametric variation, which is often where improvement will be found once feasible configurations have been generated. However, excessive use of subgraph preservation can lead to premature convergence by restricting the search space too early; this is why it is used sparingly, and why `subp` flags are reset to `var` when moving to the next metric group.

3.5 Summary

Darwin2K uses a distributed evolutionary algorithm to perform robot synthesis. The embodiment of this algorithm, called the Evolutionary Synthesis Engine (ESE), creates an initial population from one or more designer-specified kernels by randomly applying genetic operators, and then evaluates the initial population. After decimating the initial population to yield a population of smaller size, the synthesizer begins selection of configurations for reproduction based on Requirement Prioritization, and creates new configurations from them via the genetic operators. Distributed evaluation processes compute fitness measurements for the new configurations, after which the synthesizer adds the configurations to the population. This process continues until a termination condition is reached.

A key challenge is effective synthesis for multiple requirements and objective functions, and several task-based methods for this were explored before deciding on Requirement Prioritization for its efficiency, task-relevance, and ease of specification for the designer. The Parameterized Module Configuration Graph representation used by the synthesizer lets the designer easily specify partial solutions or properties to the synthesizer and allows topological and parametric synthesis of robots via a range of genetic operators. Several modifications to the basic evolutionary algorithm are included in the ESE, including diversity maintenance, population resizing, indefinite preservation of feasibly optimal configurations, and automatic adjustment of mutation probabilities.

The remaining piece of Darwin2K is the evaluation process. While the synthesizer is responsible for exploring the search space, it cannot do its job without performance estimates for each configuration. Darwin2K's utility is thus ultimately determined not only by the synthesis algorithm but by the efficiency and accuracy of the methods used for evaluation, which are the subject of the next chapter.

4 Robot Evaluation

Darwin2K relies on robot performance measurements to guide the synthesis process. The synthesizer sends configurations to evaluation tasks to measure their suitability for the task, and the evaluators return an indicator of the robot's performance as measured by one or more metrics. It is crucial that the metrics used to measure performance match the needs of the task: the synthesizer attempts to generate configurations that optimize the metrics, so if the metrics do not match the task then the synthesizer will not generate appropriate solutions.

Robot performance is difficult to predict without simulation. For this reason, Darwin2K uses simulation to generate performance measurements to the synthesizer. Simulation allows task requirements to be directly represented through the task description, choice of performance metrics, and acceptability criteria. For example, a task requirement might be "complete the task in less than twenty seconds" or "follow a family of trajectories without exceeding joint torque limits". The alternative to simulation is to use heuristics based on inherent properties of the robot, such as "legs having length between l_{min} and l_{max} are best" [Farritor98]. Heuristic metrics measure how well a robot's properties agree with the designer's expectations of the best properties; they do not measure how well the robot meets the task's requirements[1]. While heuristic methods can reflect the designer's intuition and experience with similar problems, if the designer's expectations are not accurate then the synthesizer will not generate robots that are well-suited for the task. Simulation is a more direct and accurate method of assessing a robot's suitability for a task, and thus allows the synthesizer to generate task-relevant robots.

The simulation method used for evaluation depends on the task representation and requirements. A rover and a manipulator perform different tasks, and may require significantly different simulation methods and task representations. Darwin2K's architecture separates optimization from evaluation, allowing new evaluation methods to be added as new tasks are addressed. At the same time, Darwin2K provides some core simulation and evaluation capabilities that are useful for many design problems. Evaluation methods can thus be quickly implemented and integrated so that the optimizer can be applied to novel synthesis problems.

In addition to enabling task-specific evaluation methods using existing simulation components, extensibility of the evaluation process allows task-specific planning and control algorithms to be used. Robot performance depends on how the robot is controlled, not just the inherent properties of the robot. A general-purpose controller may not be optimal for a new task and may cause a robot's performance to be underestimated. The ability to add new control algorithms to the synthesis toolkit (while using the existing task representations and optimization algorithm) is thus important in ensuring wide applicability of the synthesis system.

During simulation, metrics measure and record aspects of a robot's performance, and then convert the performance data to a standardized form usable by the optimization algorithm. Darwin2K currently provides two simulation methods: kinematic and dynam-

1. For some tasks, some of the robot's properties (such as mass) *are* task requirements. In these cases, it makes sense to use a physical property as a measure of performance.

ic. Many aspects of performance can be measured in a kinematic simulation, where the controller can directly command the acceleration for each of a robot's degree's of freedom. Some non-kinematic measurements such as joint torque and link deflection can be made based on a kinematic simulation, but dynamic simulation is required to estimate performance when inertial forces are important (e.g. in low-gravity environments), when actuators must be operated at their torque limits for a significant portion of a task, or when the robot's motions cannot be predicted from control inputs alone. Dynamic simulation can calculate the motion of a robot given a set of applied forces, thus allowing the robot's performance to be predicted in the presence of actuator saturation, controller error, or significant inertial forces. Darwin2K's kinematic and dynamic simulation methods are applicable to serial-chain and branching manipulators, with provisions made for mobile bases.

The balance of this chapter describes Darwin2K's simulation and evaluation algorithms. The kinematic control and planning algorithms are described first, followed by robot dynamics, link deflection computations, and finally the performance metrics and their interface to the optimization algorithm.

4.1 Simulator Architecture

Darwin2K's simulator architecture is designed to allow task-specific algorithms and task representations to be added while working with existing simulation capabilities. At the highest level, an `evaluator` performs initialization for simulation components (called `evComponents`) and provides high-level simulation control. The `evaluator` class is a base class for other evaluators, and does not include any type of evaluation itself -- other classes such as the `pathEvaluator` can be derived from it to meet the needs of specific task representations. The `evaluator` defines the interface between the simulation internals and the rest of Darwin2K's software, and also keeps track of the configuration being evaluated and the metrics and `evComponents` being used for simulation. The evaluator also parses the initialization file to read its own parameters and to determine which `evComponents` are required, and then passes component-specific information from the file to each `evComponent`. If any task parameters are included in the optimization process, the evaluator passes each task parameter to its corresponding `evComponent` (or interprets the task parameter itself, if one of the `evaluator's` variables was specified as a task parameter). The `evaluator` also calls the appropriate initialization and cleanup functions for each configuration being evaluated and performs other bookkeeping functions. In addition to these functions, classes derived from the evaluator also contain code for controlling the simulation: for example, at each time step in a simulation the `pathEvaluator` tells a `DEsolver` to query a `controller` for a command and then compute the robot's state at the next time step, then checks if any collisions have occurred using the `collisionDetector`, and determines if the robot has completed the task by reaching the end of the current `path`. The `pathEvaluator` is the only general-purpose evaluator in Darwin2K; other evaluators can be created for task-specific needs, while taking advantage of the infrastructure of the base `evaluator` class.

As mentioned above, many simulation capabilities are implemented as self-contained `evComponents` which share a common interface allowing them to work with the `evaluator` class. Table 4.1 lists Darwin2K's general-purpose `evComponents`; the task-specific `evComponents` used in the experiments in Chapter 5 are not shown. (See Appendix A for a complete list of `evaluators`, `evComponents`, and other classes.) Some of the `evComponents` have task variables that can be included in a configuration's list of task parameters so that they can be optimized during the synthesis process; these are listed in the right-hand column for each component.

While many simulation and evaluation capabilities are implemented as `evComponents`, several core functions that require detailed knowledge of module internals are implemented as part of the `configuration` class. These algorithms -- computing dynamic models, link deflections, and Jacobians -- depend on inherent properties of the configuration; thus, it makes sense that the configuration performs these calculations (rather

`evComponent` class	Description	Task variables
`sriController`	Jacobian-based controller used for following endpoint trajectories	singularity thresholds, variables for using redundant DOFs
`ffController`	`sriController` augmented with robot's dynamic model; useful for free-flyers	same as `sriController`
`pidController`	joint-space PID controller	none
`path`	represents an end-effector trajectory in 3D	linear and angular velocity and acceleration
`relativePath`	as `relativePath`, but trajectories are relative to a moving coordinate system	same as path
`payload`	geometric and inertial model of unarticulated payloads	initial position and orientation
`rungeKutta4`	integrates robot state during simulation	none
`collisionDetector`	detects self-collisions and collisions with obstacles	none
`motionPlanner`	generates collision-free paths for planar mobile robots	none
`reactionForceCalculator`	accounts for reaction forces between multiple end-effectors handling the same object	none

Table 4.1: General-purpose simulator components and desciptions
Listed above are the general-purpose `evComponents`, their descriptions, and their task variables. The task variables are those that can optionally be included in a configuration's task parameters, allowing optimization of their values.

than an external procedure), since some modules may have special-purpose methods that must be used.

4.2 Computing Robot State

The most fundamental job of the simulator is to compute the motion of the robot. In Darwin2K, the `DEsolver` is a numerical differential equation solver responsible for computing the motion of the robot over a short time-step given the current position, velocity, and acceleration of each of the robot's degrees of freedom. After the `evaluator` initially tells the `DEsolver` which `controller` is to be used, the `DEsolver` queries the controller for a command, computes the robot's state derivative (velocity and acceleration), and integrates the state derivative (using a method such as Runge-Kutta 4) to determine the motion of the robot. The interface between the `DEsolver` and `controller` enables new controllers to be added without affecting the physical simulation process: the controller provides an acceleration or torque command for each of the robot's degrees of freedom, while the `DEsolver` knows nothing about the `controller`'s internals. The controller can also specify a maximum time step so that the `DEsolver` knows how frequently to query the `controller` for commands.

The `DEsolver` supports two modes of simulation: kinematic and dynamic. Kinematic simulation only considers positions, velocities, and accelerations; there is no inclusion of force, torque, mass, or angular inertia when computing the motion of the robot. In contrast, dynamic simulation is based on the forces and torques applied to the robot's links and computes the accelerations caused by those forces and torques. When kinematic simulation is being used, the `DEsolver` asks the `controller` for an acceleration command for each of the robot's degrees of freedom. This set of commands is the state acceleration vector $\ddot{\Theta}$, and is appended to the robot's state velocity vector ($\dot{\Theta}$) to form the robot's state derivative $\dot{S}$ which the solver uses to compute a new robot state S. S itself is composed of the position vector Θ and the velocity vector $\dot{\Theta}$: $S = [\Theta, \dot{\Theta}]$. When the `DE-solver` is performing dynamic simulation, the controller supplies a force/torque command (one force or torque for each degree of freedom). These torques are then limited based on maximum actuator force capabilities, and then the solver uses the robot's dynamic model (see Section 4.4.2) to compute $\ddot{\Theta}$ based on the applied forces and torques. As with kinematic simulation, this computed acceleration vector is concatenated to the robot's current velocity vector to form $\dot{S}$. Depending on the specific type of `DEsolver`, a number of state derivative calculations may be made when computing the robot's new state; the simplest solver is the `euler` solver, which updates the robot's state according to the formula

$$S(t + \Delta t) = S(t) + \dot{S}(t)\Delta t \tag{4.1}$$

where Δt is the time step being used. While this update formula is very straightforward

-- simply multiply the derivative by the time step and add the result to the current state -- it has extremely bad numerical properties and is really only useful as a demonstration of the basic process. Darwin2K includes a `DEsolver` implementing the Runge Kutta 4 algorithm (see e.g. [Press92]), called the `rungeKutta4`. This method is much more stable and is used in the experiments in Chapter 5.

Choosing the stepsize Δt is critical to simulation speed and stability, and the optimal size changes with the robot's state and the controller's commands. Because of this, the `DEsolver` uses an adaptive stepsize method called step doubling [Press92] to balance computational efficiency and numerical accuracy. Briefly, this method operates by estimating the error for the current stepsize by taking one large step of size Δt, recording the resulting state, and then comparing the state to that computed from two small steps of size $\Delta t/2$. The DEsolver then computes an error estimate from the two states, which is used to adjust the stepsize based on the error properties of the integration algorithm and the maximum allowable error (set by the designer). The error estimate used in the adaptive stepsize algorithm is defined as

$$error = \max(|P_{i,\Delta t} - P_{i,\Delta t/2}|) \tag{4.2}$$

where $P_{i,\Delta t}$ is the position of the i^{th} end effector of the robot after a step of Δt, and $P_{i,\Delta t/2}$ is the position of the i^{th} end effector after 2 steps of size $\Delta t/2$. This error, and a corresponding new step size, is computed after every two steps of $\Delta t/2$, so that the simulator is always using an appropriate step size. It is worth noting that this error estimate includes the effects of the controller's commands on the simulation stability; thus, if a robot is operating near a singularity, the DEsolver will automatically decrease the time step to yield stable simulation results. Conversely, if the controller's commands are not changing rapidly and the robot's motion is smooth, the DEsolver will choose a large stepsize to decrease simulation time.

4.3 Singularity-Robust Inverse Controller

Simulating a robot requires a means of controlling it. Robot synthesis presents difficult requirements for control algorithms: A typical synthesis run may involve simulating ten to one hundred thousand different robots, which means that the control algorithm must be applicable to a huge diversity of robot properties while being computationally efficient. The robots being simulated may have less than 6 degrees of freedom, or they may be redundant; they may have fixed or mobile bases, and single, multiple, or bifurcating manipulators; they may have revolute or prismatic joints. Automatically deriving inverse kinematics for arbitrary serial-chain manipulators (including branching manipulators and redundant manipulators) is not feasible, and analytic solutions are known for only a limited range of kinematic types. Deliberative planning is sufficiently flexible to allow use with a broad range of robot topologies, but such methods are too expensive given present-day computing resources and the large number of simulations required. Other methods are numerical, local approaches based on the robot's Jacobian.

These methods can deal with the range of robots required for synthesis and are computationally efficient. In addition, Jacobian controllers have a degree of robustness to errors in motion execution (due to actuator saturation or other causes) since they generate commands based on the robot's present state. The primary drawback of Jacobian controllers is that they are local in nature and thus have limited abilities to avoid workspace obstacles and satisfy other constraints (e.g. avoidance of joint limits or minimization of time or energy). Still, the flexibility and computational efficiency of Jacobian-based control schemes outweigh their limitations, and including some pose or trajectory variables in the synthesis process can provide a means of improving their performance. Thus, Jacobian-based controllers are an appropriate match for automated synthesis, at least until sufficient computing power is available to make deliberative planning practical.

Manipulator tasks are often specified as a path through space for an end-effector. To address this common scenario, Darwin2K includes a Jacobian-based controller that uses the Singularity-Robust Inverse (SRI) to map end-effector velocity commands to joint velocities for serial-chain and branching manipulators ([Nakamura86], [Nakamura91]). The `sriController` uses a single Jacobian matrix for an entire robot (including mobile base and multiple or branching manipulators, if any) and uses the SRI to calculate the required joint velocities. Darwin2K also provides a reasonably flexible end-effector path representation which generates end-effector position and velocity commands, though other path representations can be used with the Jacobian-based controller.

4.3.1 Calculating Jacobians

The `sriController` uses a robot's Jacobian to generate commands for all of the robot's degrees of freedom. The controller's first step is to query the robot for its complete Jacobian matrix, including mobile bases and multiple or branching serial chains. This is accomplished by searching through the robot mechanism to identify all serial chains and end effectors, computing a Jacobian for each chain, computing the base's Jacobian, and forming a single Jacobian from all of the chain and base Jacobians. Each row of blocks in the final Jacobian shows how one endpoint is affected by motions of each joint; one column of blocks in the Jacobian shows how one base or serial chain affects each endpoint. For example, the robot in Figure 4.1 consists of a free-flying 6 DOF base and two 7 DOF arms attached to the base by a common 3 DOF serial chain. Each of the 7 DOF arms has an end effector, so the robot's Jacobian is:

$$J = \begin{bmatrix} J_{e1,c1} & 0 & J_{e1,com} & J_{e1,base} \\ 0 & J_{e2,c2} & J_{e2,com} & J_{e2,base} \end{bmatrix} \tag{4.3}$$

where

$$J_{e1,c1} = \text{Jacobian for endpoint 1 w.r.t. arm 1,}$$

$$J_{e2,c2} = \text{Jacobian for endpoint 2 w.r.t. arm 2,}$$

$$J_{e1,com} = \text{Jacobian for endpoint 1 w.r.t. the common chain,}$$

$$J_{e2,com} = \text{Jacobian for endpoint 2 w.r.t. the common chain,}$$

$$J_{e1,base} = \text{Jacobian for endpoint 1 w.r.t. the base,}$$

$$J_{e2,base} = \text{Jacobian for endpoint 2 w.r.t. the base.}$$

Figure 4.2 gives a schematic view relating the Jacobian blocks to the robot's degrees of freedom and end effectors. The two zero blocks in the Jacobian indicate that the joints of one arm do not affect the motion of the other arm's endpoint. In this case, the Cartesian velocity vector dx would have 12 elements, 6 for each endpoint; the joint vector $d\theta$ would have 23 elements.The algorithm for identifying serial chains is straightforward. Each joint attached to the base is the root of a serial chain; each chain is grown until either an endpoint is reached or a branch is found (a branch occurs when 3 or more joints are attached to a single link). At branches, the parent chain ends and new chains are created for each of the joints that are not part of the parent chain. For serial chains, each column of a block in the Jacobian is calculated in the usual way (see e.g. [Craig89]), using the endpoint and serial chain corresponding to the block. Because of Darwin2K's object-oriented software approach, each type of base has both data and procedures associated with it. One of these procedures is the calculation of the base's Jacobian; this provides a standard interface be-

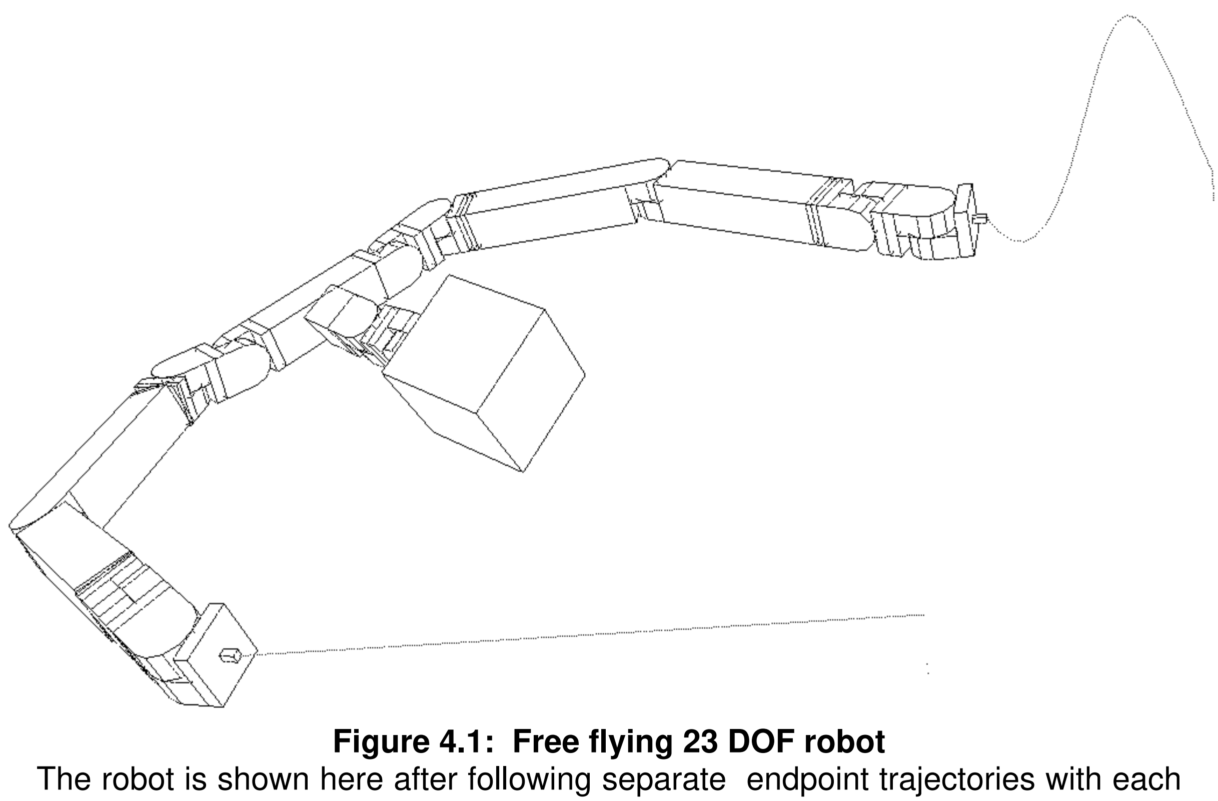


Figure 4.1: Free flying 23 DOF robot

The robot is shown here after following separate endpoint trajectories with each arm under control of the SRI controller.

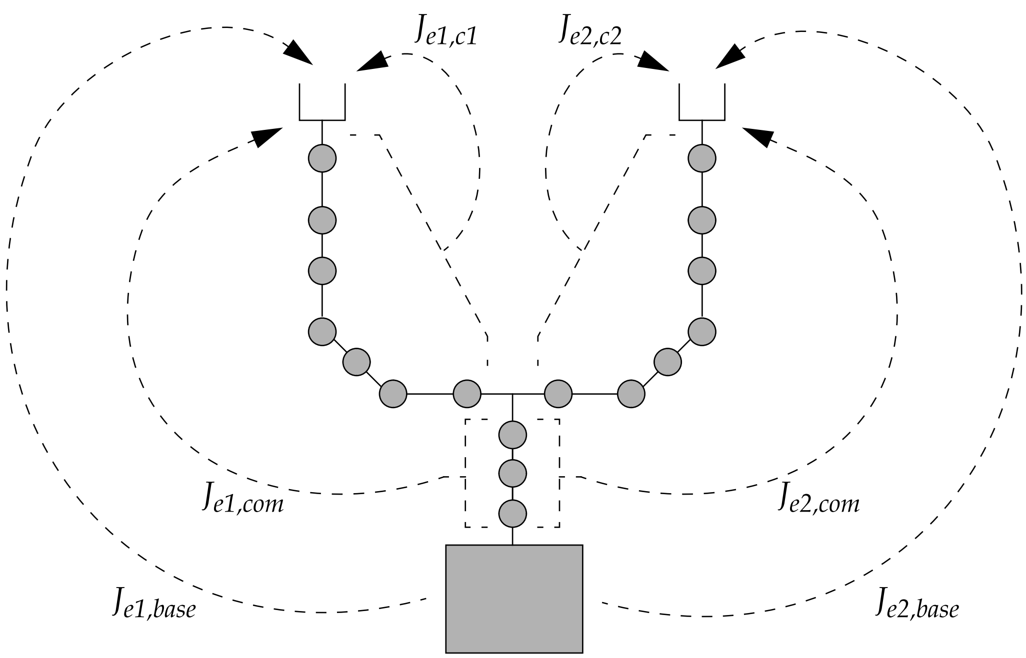


Figure 4.2: Schematic view of Jacobain sub-matrices
The dotted lines indicate how different blocks of the Jacobian in Equation 4.3 relate to the robot's degrees of freedom and end effectors. (The robot is shown in Figure 4.1.)

tween the Jacobian controller and all bases. Note that when using Jacobian control to command a robot with a non-holonomic base, some of the base's DOFs are not used: For example, the steering angle of an Ackerman base would be fixed and the corresponding column in the Jacobian would be a zero vector, since a differential change in the steering angle does not independently cause motion of the robot. The driving (forward-reverse) degree of freedom would still be used in this case, since it directly causes the robot to move.

4.3.2 The Singularity-Robust Inverse

The `sriController` uses a generalized inverse of a robot's Jacobian to compute the joint motions that will cause a desired end-effector motion. To allow stable motions to be computed near a robot's singularities, Darwin2K uses the Singularity-Robust Inverse [Nakamura86], rather than the pseudoinverse. Briefly, the Singularity-Robust Inverse (SRI) J^* of the Jacobian matrix J is identical to the pseudoinverse $J^\#$ (defined as $J^T(JJ^T)^{-1}$) except when the robot (and thus its Jacobian) approaches a singularity. Near singularities, the SRI assumes the form

$$J^* = J^T(JJ^T + \lambda I)^{-1} \tag{4.4}$$

where I is the identity matrix, and λ is a scalar. Roughly speaking, λ provides some damping when J becomes singular. λ is determined by the following equation [Kelmar90]:

$$\lambda = \lambda_o\left(1 - \frac{\omega_{k+1}}{\omega_k}\right) \tag{4.5}$$

where λ_o is a constant and ω_i is the measure of manipulability at time step *i*. It approaches zero as the robot nears a singular configuration; thus, by looking at how ω changes over time, proximity to a singular configuration can be detected. Note that when λ is zero, J^* and $J^\#$ (the pseudoinverse) are identical.

After calculating J^*, the joint motion $d\theta$ for the desired end-effector motion dx can be computed from the SRI by $d\theta = J^* dx$. Normally, the actual endpoint motion caused by $d\theta$ will be identical to dx; however, when using the SRI the motion will deviate slightly near a singularity due to the robot's inability to move along the singular direction. Note that the controller scales the $d\theta$ vector appropriately if any joint's velocity exceeds its actuator's capability; this prevents the trajectory from deviating spatially from the desired path, though it does cause temporal deviation.

Since some tasks do not require full spatial motion, the optimal robot may have less than 6 degrees of freedom for each end effector[2]. For example, many vehicles used in construction (such as an excavator or material handler) do not have 6 degrees of freedom, but it is still desirable to be able to control them. This presents a problem for the calculation of ω, which is normally computed as $\omega = \sqrt{det(J^T J)}$. If a robot has fewer than 6 degrees of freedom, ω will always be zero. A better way to compute ω would only consider the degrees of freedom that the robot has. We can do this by considering ω in a different way: it is the product of the *singular values* of J. The singular values of a matrix M are defined as the square-roots of the eigenvalues of M^TM. Since the product of the eigenvalues of M^TM is the determinant of M^TM, the product of the singular values of M is equal to $\sqrt{det(M^T M)}$. In light of this, we can redefine ω to be the product of the *non-zero* singular values of J. This allows us to use the SRI even when a mechanism is always singular. If a mechanism has less than six DOFs, one or more of its singular values will always be zero. By monitoring the product of the non-zero singular values, we can still determine when the mechanism is approaching a singular configuration.

The singular values can be computed by the Singular Value Decomposition (SVD) [Press92]:

$$M = U\Sigma V^T \tag{4.6}$$

where U and V are each orthogonal matrices and Σ is a diagonal matrix containing the singular values of M. In addition to the singular values, SVD provides several other useful pieces of information: the pseudoinverse $J^\#$ (equal to $V\Sigma^{-1}U^T$ when no singular values are 0); the minimum norm joint velocity vector $d\theta$ (via $J^\#$); and the nullspace of J (the columns of V corresponding to the zero singular values form an orthonormal basis to the nullspace). $d\theta$ is used directly to control the robot when not in the vicinity of a singularity;

2. Kinematically redundant robots (those with > 6 DOFs for each end effector) do not require any special modification to the SRI algorithm.

the nullspace can be used to optimize an objective function when the robot has some redundant degrees of freedom, as will be discussed in the following section.

The `DEsolver` uses acceleration commands when integrating the robot state, but the method outlined above computes a vector of differential joint motions $d\theta$. The `sriController` accounts for this by using the desired end effector velocity as dx, and then using a simple divided-differences formula to compute the acceleration command $\ddot{\theta}$ from the current velocity command $\dot{\theta}$ and current joint velocity vector $\dot{\theta}_{current}$:

$$\ddot{\theta} = \frac{(\dot{\theta} - \dot{\theta}_{current})}{\Delta t}. \tag{4.7}$$

4.3.3 Using redundant degrees of freedom

If a robot has any kinematic redundancy, then any differential joint motion in the nullspace of the Jacobian will not cause any endpoint motion. The `sriController` takes advantage of this fact by using nullspace motions to optimize a function of the robot's configuration without affecting the robot's trajectory. This nullspace optimization is a simple gradient descent which tries to minimize a scalar function of the joint angles. This is done by computing the gradient of the objective function and projecting this gradient onto the nullspace. This projected vector can be multiplied by a scalar and added to the least-squares solution $d\theta$.

One way to use this redundancy is the avoidance of joint limits [Liegeois77]. To encourage each joint to remain near the middle of its range of motion, we can use the following objective function defined over the joint vector:

$$F(\vec{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\theta_i - a_i}{\theta_{i,max} - a_i}\right) \tag{4.8}$$

$$a_i = \frac{\theta_{i,max} + \theta_{i,min}}{2} \tag{4.9}$$

$F(\vec{\theta})$ is at a minimum when all joints are in the middle of their ranges, so moving opposite the gradient direction will make the robot avoid its joint limits.

After computing the gradient of the function to be optimized, the gradient must be projected onto the nullspace of the Jacobian. The basis N of the nullspace can be found in the V matrix from the Singular Value Decomposition. The matrix which projects a joint vector onto the nullspace is thus:

$$P = N(N^TN)^{-1}N^T \tag{4.10}$$

Note that this projection can also be computed as

$$P = I - J^{\#}J \tag{4.11}$$

but the former is often more efficient considering the small size of the nullspace matrix relative to J. In either case, the final joint motion can be computed as

$$d\theta = J^* dx - \varepsilon P \nabla F \tag{4.12}$$

where ε is a scalar determining the size of the gradient descent step. In practice, this method does cause some motion of the end-effector due to the finite (as opposed to infinitesimal) simulation step size, which can make very precise motions difficult. To account for this, nullspace optimization is only performed when $|\nabla F|$ is greater than a threshold. This threshold and ε can both be included as task parameters, since their effects on precision are hard to predict in advance.

4.3.4 Trajectory representations

The `sriController` causes the endpoints of a robot to move with a specified velocity; Darwin2K provides two trajectory representations -- the `path` and `relativePath` --that generates velocity commands. The `path` generates commands to move the endpoints toward successive points along a path in a straight line, and includes parameters for linear and angular velocity and acceleration. The `path` also includes a number of properties for each waypoint: whether to stop at or move through the waypoint, distance and velocity thresholds (how close the end effector must approach, and how slowly it must be moving, before moving to the next waypoint), and a force and torque to be applied. The `relativePath` is derived from the `path` and is similar except that waypoints are defined relative to a link, payload, or other moving coordinate system. Other types of trajectories can be derived from the `genericPath` class, allowing them to be used with the `sriController`.

4.4 Robot Dynamics

Accounting for robot dynamics is necessary when synthesizing the non-kinematic properties of a robot such as actuator type and the structural geometry and inertial properties of links. Link sizing depends on the loads that must be supported by the link, including actuators, tool forces, and forces applied by other links. Actuator selection depends on velocity and acceleration requirements, and also on tool forces and static forces (e.g. gravity) due to link mass. Thus, actuator selection and link geometry cannot be meaningfully optimized without accounting for dynamics. As has been noted in [Paredis93], kinematic and dynamic design are not independent problems; determining satisfactory values for a robot's dynamic properties may require modifications to kinematic properties. Thus, it is best if kinematic and dynamic properties can be simultaneously optimized.

There are two distinct but related ways of accounting for dynamics: computing the forces required to cause a desired motion (also known as inverse dynamics), and computing the motion of the robot given a set of applied forces (forward dynamics, or dynamic simulation). When a controller supplies acceleration commands and is assumed to have accurate knowledge of the robot's dynamics and actuator capabilities, the robot's motion can be accurately predicted on the basis of kinematics alone and the required actuator forces can be computed from the robot's motion using inverse dynamics equations. A robot's actuator saturation can thus be measured, allowing it to be used as a metric that is minimized by the synthesizer.

While this approach is acceptable for some synthesis problems, there are some tasks for which dynamic simulation is necessary or useful. Robots for zero-gravity or microgravity environments may experience significant motion due to reaction and inertial forces; in these cases, the robot's motion cannot be known from a purely kinematic simulation, as the robot's base cannot be considered immobile and will move due to actuator and external forces. Dynamic simulation is also necessary when modeling the effects of imperfect controllers, controllers that supply torque rather than acceleration commands, and controllers that do not use a dynamic model of the robot. For example, some tasks require that actuators are operated at maximum torque during substantial portions of the task in order to move as quickly as possible. Earthmoving machines are often controlled in joint-space when moving a payload between two positions. In this case, task execution time is limited by the acceleration and velocity of each degree of freedom, which in turn is limited by available actuator force. Since the actuators are frequently operated in saturation and since task completion time depends heavily on maximum actuator force, it is not sufficient to simply measure actuator saturation. The effects of limited actuator torque must be accounted for when determining the machine's motion since the controller cannot accurately predict how the robot will move. Finally, dynamic simulation can be useful for tasks where accurate trajectory following is required only at some points, while during most of the task some actuator saturation is acceptable. In these cases, it can be desirable to know how actuator saturation affects power consumption, collisions due to unexpected motions, execution time, and other metrics. Both the forward and inverse dynamic methods are based on the iterative Newton-Euler formulation of dynamics for serial chain manipulators, with modifications to allow multiple and branching manipulators and base dynamics.

4.4.1 Computing torques for a desired motion

When a task requires that a robot manipulator follow a trajectory closely, the robot's actuators must be sized to provide adequate torque during execution of the trajectory. To ensure this, we can measure the torque required at each joint to cause the motions necessary to follow the trajectory. If the torque at any joint exceeds the joint's torque limits, the robot will deviate from the trajectory. Recording how much a robot's actuators are saturated during a task allows the synthesizer to select designs with adequately-sized actuators.

The basic algorithm for computing the joint torques required to move each joint of a serial chain at a desired velocity and acceleration is described in [Craig 89]; the equations are replicated here in Figures 4.3 and 4.4. Briefly, joint velocities and accelerations are propagated from the base outward to the tip of each serial chain, computing link velocities, accelerations, and inertial forces and moments for each link along the way. The

Outward iterations:

joint *i+1* revolute:

$$\omega_{i+1} = \omega_i + \dot{\theta}_{i+1}\hat{Z}_{i+1}$$
$$\underline{\dot{\omega}}_{i+1} = \underline{\dot{\omega}}_i + \omega_i \times \dot{\theta}_{i+1}\hat{Z}_{i+1} + \underline{\ddot{\theta}}_{i+1}\hat{Z}_{i+1}$$
$$\underline{\dot{v}}_{i+1} = \underline{\dot{\omega}}_i \times P_{i+1} + \omega_i \times (\omega_i \times P_{i+1}) + \underline{\dot{v}}_i$$

joint *i+1* prismatic:

$$\omega_{i+1} = \omega_i$$
$$\underline{\dot{\omega}}_{i+1} = \underline{\dot{\omega}}_i$$
$$\underline{\dot{v}}_{i+1} = \underline{\dot{\omega}}_i \times P_{i+1} + \omega_i \times (\omega_i \times P_{i+1}) + \underline{\dot{v}}_i + 2(\omega_{i+1} \times (\dot{d}_{i+1}\hat{Z}_{i+1})) + \underline{\ddot{d}}_{i+1}\hat{Z}_{i+1}$$

$$\underline{\dot{v}}_{Ci+1} = \underline{\dot{\omega}}_{i+1} \times P_{Ci+1} + \omega_{i+1} \times (\omega_{i+1} \times P_{Ci+1}) + \underline{\dot{v}}_{i+1}$$
$$\underline{F}_{i+1} = m_{i+1}\underline{\dot{v}}_{Ci+1}$$
$$\underline{N}_{i+1} = I_{i+1}\underline{\dot{\omega}}_{i+1} + \omega_{i+1} \times I_{i+1}\omega_{i+1}$$

Inward iterations:

$$\underline{f}_i = \underline{f}_{i+1} + \underline{F}_i$$
$$\underline{n}_i = \underline{N}_i + \underline{n}_{i+1} + P_{Ci} \times \underline{F}_i + P_{i+1} \times \underline{f}_{i+1}$$
$$\underline{\tau}_i = \underline{n}_i \cdot \hat{Z}_i + e_i\underline{\ddot{\theta}}_i \qquad \text{(revolute joint)}$$
$$\underline{\tau}_i = \underline{f}_i \cdot \hat{Z}_i + e_i\underline{\ddot{d}}_i \qquad \text{(prismatic joint)}$$

θ_i - angle of revolute joint *i*

d_i - length of prismatic joint *i*

τ_i - torque to apply at joint *i*

$\hat{Z}_i$ - axis of joint *i* (unit length)

m_{i+1} - mass of link *i*+1

ω_{i+1} - angular velocity of link *i*+1

$\dot{v}_{i+1}$ - acceleration at base of link *i*+1

$\dot{v}_{Ci+1}$ - acceleration of link *i*+1's c.o.m.

F_{i+1} - inertial force for link *i*+1

N_{i+1}- inertial moment for link *i*+1

f_i - force applied by link *i* to link *i*+1

n_{i+1}- moment applied by link *i* to link *i*+1

I_{i+1} - world-space inertia tensor of link *i*+1

P_{Ci+1} - vector from joint *i* to link *i*+1's c.o.m.

P_{i+1} - vector from joint *i*-1 to joint *i*

e_i - effective inertia of actuator *i*

Figure 4.3: Iterative Newton-Euler equations for serial chain dynamics. Underlined terms in the equations above denote S-vals (see Section 4.4.2), and all quantities are in world coordinates (rather than link coordinates). f_i and n_i for the final link in a serial chain are computed from tool forces (for terminal chains) or from the sum of forces and moments applied to child serial chains (for chains supporting other branches). v_i, ω_i, and for the first link of each child chains are computed from the distal links of the parent chain. See Figure 4.4 for a graphical depiction of vector quantities.

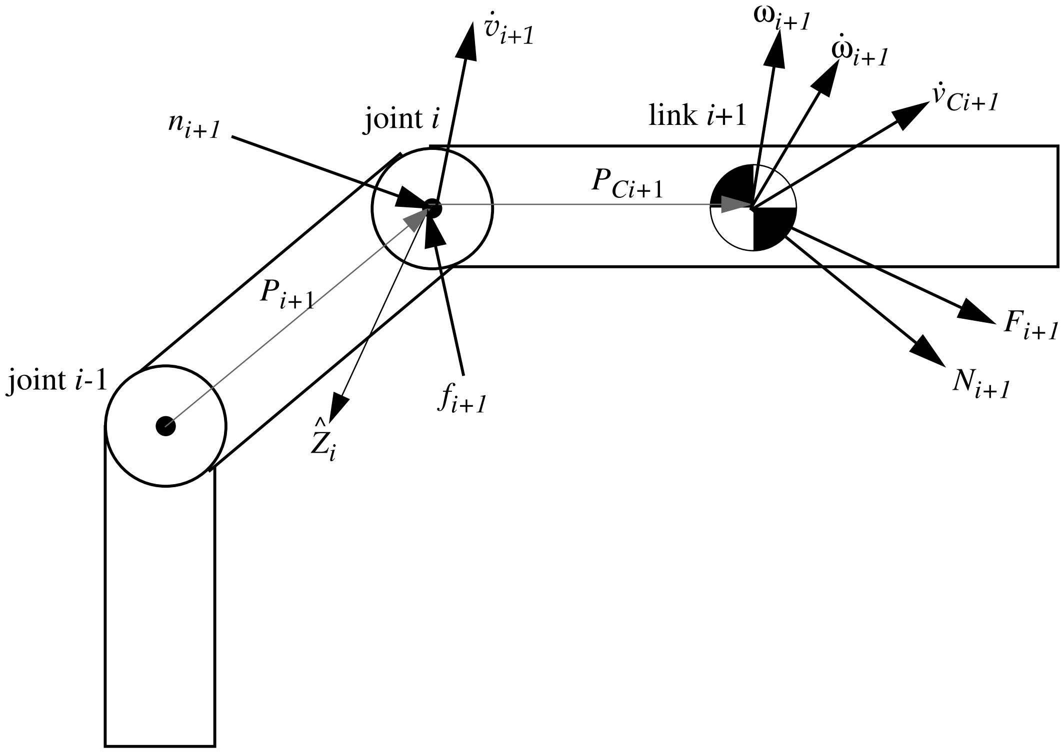


Figure 4.4: Definition of vector quantities for Newton-Euler equations
This figure shows the quantities relating to link *i*+1 for the iterative Newton-Euler equations listed in Figure 4.3.

forces acting on each link are then accumulated, moving inward back to the base. This method can be used with multiple serial chains (including those in branching manipulators) by treating each serial chain in isolation. When two or more serial chains are attached to the end of a parent chain, the linear acceleration and angular velocity and acceleration of the distal link of the parent chain are used to set the acceleration and velocity of the base links of the child chains. After propagating the velocities and accelerations down the child chains (including their children, if any), the forces are propagated back to the base of each child. These reaction forces are then applied to the distal link of the parent chain, as if they were tool forces. This method can be used with a kinematic simulation to compute the necessary joint torques for arbitrary joint accelerations and velocities at each time step of the simulation. The `actuatorSaturation` metric (see Section 4.8.2) can then be used to compare the required joint torques to each actuator's peak and continuous ratings to determine if the actuators can provide adequate torque during the task. This enables the synthesizer to synthesize an appropriate selection of actuators based on the task's requirements.

4.4.2 Computing motion from force and torque

Dynamic simulation computes the motion of the robot due to applied forces and is useful when the motion of the robot cannot be predicted solely on the basis of a controller's commands. This occurs when the robot's controller is providing torque commands, when significant actuator saturation is expected, when the controller is not commanding all of the robot's degrees of freedom, or when the controller does not have complete and accurate information about the robot's dynamic model. In these cases, dynamic simulation must be used to compute the motion of the robot. The goal of dynamic simulation is to compute the state derivative $\dot{S} = [\dot{\Theta}, \ddot{\Theta}]$ of a robot's DOFs given the current state $S = [\Theta, \dot{\Theta}]$ of the robot and a set of torques to be applied to each DOF. The robot's state derivative is then integrated to compute the robot's new state. The first step is to solve the robot's dynamic equations for the acceleration vector. (The velocity $\dot{\Theta}$ is already known.) The general form of the dynamic equation of an n-DOF robot consisting of a generalized base and one or more (possibly branching) serial chains is:

$$T = M(\Theta)\ddot{\Theta} + V(\Theta, \dot{\Theta}) \tag{4.13}$$

Where T is an n-vector of torques, M is an n x n mass matrix, and V is an n-vector that includes centrifugal, coriolis, gravity, and friction terms. Solving for $\ddot{\Theta}$ yields:

$$\ddot{\Theta} = M^{-1}(T - V) \tag{4.14}$$

When deriving the dynamic equation for a manipulator symbolically, one can apply the Newton-Euler equations to create an equation for each joint torque (i.e. an element of T). The terms in each equation can then be grouped into quantities that multiply a joint acceleration (elements of M), and quantities that do not multiply a joint acceleration (elements of V). Each element of M and V can then be numerically evaluated and, along with the applied torque vector T, can be used to solve for $\ddot{\Theta}$.

A full symbolic representation for the dynamic equations is not necessary: the only variables we need to solve for are the joint accelerations $\ddot{\theta}_i$ (i.e. the elements of $\ddot{\Theta}$) and we know from the structure of Equation 4.14 that joint accelerations will never be multiplied with each other or appear as arguments to any function. Thus, we simply need to keep each $\ddot{\theta}_i$ separate, and record the sum of the terms that do not contain any joint accelerations. Darwin2K implements this approach by separately accumulating the coefficients of each $\ddot{\theta}_i$ as the Newton-Euler equations are evaluated numerically. For example, if the expression $2\ddot{\theta}\sin\theta + l\cos\dot{\theta}$ is evaluated for

$$\dot{\theta} = 0 \qquad \theta = \frac{\pi}{2} \qquad l = 1$$

giving $2\ddot{\theta} + 1$, then the result would be the vector [1 2] since the coefficient of $\ddot{\theta}$ is 2 and the constant coefficient is 1. This vector is called an *s-val* (short for "separated value"), since each joint acceleration's coefficient (only one in this case) is recorded separately. For a robot with n degrees of freedom, s-vals will have n+1 entries (one for each DOF acceleration and one for all other terms). Thus, the expression

$$m_0 + m_1\ddot{\theta}_1 + m_2\ddot{\theta}_2 + \ldots + m_n\ddot{\theta}_n \tag{4.15}$$

is represented by the s-val $\begin{bmatrix} m_0 & m_1 & m_2 & \dots & m_n \end{bmatrix}$. Computing a scalar numeric value for an s-val based on numeric values for $\ddot{\theta}_i$'s is accomplished by taking the dot product of the s-val with an augmented vector of accelerations:

$$value = \begin{bmatrix} 1 & \ddot{\theta}_1 & \ddot{\theta}_2 & \dots & \ddot{\theta}_n \end{bmatrix}^T \begin{bmatrix} m_0 & m_1 & m_2 & \dots & m_n \end{bmatrix} \tag{4.16}$$

Note that when evaluating equations using s-vals, all terms that do not contain any joint accelerations get lumped into the constant term (e.g. m_0 in Equation 4.16).

When the entire set of torque equations is evaluated using s-vals, the result is a set of vectors (s-vals), with one s-val for each element of T. Each s-val directly represents the row of M and element of V for the corresponding element of T. After evaluating the i^{th} DOF's torque equation, the entries of the s-val for T_i are copied into the appropriate row of M and V. If T_i is the s-val $\begin{bmatrix} m_0 & m_1 & m_2 & \dots & m_n \end{bmatrix}$, then the i^{th} element of V would be m_0, and $[m_1 \dots m_n]$ would compose the i^{th} row of M. After evaluating all torque equations, M and V can be used in two ways: to compute the joint torques required for a desired set of accelerations (using Equation 4.13 above), or to compute the accelerations caused by a given set of joint torques (Equation 4.14).

Evaluating the torque equations requires arithmetic operators that work on scalars and vectors composed of s-vals. The s-vals in the Newton-Euler equations in Figure 4.3 are underlined; the operators needed are those having one or more s-val arguments. In particular, note that no two s-vals are ever multiplied with each other. This reflects the fact that the dynamic equations do not contain any product of joint accelerations.

The behavior of the s-val operators can be deduced given the s-val representation and the distributive property. After representing each s-val as the sum shown in Equation 4.15, we can apply the normal operator symbolically and then regroup the acceleration terms to derive the symbolic form of the result. For example, the addition operator for two s-vals behaves as follows:

$$\begin{aligned}
s &= \begin{bmatrix} b_0 & b_1 & b_2 & \dots & b_n \end{bmatrix} = b_0 + b_1\ddot{\theta}_1 + b_2\ddot{\theta}_2 + \dots + b_n\ddot{\theta}_n \\
t &= \begin{bmatrix} d_0 & d_1 & d_2 & \dots & d_n \end{bmatrix} = d_0 + d_1\ddot{\theta}_1 + d_2\ddot{\theta}_2 + \dots + d_n\ddot{\theta}_n \\
s + t &= b_0 + b_1\ddot{\theta}_1 + b_2\ddot{\theta}_2 + \dots + b_n\ddot{\theta}_n + d_0 + d_1\ddot{\theta}_1 + d_2\ddot{\theta}_2 + \dots + d_n\ddot{\theta}_n \\
&= (b_0 + d_0) + (b_1 + d_1)\ddot{\theta}_1 + (b_2 + d_2)\ddot{\theta}_2 + \dots + (b_n + d_n)\ddot{\theta}_n \\
&= \begin{bmatrix} (b_0 + d_0) & (b_1 + d_1) & (b_2 + d_2) & \dots & (b_n + d_n) \end{bmatrix}
\end{aligned} \tag{4.17}$$

In this case, the sum of the s-vals is simply the sum of their two vector representations. (Keep in mind that the two s-val arguments and the result ultimately represent scalar values, when evaluated numerically in the context of a set of given joint accelerations.) A more complex operator is the vector cross-product, which operates on one normal vector and one s-val vector:

$$s = \begin{bmatrix} (X = \begin{bmatrix} a_0 & \dots & a_n \end{bmatrix}) \\ (Y = \begin{bmatrix} b_0 & \dots & b_n \end{bmatrix}) \\ (Z = \begin{bmatrix} c_0 & \dots & c_n \end{bmatrix}) \end{bmatrix} \qquad v = \begin{bmatrix} x & y & z \end{bmatrix}^T \tag{4.18}$$

$$s \times v = \begin{bmatrix} Yz - Zy \\ Zx - Xz \\ Xy - Yx \end{bmatrix}$$

Since the cross-product operator is a linear operator and s is a sum, we can treat each element of the resulting s-val as the cross-product of the corresponding element of s with v:

$$s \times v = \begin{bmatrix} \begin{bmatrix} b_0 z - c_0 y & \dots & b_n z - c_n y \end{bmatrix} \\ \begin{bmatrix} c_0 x - a_0 z & \dots & c_n x - a_n z \end{bmatrix} \\ \begin{bmatrix} a_0 y - b_0 x & \dots & a_n y - b_n x \end{bmatrix} \end{bmatrix} \tag{4.19}$$

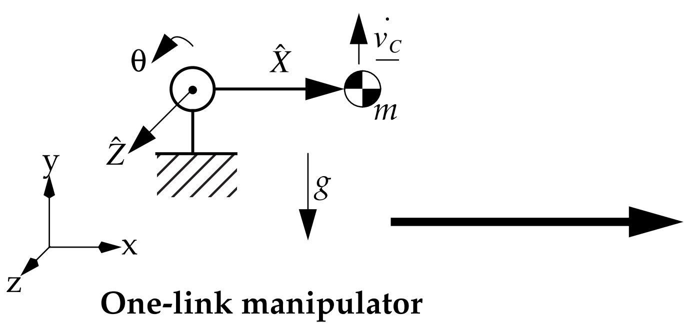
The behavior of the other operators can be derived in a similar fashion. The s-val scalar and vector types are implemented as C++ classes, and the operators are used via operator overloading. However, instead of producing a numeric value for each operation, an expression tree is built once and then evaluated numerically during each time step of the simulation.

To build the expression tree, the Newton-Euler equations are evaluated using s-val arithmetic. The tree is constructed once during an initialization phase and then evaluated numerically at each simulation step. During tree construction, each application of an operator to an s-val results in the construction of an operator node, which maintains references to its arguments. When a numerical value is requested from an operator, the operator queries its arguments (children) for their values, and then performs the necessary computation for its own value. Caching is used so that operators do not re-compute their values unless the values of their arguments have changed; this allows efficient reuse of intermediate results. The numerical evaluation process of a single torque equation is a depth-first traversal of the expression tree rooted at the torque s-val, and the result is a row from M and V (i.e. the value of the torque s-val). This is repeated for each torque s-val, thus filling in all of M and V.

Figure 4.5 shows the expression tree built by this process for a one-link manipulator. During initialization, the Newton-Euler equations for the robot are symbolically evaluated using s-vals. Each s-val arithmetic operator creates a node in the tree; the children of the node are the arguments to the operator. Each s-val on the left-hand side of an equation is essentially a reference to one of the operators in the equation; the s-val's value is the value returned by the operator. The other variables in the tree are references to scalar or vector numeric variables; their values are set (based on the robot's state) before evaluating the tree. Figure 4.6 shows how the tree is evaluated numerically. First, the numerical variables are set, and $\ddot{\theta}$ is set to (0 1) (since $\ddot{\theta} = 0 + 1\ddot{\theta}$). Next, the numeric value of τ is computed. In the figure the dynamic equations are evaluated in the normal order, though in the implementation this would be triggered by the depth-first traversal: the s-val for τ is evaluated, which causes its arguments to be evaluated before being used; this evaluation ripples down the tree, and then the values propagate back up the tree. The values of M and V (both 1x1 for the one-DOF robot) are then extracted from τ: these comprise the robot's dynamic model for the current state. The two boxed equations in the lower right of Figure 4.6 can be used to compute the torque required for a given acceleration, and the acceleration caused by a given torque. The two equations make sense: a 2kg mass at the

Figure 4.5: Building and evaluating S-val dynamic equations for a one-link manipulator with point mass.

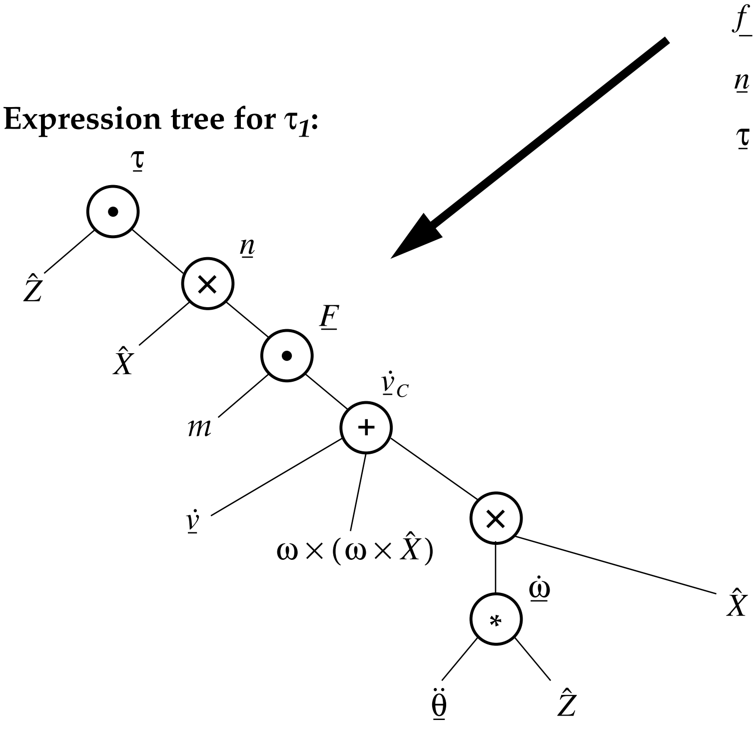
Underlined variables are s-vals, and are shown next to the operators that are assigned to them. Other variables are numeric variables; their values are set based on the robot's state each time the tree is evaluated numerically. Note that * denotes scalar multiplication in the expression tree.



Newton-Euler equations for robot:

$$\omega = \dot{\theta}\hat{Z}$$
$$\dot{\underline{\omega}} = \ddot{\underline{\theta}}\hat{Z}$$
$$\dot{\underline{v}} = -g$$
$$\dot{\underline{v}}_C = \dot{\underline{\omega}} \times \hat{X} + \omega \times (\omega \times \hat{X}) + \dot{\underline{v}}$$
$$\underline{F} = m\dot{\underline{v}}_C$$
$$\underline{f} = \underline{F}$$
$$\underline{n} = \hat{X} \times \underline{F}$$
$$\underline{\tau} = \underline{n} \cdot \hat{Z}$$

Expression tree for τ_1:



end of a 1m lever requires (2kg x 1m x 9.8m/s^2) = 19.6Nm of torque to remain at rest under 1g of gravity, and requires (2kg x 1m x 1m x $\ddot{\theta}$rad/s^2) = $2\ddot{\theta}$Nm of torque to accelerate

Figure 4.6: Numeric evaluation of the s-vals for the one-link robot.
The numeric variables are set before evaluating the s-val for τ. τ's value contains M and V (both 1x1 in this case), which can be used for computing torques and accelerations.

Numeric variables:

$$m = 2\text{kg} \qquad g = \begin{bmatrix}0 & -9.8 & 0\end{bmatrix}^T \qquad \hat{X} = \begin{bmatrix}1 & 0 & 0\end{bmatrix}^T \qquad \hat{Z} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^T \qquad \dot{\theta} = 1\text{rad/s}$$

$$\omega = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^T \qquad \omega \times (\omega \times \hat{X}) = \begin{bmatrix}-1 & 0 & 0\end{bmatrix}^T$$

S-val variables:

$$\underline{\ddot{\theta}} = (\ 0\ 1\ )$$

$$\underline{\dot{\omega}} = \underline{\ddot{\theta}} Z = \begin{bmatrix}(\ 0\ 0\ ) & (\ 0\ 0\ ) & (\ 0\ 1\ )\end{bmatrix}^T$$

$$\underline{\dot{v}} = -g = \begin{bmatrix}(\ 0\ 0\ ) & (\ 9.8\ 0\ ) & (\ 0\ 0\ )\end{bmatrix}^T$$

$$\underline{\dot{v}}_C = \underline{\dot{\omega}} \times \hat{X} + \omega \times (\omega \times \hat{X}) + \underline{\dot{v}} = \begin{bmatrix}(\ 0\ 0\ ) & (\ 0\ 0\ ) & (\ 0\ 1\ )\end{bmatrix}^T \times \begin{bmatrix}1 & 0 & 0\end{bmatrix}^T + \begin{bmatrix}-1 & 0 & 0\end{bmatrix}^T + \begin{bmatrix}(\ 0\ 0\ ) & (\ 9.8\ 0\ ) & (\ 0\ 0\ )\end{bmatrix}^T$$

$$= \begin{bmatrix}(\ 0\ 0\ ) & (\ 0\ 1\ ) & (\ 0\ 0\ )\end{bmatrix}^T$$

$$= \begin{bmatrix}(\ -1\ 0\ ) & (\ 9.8\ 1\ ) & (\ 0\ 0\ )\end{bmatrix}^T$$

$$\underline{f} = \underline{F} = m\underline{\dot{v}}_C = \begin{bmatrix}(\ -2\ 0\ ) & (\ 19.6\ 2\ ) & (\ 0\ 0\ )\end{bmatrix}^T$$

$$\underline{n} = \hat{X} \times \underline{F} = \begin{bmatrix}1 & 0 & 0\end{bmatrix}^T \times \begin{bmatrix}(\ -2\ 0\ ) & (\ 19.6\ 2\ ) & (\ 0\ 0\ )\end{bmatrix}^T = \begin{bmatrix}(\ 0\ 0\ ) & (\ 0\ 0\ ) & (\ 19.6\ 2\ )\end{bmatrix}^T$$

$$\underline{\tau} = \underline{n} \cdot \hat{Z} = \begin{bmatrix}(\ 0\ 0\ ) & (\ 0\ 0\ ) & (\ 19.6\ 2\ )\end{bmatrix} \begin{bmatrix}0 & 0 & 1\end{bmatrix}^T$$

$$= (\ 19.6\ \ 2\ )$$

2: coefficient of $\underline{\ddot{\theta}}$

19.6: constant coefficient

$$\longrightarrow \quad M = \begin{bmatrix}2\end{bmatrix} \quad V = \begin{bmatrix}19.6\end{bmatrix} \quad \longrightarrow \quad \boxed{\begin{aligned} T &= 2\ddot{\theta} + 19.6 \\ \ddot{\Theta} &= \frac{1}{2}(\tau - 19.6) \end{aligned}}$$

at $\ddot{\theta}$ rad/s.

This method handles multiple serial chains (independent, or from bifurcated manipulators) in the same way as the method in Section 4.4.1: velocities and accelerations are propagated from the distal links of a parent chain to the base links of the child chains, and reaction forces from the children's base links are added to the applied forces of the parent link. The robot state vector includes the joint positions and velocities for the DOFs of all serial chains, and each s-val contains an element for every DOF, not just the DOFs in a single serial chain. Mobile bases can be incorporated similarly: the state vector is augmented to contain the base's state (the position, orientation, and linear and angular velocities of the robots' base link), and the base module constructs dynamic equations for the base link using s-vals; in essence, there is simply a 6-DOF joint between ground and the robot's base. The acceleration and velocity of each serial chain rooted on the base is set from the base link's acceleration and velocity (taking into account the attachment location of each chain), and the reaction forces from each chain are applied to base link. Using the variables defined in Table 4.2, the base dynamic equations are:

$$\underline{F_c} = \sum_{i=1}^{numChains} \underline{f_{chain_i}} \tag{4.20}$$

$$N_c = \sum_{i=1}^{numChains} \underline{n_{chain_i}} + (P_{chain_i} - P_{COM}) \times \underline{f_{chain_i}} \tag{4.21}$$

Variable	Definition
$\underline{n_{chain_i}}$	reaction moment from the ith serial chain attached to the base
$\underline{f_{chain_i}}$	reaction force from the ith serial chain attached to the base
P_{chain_i}	world-space location of the first joint of the ith serial chain attached to the base
P_{COM}	world-space location of the base link's center of mass
$\underline{a_{frame}}$	acceleration (including gravity and inertial acceleration) of the base's reference frame
$\underline{\ddot{x}}$	acceleration of the base's center of mass relative to the base reference frame
$\underline{\dot{\omega}_{frame}}$	angular acceleration of the base's reference frame
$\underline{\dot{\omega}}$	angular acceleration of the base link relative to the base reference frame
ω_b	total angular velocity of the base link
$\underline{\dot{\omega}_b}$	total angular acceleration of the base link
I_b	world-space moment of inertia of the base link
m_b	mass of the base link

Table 4.2: Variable definitions for base dynamics equations
Underlined variables are s-vals. All variables are vectors except for I_b, which is a 3x3 matrix, and m_b, which is a scalar.

$$\underline{\dot{v}_b} = \underline{\ddot{x}} + \underline{a_{frame}} \tag{4.22}$$

$$(\underline{\dot{\omega}_b} = \underline{\dot{\omega}} + \underline{\dot{\omega}_{frame}}) \tag{4.23}$$

$$\underline{n} = I_b\underline{\dot{\omega}_b} + (\omega_b \times I_b\omega_b) + \underline{N_c} \tag{4.24}$$

$$\underline{f} = m_b\underline{\dot{v}_b} + \underline{F_c} \tag{4.25}$$

The elements of $\underline{n}$ and $\underline{f}$ (Equations 4.24 and 4.25, respectively) are added to the system of dynamic equations for the robot's serial chains, and the s-vals are augmented with six extra numbers (three each for $\underline{\ddot{x}}$ and $\underline{\dot{\omega}}$). If the base has any actuators (e.g. cold gas jets for a free-flying space robot), then the actuator forces can be applied via $\underline{n}$ and $\underline{f}$. If the base can only move in the plane, then the base module can lock the degrees of freedom for $\underline{\ddot{x}}$, $\underline{\dot{\omega}_x}$, and $\underline{\dot{\omega}_y}$ just as joint DOFs are locked (see Section 4.4.3 below). Alternatively, the out-of-plane reaction force and moments can then be calculated based on the robot's accelerations using the inverse dynamic model, with the out-of-plane accelerations set to zero. The normal force and coefficient of friction between the locomotion system and the ground can also be used to limit the tractive forces.

4.4.3 Computational considerations

As might be expected, dynamic simulation is slower than kinematic simulation due to the added burden of computing the robot's dynamic model and using it to compute the robot's acceleration. For example, dynamic simulation of a 6-DOF manipulator requires roughly twice the computation time of a kinematic simulation, and the disparity grows as the number of degrees of freedom increases. Fortunately, the optimizer tells the evaluator which metrics are currently being optimized so the evaluator can forego expensive evaluations (such as dynamic simulation) if no metrics require it.

Several improvements to the basic algorithm decrease computational complexity. In the expression trees for larger robots (especially robots with mobile bases or multiple serial chains), many s-vals will depend on only a few joint accelerations. We can take advantage of this by recording which entries are used as the tree is built, and only performing calculations on them. Each operator keeps an array of dependency flags, one for each joint and the constant element. An element of the operators's array is 1 if the corresponding element of any child's array is 1, and 0 otherwise. This eliminates many non-useful computations (i.e. multiplying by zero), thus speeding execution time. A further implementation advantage of the expression tree is that it is a fixed data structure and does not require intermediate results to be repeatedly allocated and deallocated during every evaluation; this reduces memory fragmentation and eliminates time spent allocating and initializing temporary variables. Finally, if any degrees of freedom will remain fixed over a period of time, the corresponding rows and columns in M, V, and T can be eliminated to reduce the cost of solving the system of equations. When large portions of a robot (such as one manipulator of a multi-manipulator robot) are not being moved, this can significantly reduce computational complexity and improve the accuracy and stability, which in turn allows a larger time step to be used. The drawback to this approach is that the forces and torques for the locked degrees of freedom are not computed; if power or torque estimates for these DOFs are desired, then the inverse dynamic model must also be used.

4.4.4 A controller for free-flyers that accounts for dynamics

When simulating a free-flying robot with one or more manipulators, the acceleration of the robot's base may not be known ahead of time. For example, in a space application it may be desirable to minimize use of the base's thrusters, so during manipulation the thrusters would not be used to counter reaction forces applied by the robot's manipulator. This presents a problem for the `sriController`: the desired accelerations for the joints are known (they can be computed based on the desired end-effector acceleration), and the force to be applied by the base's actuators is known (it is zero), but the joint torques and base accelerations are not known. In contrast, when the robot's base is fixed (or is being actuated), the accelerations for all DOFs are known and the dynamic model can be used to compute torque commands for all actuators. When the base is allowed to float, the controller knows some accelerations and some forces, and needs to compute the remaining accelerations and forces. The `ffController` solves this problem by breaking up the system of dynamic equations, rather than solving them simultaneously. Instead of the system

$$T = M\ddot{\Theta} + V, \tag{4.26}$$

the `ffController` uses the system

$$\begin{bmatrix} T_b \\ T_c \end{bmatrix} = \begin{bmatrix} M_{bb} & M_{bc} \\ M_{cb} & M_{cc} \end{bmatrix} \begin{bmatrix} \ddot{\Theta}_b \\ \ddot{\Theta}_c \end{bmatrix} + \begin{bmatrix} V_b \\ V_c \end{bmatrix} \tag{4.27}$$

where T_b is a 6-vector containing the force and torque applied to the base (zero in this case), T_c is a vector of n joint torques (n is the total number of DOFs in the robot's serial chains), $\ddot{\Theta}_b$ is a 6-vector containing the linear and angular acceleration of the base, and $\ddot{\Theta}_c$ is an n-vector of joint accelerations. The block matrices M_{bb}, M_{bc}, M_{cb}, and M_{cc} are extracted from M and have sizes 6 x 6, 6 x n, n x 6, and n x n, respectively. Similarly, V_b and V_c are extracted from V and have lengths 6 and n, respectively. As mentioned above, the controller knows T_b (zero since the base's actuators are not being used) and $\ddot{\Theta}_c$, which is computed from the desired end-effector accelerations using the same method as the `sriController`. Since M_{bb}, M_{bc}, V_b, and T_b are known, we can start by solving the first row of Equation 4.27 for $\ddot{\Theta}_b$:

$$T_b = M_{bb}\ddot{\Theta}_b + M_{bc}\ddot{\Theta}_c + V_b \tag{4.28}$$

$$\ddot{\Theta}_b = -M_{bb}^{-1}(M_{bc}\ddot{\Theta}_c + V_b) \tag{4.29}$$

The only remaining unknown is T_c, which can be computed directly from the second row of Equation 4.27:

$$T_c = M_{cb}\ddot{\Theta}_b + M_{cc}\ddot{\Theta}_c + V_c \tag{4.30}$$

Finally, the torque vectors T_b (which is $[0\ 0\ 0\ 0\ 0\ 0]^T$) and T_c are concatenated into a $(6+n)$-vector and given to the `DEsolver`, which first limits any joint torques that exceed actuator capabilities and then uses the dynamic model to compute the actual accelerations. (If no joint torques are beyond their respective actuators' capabilities, then the acceleration commands generated by the `ffController` are used directly, since plugging the commanded torques into the dynamic equations would yield the same accelerations.)

4.4.5 Summary

This section presented a method for automatically deriving a symbolic dynamic model for robots, which can then be used to numerically evaluate both forward and inverse dynamics. Significantly, this method allows dynamic simulation to be used for automated synthesis of robots; previous systems for automated synthesis were restricted to kinematic simulation during the synthesis process. This section also described the `ffController`, which uses the dynamic model to control free-flying robots.

4.5 Link Deflection

Optimal link sizing and actuator selection are interdependent: stiffer links are heavier than lighter ones, thus requiring more powerful actuators for a desired level of performance. But larger actuators add more mass, thus requiring greater stiffness. To optimize actuator selection and total robot mass, the stiffness of a robot's links must be accounted for. If the cross-sections of a robot's links are fixed during the synthesis process, then one of two undesirable outcomes is likely: some links will be oversized (for example, the distal links of a manipulator may not need to be as stiff as proximal links), thus increasing system mass and actuator requirements; or some links will be undersized, leading to unacceptable deflections or link failure in extreme cases. By estimating link deflection at each time step of simulation, Darwin2K can generate robot designs with appropriately-sized links and actuators.

The approach taken in this work is to compute the deflection of each endpoint of the robot by combining deflection estimates from the robot's links. Some important simplifying assumptions are made to facilitate fast and general computation:

- each link of the robot is modeled as a set of segments, each having uniform cross-section and material;
- deflections are computed only for links in the interior of serial chains (Figure 4.7);
- link deflection reaches unacceptable levels before the maximum strength of a link's material is reached, and before buckling occurs;
- inertial forces decrease linearly along a link, moving towards the distal end;
- forces are applied at points lying on the principal axis of each link; and
- forces are aligned with the centroidal axes of each link.

These assumptions have implications on the accuracy of the deflection calculations; their effects are discussed in Section 4.5.3. The last two assumptions can be satisfied by designing modules appropriately: for example, circular cross-sections can be used when the direction of loading of a link is unknown, and a module's joint and connector locations can be chosen to lie along the major axis of a module segment.

4.5.1 Computing deflections

The first step in computing link deflection is to compute the forces applied to each link; the computed torque method (see Section 4.4.1) is used for this. The link is treated as a cantilevered beam, with the proximal end fixed and inertial forces and moments distributed along the length of the link. Darwin2K computes a link force context for each link; the context contains all of the kinematic and dynamic information required to compute deflections for the link. Each link is then split into *module segments*, consisting of continuous segments belonging to the same module. For each module segment, the containing module is queried for the segment's deflection; the module, in turn, sets up the structural parameters (cross-sectional areas, moments, and material moduli) for a deflection calcu-

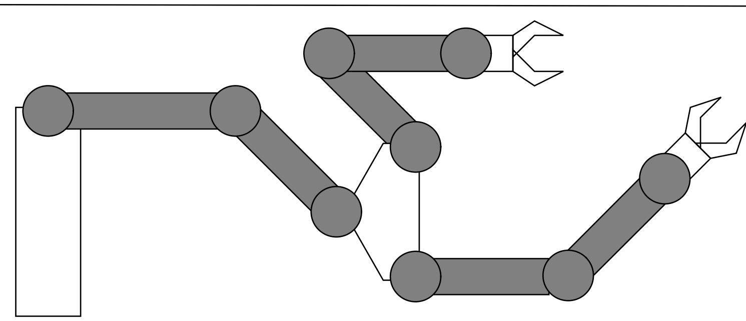


Figure 4.7: Links for which deflections are computed
Link deflection is computed only for links in the interior of serial chains, indicated by dark shading in this figure. Links at the beginnings, ends, and bifurcations of serial chains are not considered (unshaded links).

lation and uses Darwin2K's generic deflection computation procedure. Each of these calculations is performed on a segment with uniform cross section and material; a module may subdivide its module segments further if the cross-section varies discretely[3]. Darwin2K then combines the deflections from each module segment to compute the total link deflection, and combines the link deflections for all links to compute the deflection at each of the robot's endpoints.

Most of the process described above is essentially bookkeeping; the bulk of the deflection computation lies in integrating forces and moments along each segment of a link. The forces and moments acting in the directions of a segment's centroidal axes are integrated separately along the axis of the link to determine the link's deflection. (This is a standard method of computing deflections of simple beams; see a text such as [Shigley89] for the derivation.) The deflections are then superposed to yield the translational and rotational deflections (in three dimensions) at the end of the segment. Figure 4.8 shows the coordinate system used in the deflection computations. The forces and moments in module segment m_2 are integrated along the x-axis from x_1 to x_2. θ_y (the angular deflection about the y-axis) is computed as:

$$\theta_y = \frac{-1}{EI_y}\int_{x_1}^{x_2} n_y(x)dx \tag{4.31}$$

where E is the modulus of elasticity for the module segment, I_y is moment of inertia about the y-axis, and $n_y(x)$ is the bending moment about the y-axis. The linear deflection in the z direction is given by:

$$d_z = -\int_{x_1}^{x_2}\theta_y(x)dx = \frac{1}{EI_y}\int\int_{x_1}^{x_2} n_y(x)dx \tag{4.32}$$

The calculations of angular deflection about the z-axis and of linear deflection in the y direction are similar. The shortening or lengthening of the link in the x direction, and the twist about the x-axis, are given by:

$$d_x = \frac{-1}{AE}\int_{x_1}^{x_2} f_z(x)dx \tag{4.33}$$

$$\theta_x = \frac{-1}{JG}\int_{x_1}^{x_2} n_z(x)dx \tag{4.34}$$

where A is the cross-sectional area of the link, J is the polar moment of inertia of the link, G is the shear modulus of elasticity, f_x is the tensile/compressive force along the x-axis,

3. It is also possible for a module to perform its own special-purpose deflection calculations, though this is not likely to be necessary to achieve a reasonable level of simulation accuracy.

and n_x is the twisting moment about the x-axis. Each module segment contains its own values (computed from the module's parameters) for A, J, I_y, I_z, G, E, x_1, and x_2, and calls Darwin2K's functions for computing the integrals above.

4.5.2 Computing internal forces and moments

The first step in evaluating these integrals is to determine the functions $f_x(x)$, $n_x(x)$, $n_y(x)$, and $n_z(x)$. For a non-moving cantilevered beam with a point force applied at the free end, $f_x(x)$ and $n_x(x)$ are constant and $n_y(x)$ and $n_z(x)$ are linear functions and can be computed at a point z by writing the force and moment equilibrium equations. However, when including an approximation of inertial forces (that is, forces due to the mass of the link when the link is in motion), the order of these functions is raised by one: f_x and n_x are linear, and n_y and n_z are quadratic. We can derive expressions for f_x, n_x, n_y, and n_z by considering the force f_1 and moment n_1 applied at the base of the link, as computed by the computed torque algorithm:

$$f_1 = f_2 + m\dot{v}_c \tag{4.35}$$

$$n_1 = n_2 + (p_c - p_1) \times m\dot{v}_c + (p_2 - p_1) \times f_2 + I\dot{\omega} + \omega \times I\omega \tag{4.36}$$

$\dot{v}_c$ is the acceleration of the link's center of mass, I is the link's inertia matrix, and ω and $\dot{\omega}$ are the angular velocity and acceleration, respectively. (See Figure 4.8a for definitions of other variables). Instead of computing f_1 and n_1 (the force and moment at p_1), we want the force and moment at a distance x from p_1. To do this, we consider the segment of the link remaining from x to p_2. First, we assume that the location of the center of mass of the remaining portion of the link $p_{cs}(x)$ varies linearly: it is equal to p_c at x=0 (i.e. at p_1) and is equal to p_2 at x=l (i.e. at p_2). The location of the center of mass of the remaining segment of the link is thus:

$$p_{cs}(x) = p_2 + (p_c - p_2)\left(1 - \frac{x}{l}\right) \tag{4.37}$$

and the mass of the remaining segment is

$$m_s(x) = m\left(1 - \frac{x}{l}\right) \tag{4.38}$$

See Figures 4.8b and 4.8c for graphical depictions of $p_{cs}(x)$ and $m_s(x)$. The angular velocity and acceleration of the remaining portion of the link are the same as the for link as a whole, but the linear acceleration varies with x:

$$\dot{v}_s(x) = \dot{v}_1 + \dot{\omega} \times (p_{cs}(x) - p_1) + \omega \times (\omega \times (p_{cs}(x) - p_1)) \tag{4.39}$$

where $\dot{v}_1$ is the linear acceleration at p_1. By substituting m_s and $\dot{v}_s$ into the expression for

f_1 (Equation 4.35), we can compute $f(x)$, the internal force acting at point x on the remaining segment of the link:

$$f(x) = f_2 + m_s(x)\dot{v}_s(x) \tag{4.40}$$

$n(x)$ can be computed by performing similar substitutions into Equation 4.36:

$$n(x) = n_2 + \underbrace{(p_{cs}(x) - (p_1 + x\bar{X}))}_{\text{moment at x due to } f_2} \times \underbrace{m_s(x)\dot{v}_s(x)}_{\text{inertial force from acceleration of c.o.m.}} + \underbrace{(l-x)\bar{X} \times f_2}_{\text{moment arm for inertial force}} + \underbrace{\left(1 - \frac{x}{l}\right)I_{lumped}}_{\text{linearly decreasing angular momentum term}} \tag{4.41}$$

where $\bar{X}$ is a unit vector along the x-axis (i.e. pointing from p_1 to p_2) and

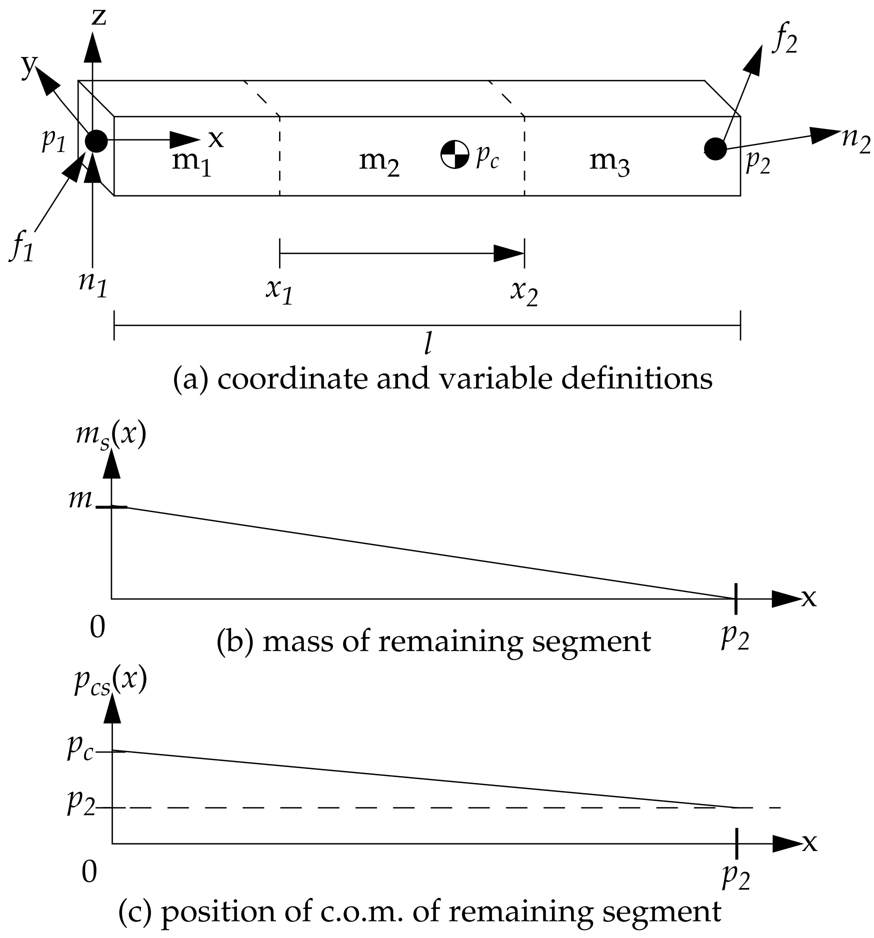


Figure 4.8: Coordinate system used for computing link deflections.
(a) The moments and forces in the beam are integrated from x_1 to x_2 when computing the deflection of module segment m_2. p_1 is the location of the proximal joint, p_c is the location of the center of mass, and p_2 is the location of the distal joint. f_1 and n_1 are applied at p_1 to the link, and f_2 and n_2 are applied at p_2 to the next link. l is the distance from p_1 to p_2. (b) is a graph of $m_s(x)$, the mass of the remaining portion of the link (i.e. to the right of x), and (c) is a graph of $p_{cs}(x)$, the position of the center of mass of the remaining portion.

$I_{lumped} = I\dot{\omega} + \omega \times I\omega$. While this equation looks complex, an understanding of it can be had by realizing that the individual terms are linear in x, and that $n(0) = n_1$ and $n(l) = n_2$. These two equations give the internal forces and moments in three dimensions (with the assumptions noted above) at a point x along the axis of the link. We can use the projections of $n(x)$ and $f(x)$ onto the axes to compute the integrals described in the previous section. Some observations about $n(x)$ and $f(x)$ can be used to simplify the computation of these integrals:

- $f(x)$ is a quadratic function of x;
- $n(x)$ is a cubic function of x; and
- $f(x)$ and $n(x)$ are linear with respect to the forces, moments, velocities, and accelerations.

Taken together, these facts allow Darwin2K to use closed-form solutions to the integrals. This process begins by computing $f(x)$, $n(x)$, and their respective derivatives $f'(x)$ and $n'(x)$ at both ends of each module segment (i.e. x_1 and x_2 in Figure 4.8) and then projecting these vectors onto the x, y, and z axes. Next, a quadratic for $f_x(x)$ (the x component of $f(x)$) is fit to the boundary conditions $f_x(x_1)$, $f_x(x_2)$, and $f_x'(x_1)$. Similarly, cubics are fit to each of $n_x(x)$, $n_y(x)$, and $n_z(x)$ based on their respective boundary conditions. The integrals of the quadratic and cubic functions can then be efficiently found with closed form equations.

To demonstrate this process, consider the robot shown in Figure 4.9. The robot is acted on by a gravitational acceleration of 9.8m/s^2 and is applying a tip force of 98N in the +z direction. We will calculate the linear and angular deflection of the hollow cylindrical portion of the robot's second link (a `hollowTube` module). The tube is aluminum, with an outer diameter of 8cm, a wall thickness of 2.1mm, and a length of 1.1m. The two joints are both `elbowJoint` modules, and the actuator of the second joint is part of the second link (i.e. it is attached to the `hollowTube`). The overall length of the link (between the two joints) is 1.189m. The link's angular velocity is zero, but its angular acceleration is -π/6 rad/s^2 about the y axis. The relevant quantities for this link (including those calculated by the computed torque algorithm such as f_2 and n_2) are:

$$\begin{aligned} p_1 &= (0, 0, 0.335)^T & \dot{v}_1 &= 9.8\text{m/s}^2 \\ p_2 &= (1.189, 0, 0.335)^T & \omega &= 0 \\ p_c &= (0.959, -0.01, 0.335)^T & \dot{\omega} &= (0, -0.524, 0)^T \\ l &= 1.189 & I_{lumped} &= (-0.00851, -0.287, 0)^T \\ x_1 &= 0.045 & f_2 &= (0, 0, 98.97)^T \\ x_2 &= 1.145 & n_2 &= (0, 0, 0)^T \\ m &= 4.11\text{kg} \end{aligned}$$

Given this information, we first need to compute the forces, moments, and their derivatives at x_1 and x_2, the endpoints of the link segment corresponding to the hollow tube. Since the forces in the x and y directions are negligible, as are the moments about x and z, the only significant component of deflection will be from n_y. The boundary values for n_y (as computed by Equation 4.41) are:

$$n_y(x_1) = -151.2 \qquad n_y(x_2) = -4.47$$
$$n_y'(x_1) = 164.6 \qquad n_y'(x_2) = 101.8$$

Since $n_y(x)$ is cubic, we can fit a cubic to these conditions and be assured that it matches n_y exactly. The cubic is:

$$n(d) = -0.3757d^3 - 33.99d^2 + 181.1d - 151.2 \tag{4.42}$$

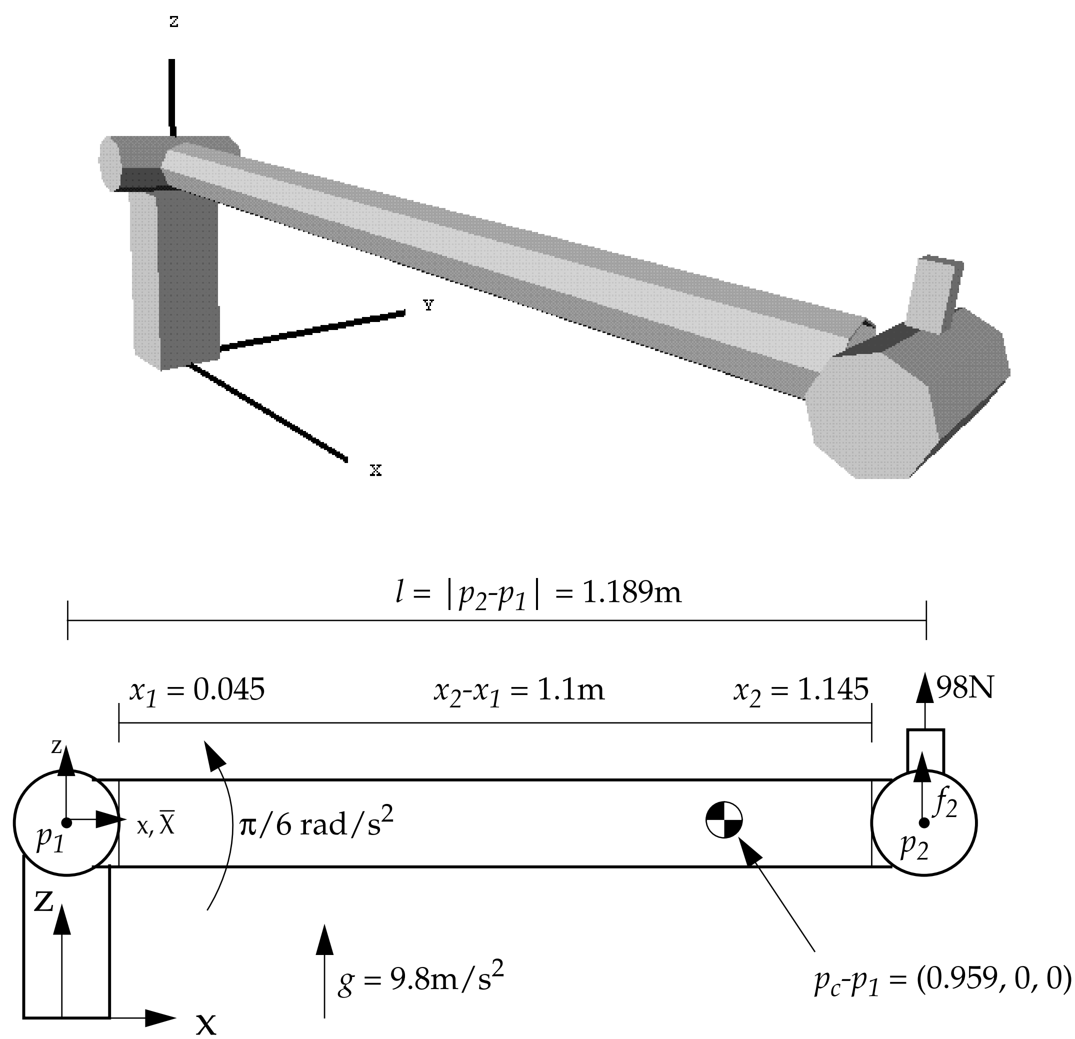


Figure 4.9: Manipulator used in link deflection example.
The world coordinate system is at the base of the manipulator; the coordinate system for deflection calculations is located at p_1. Note that the second joint's actuator is part of the same link as the tube, so the link's center of mass is not at the midpoint of the tube.

where $d = (x - x_1)/l$. To compute $\theta_y(x)$, we must integrate $n(d)$, multiply it by $-1/EI_y$, and account for the scaling introduced by converting from x to d. The tube is made from aluminum, and given its outer diameter of 8cm and wall thickness of 2.14mm, we have:

$$\frac{1}{EI_y} = \frac{1}{(71 \times 10^9)(4.046 \times 10^{-7})} = 3.48 \times 10^{-5} \tag{4.43}$$

The angular deflection is thus:

$$\begin{aligned} \theta_y(x_2) &= \frac{-l}{EI_y}\int_{x1}^{x2} n(x)dx = \frac{-l}{EI_y}\int_0^1 n(d)dd \\ &= \frac{-l}{EI_y}(-0.0939d^4 - 11.33d^3 + 90.55d^2 - 151.2d)\Big|_0^1 = \mathbf{2.76mrad} \end{aligned} \tag{4.44}$$

Integrating and multiplying by l again to find linear deflection in the z direction, we have:

$$d_z(x_2) = \frac{l^2}{EI_y}\int\int_0^1 n(d)dd = \frac{l^2}{EI_y}(-0.01878d^5 - 2.8325d^4 + 30.18d^3 - 75.6d^2)\Big|_0^1 = -\mathbf{2.03}\boldsymbol{mm} \tag{4.45}$$

In this example, only one link segment contributed significantly to the total deflection. In more complex situations, many module segments would be involved, and deflections would not be restricted to a single dimension.

4.5.3 Limitations

As noted in the list of assumptions at the beginning of this section, Darwin2K's analysis of link structural adequacy is based on stiffness rather than peak stress. This assumes that link deflections will become unacceptable before the maximum stress limit of a link's material is reached. This assumption could be eliminated by computing the peak stress in each module segment from the force and moment functions and the cross-section for the segment. A metric could then record the ratio of the peak stress to the material's stress limit, so that links could be optimized to have a desired safety factor with respect to stress.

In the example, the only source of deflection was bending of a beam. It is not unreasonable to assume that the robot's base can be made stiff enough to prevent significant deflection; however, compliance in the robot's actuators, bearings, and fittings are usually significant. If stiffness measurements are available for actuators and bearings, they can easily be incorporated: when a module computes the deflections for a segment containing an actuator, the deflection due to actuator and bearing compliance can be added. Unfortunately, this information is not always readily available, in which case allowances must be made in the acceptability criteria of the optimizer.

Darwin2K's analysis of link deflections is quasi-static: although the effects of inertial forces are approximated, the deflection computations assume that a steady-state deflection has been reached. That is, at every time step, deflection is computed as if the link had been subjected to constant force and moment for a long period of time. Oscillation due to flexure of links is not computed, so the peak deflection measured by Darwin2K may be an underestimate. However, increased deflection due to oscillation will occur mainly when the robot undergoes high acceleration or deceleration; in many cases, high positional accuracy is not required during these period so the transients in link deflection may not be significant in terms of task performance.

Another source of error is the assumption of linearly-varying inertial forces along the module segment. For this assumption to be true, the link's cross section would have to be constant for the entire length between joint axes. In the example above, this was not true since the joint modules at either end of the link also contributed mass. However, in this example and in typical manipulator scenarios, the error in deflection due to the approximation of inertial forces of the link itself will be small compared to the deflections caused by forces applied by the link to a tool's endpoint or to the next link in the robot.

4.6 Path Planning

Because the Jacobian cannot fully express the mobility of a nonholonomic base, planning is required to move the base between poses. Additionally, the Jacobian controller does not perform any obstacle avoidance, so a motion planner is also needed for moving holonomic bases between base poses in the presence of obstacles. While Jacobian-based controllers that deal with obstacles do exist [Khatib86], Jacobian control is a local method and can get trapped by local optima. In contrast, deliberative planning approaches can avoid local optima and generate globally-optimal shortest paths that avoid obstacles. Currently, the `motionPlanner` only generates plans for bases maneuvering in the plane. and obstacles only restrict the motion of the base (not any attached serial chains).

The motion planner utilizes a best-first search of configuration space and uses numerical potential fields to guide the search, as detailed in [Barraquand89]. The robot base's configuration space is discretized (with resolution selected by the user), and a numerical potential field is computed for each cell (C-space state) that is explored by the planner. This C-space potential field is computed from several workspace potentials, each of which is computed for a specific *control point*. A workspace potential guides its corresponding control point towards a goal position and away from obstacles. For planar bases, two control points (defined by each type of base module) are needed to uniquely determine pose.

The workspace potential field for each control point is computed as follows: First, a repulsive potential field is generated by propagating waves from the obstacles (and workspace boundary) through free space. This process starts by labelling all obstacle and boundary cells with a value of one, then assigning a value of 2 to all of the unlabeled neighbors of these cells, then assigning a value of 3 to all of new cells' neighbors, and so on. When two or more of these "waves" from different obstacles meet, the resulting ridge in the potential field forms part of a generalized Voronoi diagram (see e.g. [Preparata88] for more information on Voronoi diagrams), as shown in Figure 4.10. This Voronoi diagram is recorded in a separate grid and contains paths which are a maximum distance away from all obstacles; it provides a skeleton of the workspace along which the safest paths exist.

After computing the skeleton, the goal location of the control point is connected to it by following the gradient from the goal to the nearest point of the skeleton. Next, a potential is computed for points on the skeleton by starting at the goal and again propagating a wave, this time along the skeleton. Finally, the skeleton is expanded in a similar manner to the way obstacles were expanded earlier, with each empty cell receiving a label that is one greater than the smallest label of its neighbors.

The resulting potential field contains a single minima at the goal; for a point robot, following the gradient of the potential will always lead to the goal. To plan a path for a robot with multiple control points, the potential fields for all of the control points must somehow be combined. [Barraquand89] suggests using the minimum of the control points' potential fields for a given state, plus a small fraction (say, one-tenth) of the maximum potential field over all control points. This results in a single C-space potential field (as opposed to the numerous workspace potential fields for the control points) with few local minima.

These local minima can be avoided in the final path by using a best-first search. The search starts at the initial state of the robot and generates all successor states (neighboring cells in the C-space grid). The successors are placed in a priority queue, and the best state in the queue is then expanded as the initial state was. Each base module specifies a list of commands and a function which integrates these commands to move from one state to the next. Using this approach, holonomic and non-holonomic bases can be treated in the same manner. The difference is that holonomic bases can generate commands which move from one state to all neighboring states, while nonholonomic bases are restricted to moving to a subset of the neighboring states.

It is important to note that the paths generated by this algorithm may not exactly reach the goal configuration; the amount of error is bounded by the resolution of the C-

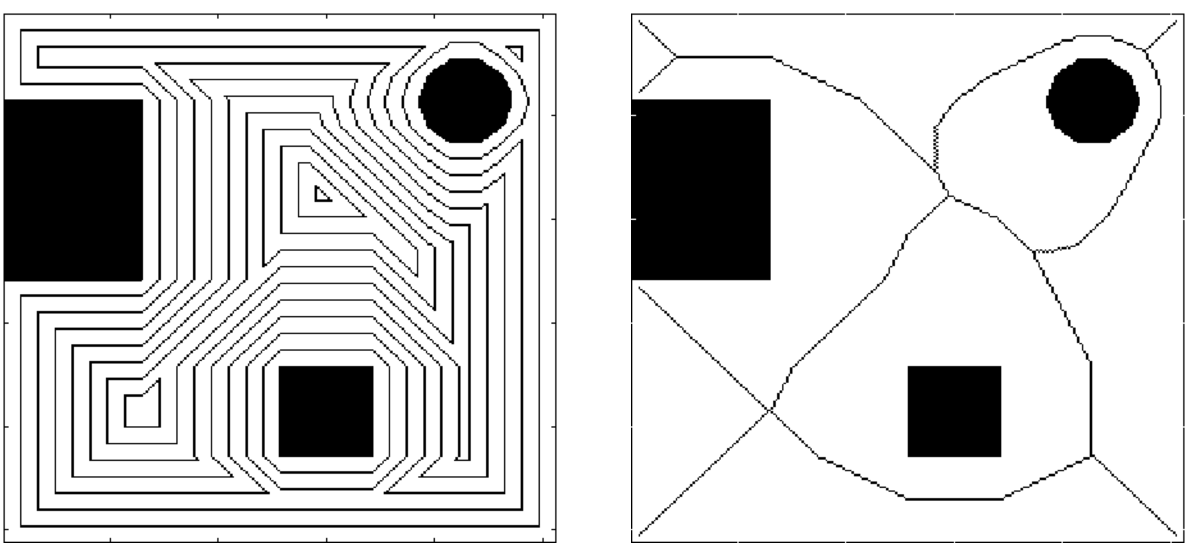


Figure 4.10: Repulsive potential field and workspace skeleton
(a) shows a numerical potential field grown from obstacle boundaries, and (b) shows the generalized Voronoi diagram, or workspace skeleton

space grid. When the robot reaches the end of the path, its pose is set to the desired base pose at the beginning of the next tool trajectory. While a more exact method (or much finer resolution) could be used, there is little useful performance information to be gained by doing so; the end result of the simulation will not be significantly different, but simulation time could increase substantially.

4.7 Other Capabilities and Components

4.7.1 The pathEvaluator

Following Cartesian trajectories with one or more end effectors is common task for robots. To accomodate this task, Darwin2K includes the `pathEvaluator`, which represents tasks as a series of end-effector trajectories, optionally with obstacles in the workspace and payloads to be moved along the trajectories. Figure 4.11 shows two sample tasks that can be simulated by the `pathEvaluator`. In the simplest case, a fixed-base manipulator might follow a single trajectory; a more complex task might require a mobile robot with multiple end-effectors to move between base poses and follow paths with one or both manipulators at each pose. To account for this variation, the trajectories (represented as `path` or `relativePath` objects) are organized in *path groups*: each path group can contain a trajectory for each of the robot's end-effectors, and can have a different pose for the robot's base. If multiple paths are specified in a path group, the corresponding manipulators follow them simultaneously and all paths must be completed before moving on to the next path group. The `pathEvaluator` requires several `evComponents` to be specified in the initialization files: a `DEsolver` for integrating robot state, an `sriController` for controlling the robot, and one or more `paths`. Additionally, the `pathEvaluator` can use a `motionPlanner` to plan paths between path groups for the robot's base, and a `collisionDetector` (discussed in Section 4.7.4) to check for collisions during simulation.

The initial position and orientation of the robot's base can be specified in the initialization file, or can be included as task parameters so that the base position of a manipulator can be optimized. However, when using a mobile base a different base location may be required for each path group, and different mobile robots may require different base poses in order to reach the same path since they may have different manipulator kinematics. The `pathEvaluator` uses the `sriController` in a two-stage process to determine the base pose of mobile robots for each path group: at first, the robot's base is allowed to move freely (i.e. without nonholonomic constraints), and then all of the robot's degrees of freedom (including the base) are moved. If we were to initially use all of the robot's degrees of freedom, then the serial chains might end up at the edge of their workspaces. Instead, the `pathEvaluator` first allows the base to move freely in 3 dimensions (or in the plane for a planar base) while holding all serial chains fixed. Once the robot cannot move any closer to the start of the paths in the path group, serial chain motions are enabled but with a high cost[4], forcing the robot to move the base instead of the serial chains if possible. During this second stage, the robot base is still able to move freely. If the robot still cannot reach the initial points in the paths, we know it cannot complete the task. If it does find a successful starting pose, the base motion mode is returned to normal (i.e. non-holonomic constraints are enforced, for example) and the serial chain cost is returned to normal; this is the normal operating mode for the robot.

If the task requires the robot to move between several base poses, the aforementioned method is used to compute each initial base pose before the actual evaluation takes place. During evaluation, the `sriController` can be used to move holonomic robots between base poses if there are no obstacles present. When a robot with a nonholonomic base moves between base poses, or if there are obstacles in the workspace, the robot can be controlled using the `motionPlanner` described in Section 4.6.

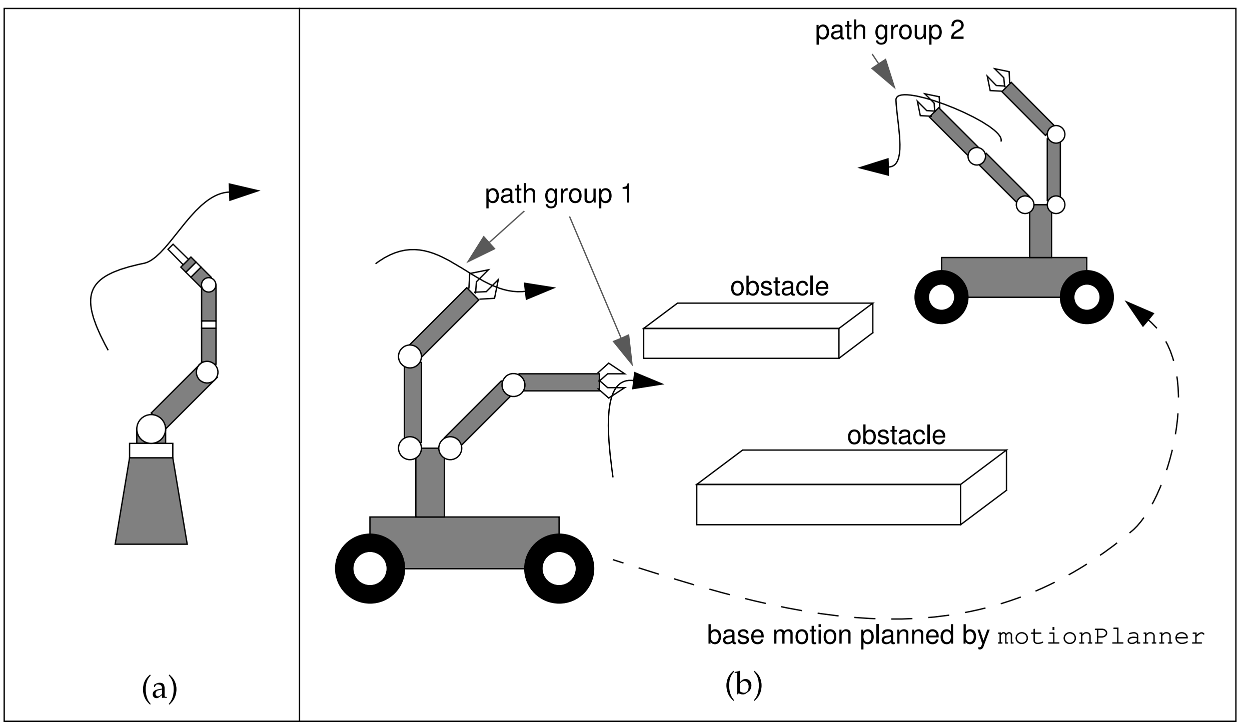


Figure 4.11: Schematic of `pathEvaluator` task representation
The `pathEvaluator` is a general-purpose evaluator for tasks involving moving one or more end effectors along one or more trajectories. Shown here are two example tasks that can be simulated by the `pathEvaluator`: (a) shows a simple task, with a fixed-base robot following a single trajectory, and (b) shows a more complex task, with two groups of paths and a base motion between the path groups planned by a `motionPlanner`.

4. The "high cost" of moving serial chains is implemented by scaling the elements of the Jacobian. The elements for each serial chain are multiplied by a scale factor $s < 1$; SVD will compute a minimum-norm solution, thus using the base's DOFs in preference to those of the serial chains. The elements of $d\theta$ for the serial chain must then also be multiplied by s before being used, since SVD will give values too small by a factor of s.

4.7.2 PID controller

The pidController can be used when it is desirable for the robot to follow joint-space (rather than Cartesian) trajectories. If inverse kinematic equations for a robot are known, they can be used to compute joint positions corresponding to specific endpoint locations; if not, then the sriController (or another method) can be used to compute the corresponding joint positions. The pidController implements a PID control loop for each joint; position, derivative, and integral gains and position and velocity goals can be independently set for each joints. The pidController has different sets of gains for acceleration and torque commands, since the DEsolver will ask for acceleration commands when kinematic simulation is being used and torque commands when dynamic simulation is being used. In the future it may be desirable to add another mode that uses the robot's dynamic model to compute torque commands based on the desired acceleration of each joint, which itself could be computed with the PID control model; however, the pidController was not required for any of the experiments in Chapter 5 so this was not explored.

4.7.3 Payloads

The payload class is an evComponent which represents the geometric and inertial properties of unarticulated (rigid body) payloads. The payload class contains code for reading geometric data from a file, including one or more polyhedra and an arbitrary number of connectors (coordinate frames defined relative to the payload's origin). The payload uses Coriolis ([Baraff96]) to compute its inertial properties from the polyhedra (each of which has a specified density); the payload's connectors can be used to align the payload with other coordinate frames such as an end effector's TCP. Classes derived from the payload can also provide methods to create customized or parameterized geometry, rather than reading geometric data from a file.

4.7.4 Collision detection

Collision detection is an important capability, as it enables the synthesizer to penalize robots that collide with obstacles in the environment or with parts of themselves. The collisionDetector uses the University of North Carolina's RAPID[5] collision detection package [Gottschalk96] to detect intersections between polyhedra representing the geometry of the robot and any obstacles. The collisionDetector computes the number of intersections between polyhedra belonging to different bodies (links, obstacles, or payloads), which can then be recorded by an appropriate metric. To decrease the computational complexity of collision detection, each module can define a low-detail polyhedral model of its geometry and can indicate that collisions between specific polyhedral should be ignored. For example, the prismaticTube shown in Figure 4.12 has a

5. At the time of writing, the RAPID source code is available on the World Wide Web at http://www.cs.unc.edu/~geom/OBB/OBBT.html

normal polyhedral model with 28 polyhedra (8 for each tube section, one for the motor, one for the gearbox, and two for the lead screws) and a low-detail model with only 3 polyhedra (1 for each tube section). Since the three low-detail polyhedra would intersect as the joints move, the `prismaticTube` specifies that collisions between the three polyhedra should be ignored. The use of the low-detail model thus drastically reduces the number of pairs of polyhedra that must be checked for intersections.

4.7.5 Stability and tipover for mobile robots

Tipover can be a major concern for many mobile robots, usually due to either extreme terrain or large payloads. To measure how closely a planar mobile base approaches tipover, the `configuration` class contains code for computing the robot's energy stability [Messuri85]. Each planar (i.e. non-free-flying) mobile base module can define a number of support points relative to its geometry, and the robot's support polygon is computed from the world-space locations of the support points. The configuration then computes the robot's center of mass and determines the minimum amount of energy required to rotate the robot about any leg of the support polygon so that its center of mass is outside the polygon. When the energy is less than zero, the robot is in a tipover configuration. The `pathEvaluator` will abort the simulation if the energy stability is less than zero at any point; however, the energy stability margin can also distinguish between configurations with varying degrees of stability, making it suitable for use as a metric.

4.7.6 Actuator models: power and torque

Every module derived from the `dofModule` (e.g. any `jointModule` or `baseModule`) can define a member function that computes the maximum actuator torque or force for a degree of freedom based on the DOF's velocity, and another member function that computes the power used by a DOF for a given velocity and torque or force. Some of Darwin2K's older joints modules do not include actuator models; these do not have torque limits, and compute power as the product of force and velocity. Darwin2K's newer joint modules include actuator models and use equations based on the motor and gearhead properties to compute power and maximum torque. The properties specified in the component database for each motor and gearbox are listed in Tables 4.3 and 4.4, respec-

Figure 4.12: High- and low-detail models
The prismaticTube module includes both a high-detail model for computing inertial properties and a low-detail model for collision detection. The high-detail model contains 28 polyhedra, while the low-detail model contains only 3.

tively. Each `jointModule` with a motor and gearhead computes a power function $P(\tau,\omega)$ and maximum torque function $T_{max}(\omega)$ from the motor and gearhead parameters:

$$P(\tau, \omega) = V(\tau, \omega)I(\tau) \tag{4.46}$$

where ω and τ are the angular velocity and torque, respectively, at the actuator's output (i.e. at the output shaft of the gearhead), and $I(\tau)$ and $V(\tau,\omega)$ are defined as:

$$I(\tau) = \frac{\tau'}{rk_\tau} \tag{4.47}$$

$$V(\tau, \omega) = \frac{r\omega}{k_s} + I(\tau)R = \frac{r\omega}{k_s} + \frac{\tau' R}{rk_\tau}, \tag{4.48}$$

where the friction torque $\tau_f = rk_t i_o$ and the combined torque $\tau' = \tau + \tau_f$. (These equations are based on motor documentation in [Maxon98].) Note that since τ and ω are the torque and velocity at the actuator's output rather than the motor's output, the equations for I and V include appropriate factors of r to reflect the gear ratio of the gearhead. Sub-

Property	symbol	Property	symbol
mass	m	no-load current	i_0
length	l	resistance	R
diameter	d	rotor inertia	I_{roto}
stall torque	τ_s	torque constant	k_t
continuous torque	$\tau_{c,motor}$	speed constant	k_s
maximum velocity	ω_{motor}	speed/torque gradient	g

Table 4.3: Motor properties

Property	symbol	Property	symbol
length	l	maximum torque	τ_{max}
diameter	d	continuous torque	$\tau_{c,gearhead}$
mass	m	efficiency	η
ratio	r	maximum output velocity	ω_{max}

Table 4.4: Gearhead properties

Actuator models in Darwin2K contain data for motors and gearheads. For geometric purposes, both are modeled as cylinders with specified length, diameter, and mass. The other properties are used to compute an actuator's maximum velocity, maximum torque for a given velocity, and power consumption for a given velocity and torque.

stituting Equations 4.47 and 4.48 in to Equation 4.46, we have:

$$P(\tau, \omega) = \frac{1}{\eta}\left(\frac{\tau'\omega}{k_t k_s} + \frac{\tau'^2 R}{r^2 k_t^2}\right). \tag{4.49}$$

The formula for $T_{\max}(\omega)$ is more straightforward:

$$T_{max}(\omega) = max\left(\left(\tau_s - \frac{\omega r}{g}\right)r, \tau_{max}\right). \tag{4.50}$$

In addition to these two formula, the maximum velocity $\max(r\omega_{motor}, \omega_{max})$ is used to limit joint velocities, and the actuator inertia $I_{act} = rI_{rotor}$ is included in the computed torque model. Finally, the continuous torque rating for an actuator is

$$T_{cont} = max(\tau_{c,\,gearhead}, r\tau_{c,\,motor}). \tag{4.51}$$

These equations can be applied at each simulation time step to limit the torque applied, and power consumed, by each actuator. Since Equations 4.49 and 4.50 account for varying actuator efficiency and maximum torque throughout the actuator's operational ranges, allowing the synthesizer to optimize motor and gearbox selection based on the conditions encountered during task execution.

4.8 Metrics

Metrics provide an interface between Darwin2K's simulation code and the optimization engine by recording aspects of a robot's performance as specified by a task's requirements. Most metrics measure quantities such as power consumption during simulation, though some may directly measure properties of the robot, such as mass. Darwin2K has two broad types of metrics, both of which ultimately produce a single scalar called the *raw fitness*. State-dependent metrics measure quantities that are dependent on the robot's state, which varies with time: thus, they record data at *each* simulation time step before converting the data to a scalar. State-independent metrics do not depend on the robot's state over time, and thus measure performance at the end of simulation. State-dependent metrics can condense a time series of data into a scalar in several ways: finding the minimum or maximum value, calculating the mean, the integral, or the root-mean-square value. For example, a metric that measures power consumption can compute the peak power by finding the maximum in its series of data, or the total energy consumption by computing the integral.

Each metric converts the raw fitness into a form usable by the optimizer, called the *standardized fitness*. Standardized fitness is simply a number greater than zero, with zero indicating the best possible value. Since the minimum and maximum values that are meaningful for each metric can vary with the task requirements, the designer can specify the bounds to which raw fitness values will be constrained. Each metric also specifies the sense of fitness values--a metric is said to have positive sense if a larger raw fitness value is better, and negative sense if smaller raw fitness is better. The metric's sense, bounds on raw fitness, and a scale factor are used to compute standardized fitness:

$$standard\,fitness \;=\; scale \times \begin{cases} max - raw\,fitness & \text{positive sense} \\ raw\,fitness - min & \text{negative sense} \end{cases} \tag{4.52}$$

This provides a uniform interface between metrics and the optimizer and allows meaningful raw fitness values to be mapped to a roughly consistent range. Both the standardized and raw fitness (along with the metric's sense) are reported to the optimizer for each metric. The standardized fitness is used for selecting configurations for reproduction, while the raw fitness (and sense) for each metric determine whether or not a configuration is feasible. Since some metrics may require costly evaluation (e.g. `dynamicPathCompletionMetric` requires dynamic simulation), the optimizer indicates which metrics are currently being used each time a configuration is evaluated. This allows costly simulation methods to be used only when necessary. For example, the early stages of optimization may require only a few kinematic metrics, while later stages may need dynamic simulation to evaluate some metrics.

Darwin2K contains a set of core metrics that are likely to be useful for many synthesis problems. They are described in the following two sections, and summarized in Table 4.5. These metrics interface with most of Darwin2K's existing simulation capabilities such as dynamic simulation and link deflection computation.

4.8.1 State-Independent metrics

State-independent metrics measure performance at the end of simulation. While some state-independent metrics (such as the mass metric) can make their measurements before simulation, others depend on the final simulation result (but not on the robot's state at intermediate time steps). Currently, Darwin2K include five state-independent metrics. The `timeMetric` records the time required to complete a task. The `massMetric` computes the mass of the robot. The `pathCompletionMetric` and `dynamicPathCompletionMetric` both measure the fraction of trajectory waypoints reached by the robot. The two are functionally identical, though the `dynamicPathCompletionMetric` indicates that dynamic simulation should be used. This allows dynamic simulation to be used only in the later stages of optimization, when the system has generated robots that meet the kinematic task requirements. Finally, the `taskCompletionMetric` is similar to the other two completion metrics but is more general and is not tied to Darwin2K's trajectory representation; it can be used by any evaluation method to indicate the degree of task completion.

4.8.2 State-Dependent metrics

State-dependent metrics measure physical quantities that depend on the robot's state over time. Quantities such as joint velocity can be copied directly from the robot's state, while others (such as link deflection) must be computed explicitly as they are not normally calculated during simulation.

The `powerMetric` records the power used by the robot at each time step. Power for a degree of freedom is computed by the DOF's corresponding `module`, and power for all DOFs is summed at each time step. The most useful measurements for this metric are the maximum (maximum instantaneous power) and integral (total energy consumption). The `peakVelocityMetric` records the maximum joint velocity over all joints at each time step, and is useful when actuator models are not available to limit joint velocities.

There are three metrics for quantifying actuator forces: the `actuatorSaturationMetric`, the `continuousSaturationMetric`, and the `peakTorqueMetric`. The `peakTorqueMetric` should be used only when joint modules do not include actuator models; it assesses actuator requirements by recording the maximum joint torque (over all of the robot's degrees of freedom) at each time step. The `actuatorSaturationMetric` and `continuousSaturationMetric` are used to evaluate the adequacy of a robot's actuators by measuring how closely each actuator approaches its torque or force rating during simulation. The former considers both peak and continuous torque ratings, while the latter considers only continuous torque ratings and is used in conjunction with dynamic simulation (when each actuator's applied torque is already limited to its peak rating). The general idea for both metrics is that they should be less than or equal

Table 4.5: Summary of core metrics

Metric	State Dep?	Description
`timeMetric`	no	Measures task completion time
`massMetric`	no	Measures robot mass
`pathCompletionMetric`	no	Measures fraction of path waypoints reached
`dynamicPathCompletionMetric`	no	Same as `pathCompletionMetric`, but implies that dynamic simulation should be used
`taskCompletionMetric`	no	Measures fraction of task completed (general-purpose)
`actuatorSaturationMetric`	yes	Measures peak and continuous actuator saturation
`continuousSaturationMetric`	yes	Measures continuous actuator saturation
`peakTorqueMetric`	yes	Measures peak joint torque over all joints
`peakVelocityMetric`	yes	Measures peak joint velocity over all joints
`powerMetric`	yes	Measures power
`positionErrorMetric`	yes	Measures cross-track error of end effector
`rotationErrorMetric`	yes	Measures rotation error of end effector
`stabilityMetric`	yes	Measures energy stability (closeness to tipover)
`collisionMetric`	yes	Measures self-collisions and collisions with obstacles
`linkDeflectionMetric`	yes	Measures linear link deflection at robot's endpoint(s)

to 1 if all of the robot's actuators are operated within their torque limit during the entire simulation, and greater than 1 if any actuator exceeds it peak or continuous torque rating. The `continuousSaturationMetric`'s value is computed as follows: At each time step, the saturation (ratio of applied torque to continuous torque rating) is computed for each actuator, and a running average of saturation for each actuator is computed. At the end of simulation, the running average for actuator i contains the continuous saturation C_i: if it is less than one, the continuous torque was less than the actuator's continuous torque rating, and if it was greater than one the continuous torque rating was exceeded. For the `continuousSaturationMetric`, the maximum value (computed as the maximum over all C_i) is normally used as the standard fitness.

The `actuatorSaturationMetric` is more complicated, as it must determine whether (and how much) any actuator's peak *or* continuous torque ratings are exceeded. It records the continuous saturation for each actuator in the same manner as the `continuousSaturationMetric`, and also computes the peak saturation $P_i(t)$ for actuator i at every time step:

$$P_i(t) = \min\left(0, \frac{\tau_i(t)}{\tau_{i,\,max}} - m\right) \tag{4.53}$$

where $\tau_i(t)$ is the torque applied by actuator i at time t, $\tau_{i,max}$ is the maximum torque rating for actuator i, and m is the metric's minimum value (as specified by the designer). The maximum $P_{max}(t)$ over all $P_i(t)$ is recorded for each time step; if $P_{max}(t)$ is greater than zero (that is, if any actuator had saturation greater than the minimum m), then m is added to $P_{max}(t)$. This may seem like a complicated way to compute saturation, but it has the following convenient properties:

- if all actuators have saturation < m, then $P_{max}(t)$ is 0;
- if any actuator has saturation $\geq m$ but all have acceptable saturation, then $m \leq P_{max}(t) \leq 1$; and
- if any actuator has saturation > 1, then $P_{max}(t)$ is > 1

This allows differentiation between very low saturation (i.e. less than m), acceptable saturation (between m and 1), and unacceptable saturation (greater than one). After simulation is complete, the continuous and peak saturations are combined by adding a total continuous saturation C_{total} to every recorded value of P_{max}:

$$C_{total} = offset + \sum_{i=1}^{numActuators} \min(0, C_i - m) \tag{4.54}$$

where *offset* is 0 if all C_i were 0, and m otherwise. After adjusting P_{max} in this manner, the statistics (minimum, average, etc.) are computed as normal over P_{max}. This formulation was successfully used for the experiments presented in the next chapter; however, in retrospect it has the unfortunate property that even if all actuators have continuous saturation less than one but a sufficient number have continuous saturation greater than m, then

C_{total} can be greater than one and thus make the configuration's actuators appear exceed their continuous torque ratings.

Two metrics measure how closely a robot follows a trajectory: the `positionErrorMetric` and `rotationErrorMetric`. Both of these metrics work with Darwin2K's path representation, and measure deviation from the commanded trajectory. Specifically, the `positionErrorMetric` measures the end-effector's cross-track error as it moves between waypoints, and the `rotationErrorMetric` measures the angle between the end-effector's orientation and the commanded orientation.

Several other metrics remain; these record robot performance as computed by methods described earlier in this chapter. The `stabilityMetric` measures how closely a mobile robot approaches tipover by recording the robot's energy stability, as computed by the method described in Section 4.7.5. The `collisionMetric` measures the robot's self-collisions and collisions with obstacles during task execution, as described in Section 4.7.4. At each time step, the `collisionMetric` queries the `collisionDetector` for the number of intersecting polyhedra. Either the maximum or integral should be used for the `collisionMetric`. Finally, the `linkDeflectionMetric` records the sum of the linear deflections at each of the robot's end effectors, as computed by the method described in Section 4.5.

4.9 Summary

This chapter detailed Darwin2K's simulation architecture and capabilities. A method for deriving a symbolic model of a robot's dynamics, which can then be used for forward and inverse dynamic calculations, was presented, as was a method for estimating link deflections based on actuator and applied forces and torque. Three controllers (the `sriController`, `ffController`, and `pidController`) were described, and methods for evaluating a robot's stability, detecting collisions, and modeling actuator behavior were presented. This chapter also described the `pathEvaluator`, which can be used for evaluating fixed-base and mobile manipulators for trajectory-following tasks. Finally, this chapter described the metrics used by Darwin2K to measure robot performance. In the next chapter, we will see how these capabilities are combined with those of the synthesizer to create robots for a variety of tasks.

5 Experiments and Demonstration

The preceding chapters described the details of Darwin2K's principle components: robot representations, system architecture, the synthesis algorithm, and simulation and analysis methods for evaluating robot configurations. This chapter puts those pieces together to demonstrate Darwin2K's capabilities, describe its usage, support claims about the synthesizer, and provide insight into key issues and limitations. Results are presented for six synthesis tasks; some demonstrate the breadth and depth of Darwin2K's scope, while others characterize the synthesizer's performance. The first experiment is synthesis of a free-flying, dual-armed robot for satellite servicing, and should give the reader an understanding of how tasks are specified in Darwin2K while demonstrating synthesis of a complex robot. The second experiment synthesizes fixed-base manipulators for a trajectory-following task, and includes results for five different sets of initial conditions which vary the modules and starting configurations used by the synthesizer. The third experiment uses a simplified version of the second task, and characterizes the performance of the Commonality-Preserving Crossover operator and subgraph preservation through over eighty runs of the synthesizer. The third experiment also compares performance of the selection algorithms described in Chapter 3 through another series of synthesizer trials.

The fourth experiment synthesizes a mobile manipulator for a material-handling task. Chronologically, this was the first synthesis task addressed by Darwin2K and it revealed several shortcomings which motivated the development of some of Darwin2K's features. The fifth experiment performs kinematic synthesis of an antenna-pointing mechanism, resulting in several designs whose optimality is easily understood and which are similar to manually-designed mechanisms. Finally, the sixth experiment synthesizes walking machines for a zero-gravity truss inspection task. This synthesis task makes use of dynamic simulation and includes a number of task and control parameters in the synthesis process. The synthesized configurations are novel and are at first surprising, though analysis reveals a well-optimized kinematic structure with inherent minimization of self-collisions. This experiment also synthesizes robots for only one portion of the entire task, demonstrating the importance of the task description in determining the final form of the synthesized robots.

The experiments in this chapter demonstrate and characterize Darwin2K's simulation and synthesis capabilities, including dynamic simulation, inclusion of task-specific simulation components, synthesis of robot actuator selection and structural properties, optimization of task parameters, and optimization methods for multiple metrics. The synthesis experiments should give the reader a grasp of Darwin2K's abilities and limitations as well as provide motivation for future extensions. The chapter concludes with a summary and discussion of important issues raised by the experiments.

5.1 Task 1: A free-flying robot for orbital maintenance

The Ranger Telerobotic Flight Experiment [SSL99] was designed to evaluate on-orbit telerobotic operation of a highly capable robot for maintenance and assembly. The Ranger vehicle has a torso to which four arms -- one for grappling with a satellite or other payload, two for dextrous manipulation, and one for sensor positioning -- are attached. The torso is also attached to a housing containing power, computing, and control electronics. Originally, Ranger was to also have a free-flying base with solar panels, a reaction control system, and positioning sensors. Due to programmatic issues, the flight experiment was scaled back to take place on board the Space Shuttle payload bay, with the grappling arm fixed to a pallet in the Shuttle's payload bay. However, the Ranger Neutral Buoyancy Vehicle (NBV) has been successfully used for teleoperation experiments in an underwater simulated space environment. The synthesis task presented in this section is loosely based on the tasks for which Ranger was designed. An earlier version of this experiment was presented in [Leger99].

We will use Darwin2K to synthesize a robot similar to Ranger in terms of its capabilities, and to generate kinematic, dynamic, actuator, structural, and controller properties for the robot. Since the robot is free-flying and will interact with large, freely-moving payloads, reaction forces from manipulator motions will cause motion of the robot's base; thus we will need dynamic simulation to evaluate each robot. To limit the complexity of the synthesis problem, we will limit the synthesis process to symmetric two-armed designs with one arm acting as the grappler, rather than four-armed configurations like Ranger. A free-flying robot with two dextrous manipulators will have 18 or more degrees of freedom; dynamic simulation of robots of this complexity is fairly costly, and simulation of a four-armed robot would be even more expensive (though within the capabilities of Darwin2K). The restriction to a two-armed design reduces evaluation time and the size of the design space, making the synthesis problem more tractable given available computing power.

5.1.1 Task specification

The first step in the synthesis process is to enumerate the capabilities that are desired in the robot. At the highest level, some key operational abilities and features for on-orbit maintenance are:

- grappling with target payload and performing relative body positioning;
- adequate workspace for manipulators without mechanical interference (collisions); and
- ability to remove and insert Orbit-Replaceable Units (ORUs).

These capabilities will drive the task description used for evaluating configurations. Clearly, we cannot hope to simulate every possible task during evaluation; instead, we must create a *representative task* that exercises candidate designs in the operations that will

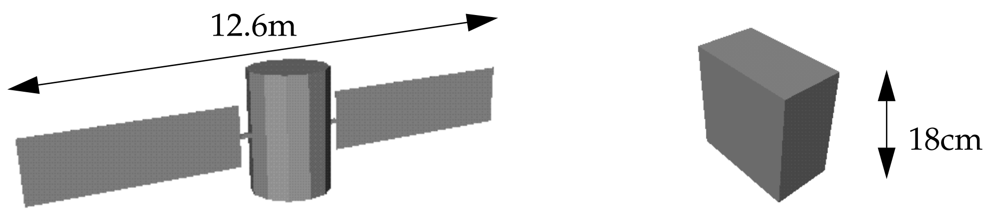


Figure 5.1: Satellite and ORU payloads for free-flyer
The satellite (left) is 3m tall, 2m in diameter, has two 5m by 1.5m solar panels, and has a mass of 9081kg. The ORU is 10cm wide, 18cm tall, 18cm deep, and has a mass of 4.86 kg. During execution of the task, the robot removes the ORU from the satellite and re-inserts it at another location.

be typical of normal usage. The representative task should require motions and forces that are characteristic of those that will be encountered during orbital servicing.

During the chosen task, one manipulator will be used for grasping the satellite and repositioning the robot's base, while the other will be used for manipulating the ORU. The satellite and ORU are shown in Figure 5.1. The robot's base will be unactuated during this task: the grapple manipulator will be used for all base motions, as the robot will start within reach of the satellite and will not have to maneuver to it. The first step of the task is to move both of the robot's end effectors to neutral positions in front of the robot (Figure 5.2a). Since the robot's joints may start in arbitrary positions, collision detection is not performed as the robot moves to the initial neutral position. Next, the work manipulator traces a rectangular path (Figure 5.2b). All of the trajectories in the task are relative to either the robot or satellite, so their world-space positions move as the robot and satellite move. The next step is for the grappling manipulator to move to the grasp point on the satellite (Figure 5.2c). After grabbing the satellite, the robot uses the grapple manipulator to move the robot base to a known location relative to the satellite (Figure 5.2d). The work manipulator then approaches the ORU and grasps it (Figure 5.2e). Removing the ORU (again Figure 5.2e) consists of using a special tool (the Microconical End Effector or MEE ([SSL99])) to rotate a fixture on the ORU through 90 degrees (requiring 10 ft-lbs (or 13.6Nm) of torque), then pulling the ORU out of its docking bay (requiring 10 lbs (44.5N) of force)[1]. Once the ORU has been extracted, the end effector is moved further away from the satellite so that the ORU does not collide with the satellite during the base repositioning that follows (Figure 5.2f). Finally, the work manipulator moves the ORU to an approach position before re-inserting it (Figure 5.2g); the insertion motions, torques, and forces are opposite of those used for removal.

It is hard to predict the optimal values for some of the task and controller properties: where should the base be located relative to the satellite at the beginning of the ORU removal phase of the task? How fast should the robot accelerate and move when moving between base positions? These factors can influence the robot's task completion, number of collisions, and the time and energy required to executes the task, and it would be useful

1. These figures are based on the description of the Ranger Telerobotic Shuttle Experiment, one part of which consists of removing an ORU and re-inserting it.

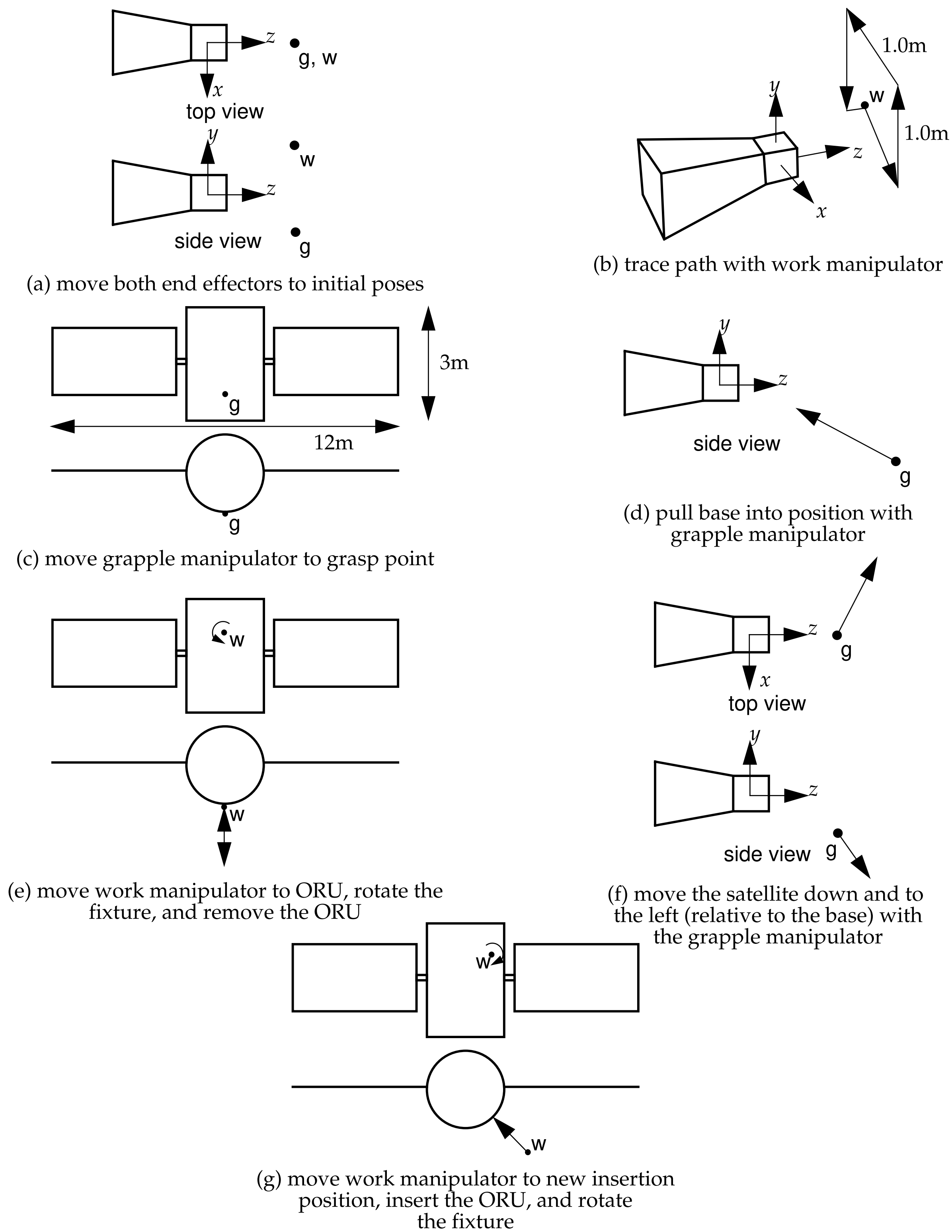


Figure 5.2: Satellite servicing task

The task consists of a number of trajectories for each end effector. Paths for the work and grapple manipulators are denoted by w and g, respectively; paths relative to the base are shown with the base, and paths relative to satellite are shown with the satellite.

class	component label	task parameter	min	max	# bits
pathEvaluator	(none)	originPosY	2.0m	2.5m	4
		originPosZ	1.0m	2.0m	4
relativePath	basePath	vel	0.1m/s	1.0m/s	4
		maxAcc	0.3m/s^2	2.0 m/s^2	5
relativePath	basePath2	vel	0.1m/s	1.0m/s	4
		maxAcc	0.3m/s^2	2.0 m/s^2	5

Table 5.1: Task parameters to be optimized

The first two task parameters are for the `pathEvaluator`; the remainder are for the two `relativePaths` that describe the base repositioning motions.

for the synthesizer to optimize these parameters. Fortunately, these variables can be included as task parameters and can thus be optimized by the synthesizer. We will include the maximum velocity and acceleration for each base motion as task parameters, as well as the Y and Z components of the initial base pose (the X component will be zero so that the robot is aligned with the satellite's centerline). These 6 task parameters (detailed in Table 5.1) will be included with *each* configuration sent by the synthesizer for evaluation.

5.1.2 Constructing a task-relevant simulator

Based on the task description, the next step in the synthesis is to create the simulator that will be used for evaluation. This includes the selection of appropriate `evComponents` for simulation and control, and also includes a task-specific `evaluator`, the `ffEvaluator`. For this task we will need two `payload` components: one for the satellite, and one for the ORU. Some of the endpoint trajectories for the robot will be defined relative to the satellite, since the robot will be performing actions on it (grappling with it, removing the ORU, and re-inserting it) and reaction forces will cause the satellite to move. We will also need the `collisionDetector`, `rungeKutta4`, and `ffController` components for the simulation to measure collisions, perform numerical integration of the robot's state, and provide torque commands for the robot, respectively. The trajectories for this tasks are specified as `relativePaths`. The initial kinematic trajectories and the trajectories for base motions are specified relative to the robot's base, and the remaining trajectories are specified relative to the satellite.

The simulated task needs to exercise the robot's capabilities as mentioned above: grappling with the satellite, removing and inserting ORUs, and performing manipulation within a reasonable workspace. Most of these capabilities could be simulated with a `pathEvaluator` (and the `evComponents` outline above), but two specific assumptions that simplify the simulation of this task require the use of a task-specific `evaluator`. First, we will be using kinematic simulation for part of the task (when evaluating the adequacy of the robot's workspace), and dynamic simulation for the rest of it. Second, we will be locking the joints of the grappling arm when it is not actively moving the robot relative to the satellite. While this assumes that the brakes on the robot's grappling arm are adequate, it also greatly improves the numerical stability and computational efficiency of the dynamic simulation. When the grappling arm's joints are locked, the dynamic equations are simplified reflecting the removal of the equations and variables corresponding to the grappling arm's joints. Another simplification is that the satellite's geometry does not contain ports for inserting the ORU, so we must disable collision detection between the ORU and satellite. This may seem like a bad idea, but as long as we require the robot to follow trajectories with reasonable precision and the specified trajectories would not cause the ORU to collide with the satellite in reality, then this does not significantly affect the outcome of the simulation. Apart from these changes, the `ffEvaluator` behaves like the `pathEvaluator`: the robot's end effectors follow a series of trajectories. All told, the `ffEvaluator` requires 5 member functions to be specified:

- a constructor to set initial values (4 lines of C++)
- `readParams` - reads parameter values from the initialization file (17 lines)
- `postComponentInit` - a function to disable collision detection between the ORU and satellite (17 lines)
- `init` - initializes paths at beginning of each simulation(19 lines)
- `evaluateConfiguration` - the main simulation loop (217 lines; largely copied from `pathEvaluator::evaluateConfiguration`)

While 300 lines is not a trivial amount of code, it is not a large amount considering that it creates an efficient, task-specific dynamic simulation of a free-flying robot with multiple manipulators. It is important to note that the code does not contain any "simulation guts"; rather, it initializes, controls, and monitors various `evComponents` to customize their behavior.

During the insertion and removal portions of the task, a `reactionForceCalculator` is used to compute the reaction forces and torques that are felt at the grapple manipulator's endpoint due to the forces and torques applied by the work manipulator. Since the robot and satellite are a closed system (i.e. there are no external forces or moments acting), any forces exerted by one end effector are felt by the other, transmitted through the satellite's structure. Darwin2K's dynamic simulator does not model forces arising from contact or friction, which could be used to simulate the insertion and removal forces and moments; thus, the `reactionForceCalculator` is needed to account for the fact that the forces applied by the end effector must be also applied *to* something. Note that if the `reactionForceCalculator` were not used, then the entire system (robot, satellite, and ORU) would accelerate during the insertion and removal phases since an unbalanced (or, equivalently, external) force would be applied.

While constructing the simulation (i.e. writing the code, specifying trajectories, and setting control and simulation variables) it is useful to have a manually-designed robot to use for debugging purposes. This robot does not need to be able to complete the entire task or meet all performance requirements, but it is helpful if the robot can at least kinematically perform most of the task so that the designer can make sure the simulator is properly set up. Fortunately, it is easy to for the designer assemble a robot from parameterized modules, and changes to the test robot and `evComponent` and evaluator variables can be quickly iterated since they are specified in text files. In addition to debugging the simulation, this process gives insight into robot topologies or parameter values that may be useful to include in the kernel configurations for the task. It also gives the designer an idea of the expected range of values for performance metrics and for `evComponent` variables that will be included as task parameters. For this task, initial simulations with a manually-specified robot indicated that task completion time would be in the neighborhood of 50 seconds, energy usage would be about 1 to 2 kJ, and robot mass would be at least 300kg. These values will be used in setting the ranges for performance metrics, as described in the next section.

5.1.3 Performance metrics

We have just detailed the task specification and the simulator that will be used to evaluate robots; the next step is to specify how each robot's performance will be measured by selecting a relevant set of performance metrics and acceptance thresholds. The synthesizer will be using Requirement Prioritization to select configurations for reproduction and deletion, so we must specify priorities and acceptance thresholds for the metrics based on their significance to the task. First and foremost, there is task completion: the robot should be able to complete the entire task we assign it. Secondly, there should be no collisions between either robot and satellite, or between different links of the robot. Thirdly, the robot should be able to follow end-effector trajectories with some degree of accuracy since precise positioning of the end-effector will be necessary when removing and inserting ORUs. These three requirements will comprise the first requirement group:

- `pathCompletionMetric` = 100%
- `collisionMetric:` integral = 0
- `positionErrorMetric:` maximum <= 3mm

Additionally, we can give some constraints to ensure appropriate actuator selection and sizing of structural geometry. To ensure that the robot's actuators can provide necessary torque during operation, we can require that the continuous (average) torque of each actuator is less than its continuous torque rating. Since we will be using dynamic simulation and since the torques commanded by the robot's controller will be clipped to the actuator's peak limits, any effects of peak saturation (i.e. demanding more torque than an actuator is capable of) will directly show up in the robot's behavior as deviations in trajectory. Thus, we will only monitor continuous saturation, and allow brief periods of peak saturation as long as they do not adversely affect performance with respect to the other metrics such as path tracking error. We will also use link deflection as a metric to ensure that link cross-sections are adequate for the forces and moments applied during the task; a maximum allowable link deflection of 1mm over the length of the robot's arm seems reasonable. Thus, for the second requirement group we have:

- `continuousSaturationMetric` < 80%
- `linkDeflectionMetric:` maximum <= 1mm

Two other factors that are particularly important for space applications are mass and energy; a robot should ideally minimize both of these. While we could specify acceptability thresholds for these, it is generally preferable to reduce them as much as possible as they allow for additional capabilities to be added and provide a margin for growth in case any subsystems require more mass or energy than anticipated. Finally, we would like to minimize task completion time as well to make efficient use of the robot. These three metrics constitute the third, and final, requirement group:

- `massMetric`: minimize
- `powerMetric`: minimize integral
- `timeMetric`: minimize

One observation is that the first requirement group consists of kinematic metrics, although path tracking error may also have components due to actuator saturation. To reduce total runtime for the synthesis process, we can use kinematic simulation for the entire task while optimizing the first requirement group, and then use dynamic simulation only for later metrics. This can be accomplished by adding the `dynamicPathCompletion` metric to the second requirement group, which indicates to the evaluator that dynamic simulation should be used for simulating part of the task (as described in Section 5.1.2) when the `dynamicPathCompletion` metric is being considered by the synthesizer. This will speed the early stages of synthesis, when few robots are able to meet the kinematic requirements. While the effects of actuator saturation on accuracy will be ignored during the first requirement group, the effects of kinematic singularities and controller parameters will be accounted for and actuator saturation will be addressed during the second requirement group. (Note that if accuracy of the robots does not meet the 3mm acceptance threshold when using dynamic simulation, then Requirement Prioritization will increase the weight for position error to lead the synthesizer to improve accuracy.)

Based on these values, the minimum and maximum values were set outside the expected range of values, since there is no particular reason to clip metric values for mass, energy, and time. For position error and link deflection, however, it makes sense to set lower bounds: a maximum link deflection of 0.5mm is not effectively any better than a maximum link deflection of 1mm, and maximum position error of 1mm is not any better (for the task at hand) than 3mm. Thus, we set a minimum of 1mm for link deflection, and 3mm for position error because there is no significant operational difference between values below these thresholds.

Table 5.2 summarizes the metrics used for this task. The minimum and maximum values listed for each metric specify the range of values that are meaningful for this task; all metric values greater than the maximum are to be considered equivalent in terms of suitability for the task, and all values less than the minimum are equivalent to each other as well. The scale values were computed using Equation 3.3; for convenience, we can define the adjusted fitness doubling increment (AFDI) as the improvement in the raw value of the metric that leads to a doubling in adjusted fitness:

$$AFDI = \frac{\ln(2)}{scale} = \frac{0.693}{scale} \tag{5.1}$$

The AFDI gives a feeling for how changes in performance affect selection probability, and

Table 5.2: Metrics for free-flying robot

Metric name	Acceptance threshold	Min	Max	Scale	AFDI
`pathCompletion`	= 1.0 (100%)	0	1	6.93	0.1 (10%)
`collisionMetric`	integral = 0	0	100	0.4	1.73
`positionErrorMetric`	max < 0.003m	0.0029m	0.1m	100	7mm
`dynamicPathCompletionMetric`	= 1.0 (100%)	0	1	6.93	0.1 (10%)
`linkDeflectionMetric`	max < 0.001m	0.0009m	0.01m	693	1mm
`continuousSaturationMetric`	max < 0.8 (80%)	0.3	100.0	6.9	0.1 (10%)
`massMetric`	none	350kg	700kg	0.035	20kg
`powerMetric`	none	400J	20kJ	0.002	350J
`timeMetric`	none	30s	60s	0.34	2s

was used as a guide when selecting scale values.

Some of the values for scale (or, equivalently, AFDI) and for minimum and maximum were set based on previous experience in applying Darwin2K to other synthesis tasks; others were based on the initial simulations with manually-generated designs. The ranges for `pathCompletion` and `dynamicPathCompletion` are effectively fixed; it is impossible to have a path completion of less than zero or greater than one. An AFDI of 0.1 (or scale of 6.93) for path completion (and dynamic path completion) has worked well in the past; similarly, a scale of 0.4 (AFDI = 1.73) works well for the collision metric, and an AFDI of 0.1 (scale of 6.9) is adequate for actuator saturation. These values are not particular task-dependent and are reasonable default values. They provide very strong selection pressure towards early configurations that perform well, and still provide sufficient selection pressure when approaching the acceptability thresholds for their respective metrics. The other values, however, are task-specific: position error, link deflection, mass, time, and energy can vary substantially between different tasks, and so the scale, minimum, and maximum values should be set after running initial simulations to obtain ballpark estimates (e.g. will a typical robot weigh 5kg or 500kg? Will it require 200J or 20kJ of energy?) of the range for each metric. The synthesizer can tolerate a large amount of variation in scale values and still provide effective optimization; the effect of the scale value is to determine how much the synthesizer focuses on good configurations, which in turn affects the rate of improvement. When in doubt, it is better to choose a larger scale value rather than a smaller one so that there is sufficient selection pressure when metrics are well-optimized; the initial optimization of a metric is often fairly rapid, and most of the optimization time is spent on the last 10 percent or so of improvement. Thus, the AFDI values for mass, energy, and time were set to provide reasonable selection pressure for well-optimized configurations: a mass reduction of 20kg can be a significant improvement for a 400kg robot; 350J is a significant reduction in energy consumption, and 2 seconds is a moderate improvement over a 50s task. For link deflection, the AFDI is 1mm;

for position error, it is 7mm. These values are not very critical, but again were set so that there would be significant selection pressure as configurations approached the minimum values. As mentioned previously, the scale, minimum, and maximum values for the remaining metrics (`pathCompletionMetric`, `collisionMetric`, `dynamicPathCompletionMetric`, and `continuousSaturationMetric`) are not task-specific and were set based on experience with previous synthesis tasks. These values are reasonable defaults, and there is no immediate reason to suspect they will be inadequate for any other synthesis task.

5.1.4 Selecting synthesizer primitives and parameters

After composing the simulation and selecting appropriate performance metrics, the next step is to specify the module database and kernel configuration used by the synthesizer. The kernel configuration (see Section 3.4.1) provides a starting point for creating configurations, and determines the overall topology of the robot--in this case, a free-flying base with two symmetric arms. The module database contains modules that the genetic operators will insert into configurations, or replace other modules with (Figure 5.3). Based on the task outlined above, two task-specific (as opposed to general purpose) modules will be required: a free-flying base module, and a tool module representing the MEE (Figure 5.3). The specifications for the MEE module, called the `MEETool`, were based on the on-line Ranger documentation. The `MEETool` is cylindrical in shape with a length of 30cm and a diameter of 7cm; its mass is 4.6kg. The `MEETool` has no parameters. The `ffBase` (free-flyer base) is also based on Ranger. It consists of a 30cm cube to which the arms are attached via two connectors, and a truncated prism 60cm by 60cm at the bottom, 30cm by 30cm at the top, and 90cm tall; it has a mass of 260kg. The `ffBase` has one parameter, the front-to-back location of the coordinate system. Since some paths for the end effectors are specified relative to the base's coordinates, this parameter allows the synthesizer to alter task properties such as the location of the neutral position of the manipulators and

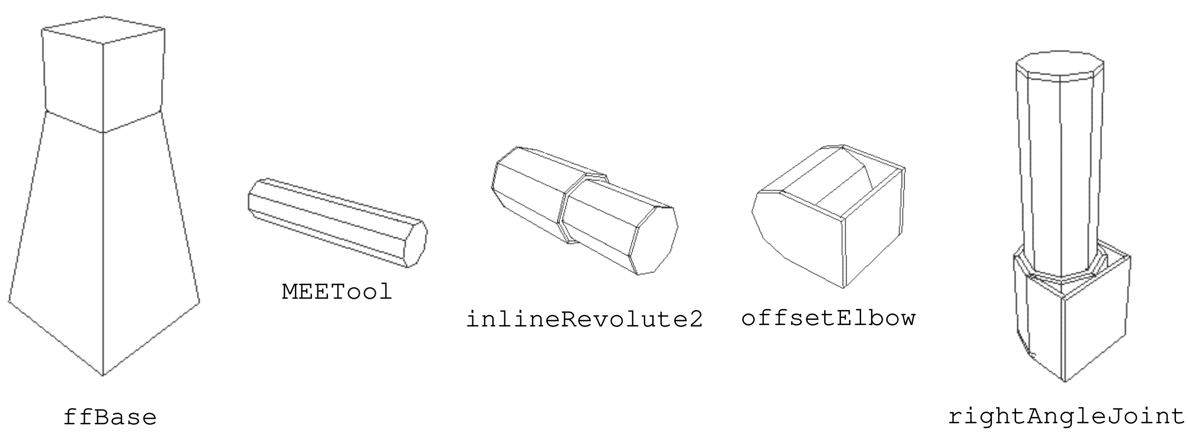


Figure 5.3: Modules for the free-flyer task
The `ffBase` and `MEETool` are task-specific modules for the free-flyer task; the others are general-purpose joint modules. The `inlineRevolute2` and `rightAngleJoint` have parameters for overall length, so no link modules are necessary.

Module	Parameter	Range	# bits	total # module combinations
`ffBase`	origin location	-0.5m to 1.0m	4	16
`MEETool`	(none)	n/a	n/a	1
`inlineRevolute2`	motor selection gearbox selection material selection diameter wall thickness overall length	from list “motors” from list “gearboxes” aluminum (const) 10cm to 20cm 3mm to 1cm 5cm to 1m	n/a n/a n/a 3 3 4	135,168 (10 bits for parameters x 132 component combinations)
`rightAngleJoint`	motor selection gearbox selection material selection diameter wall thickness overall length	from list “motors” from list “gearboxes” aluminum (const) 10cm to 20cm 3mm to 1cm 5cm to 1m	n/a n/a n/a 3 3 4	135,168 (10 bits for parameters x 132 component combinations)
`offsetElbow`	motor selection gearbox selection material selection initial joint angle wall thickness bracket clearance	from list “motors” from list “gearboxes” aluminum (const) (not used) 5mm to 3cm	n/a n/a n/a n/a n/a 2	528 (2 bits for parameters x 132 component combinations)

Table 5.3: Parameter ranges for free-flyer modules

This table shows the range and number of bits for the modules in the module database. The three joint modules (`inlineRevolute2`, `rightAngleJoint`, and `offsetElbow`) have selection parameters for motors and gearboxes; the components for these parameters are listed in Table 5.4 and Table 5.5, respectively. The ‘total # module combinations’ column gives the number of different parameter settings for each module.

the desired distance between the robot base and satellite during manipulation tasks.

In addition to the `ffBase` and `MEETool` modules, three of Darwin2K’s general-purpose joint modules are included: the `inlineRevolute2`, the `offsetElbow`, and the `rightAngleJoint`. The `inlineRevolute2` and `rightAngleJoint` each include a parameter for overall length, so no link modules will be needed for this task. Table 5.3 shows the range and number of bits for the parameters of each module in the module database, and the components for the modules are listed in Tables 5.4 and 5.5. While a prismatic joint could also be included, revolute joints are typically preferred for space applications as they present fewer difficulties for cabling and environmental protection.

As mentioned earlier, we want the robot to have two symmetric arms, and we also want an `MEETool` module at the end of each arm. The kernel configuration shown in Figure 5.4 encodes these preferences so that any robot generated by the synthesizer will have two symmetric arms with `MEETools` at their ends. The kernel consists of an `ffBase` module with each connector attached (via a `const` connection) to a single

Table 5.4: List of motors

Maxon RE25.118755
Maxon 2260.815
Maxon 2260.889
Maxon RE35.118778
Maxon RE36.118800
Maxon RE75.118825

Table 5.5: List of gearheads

Maxon 62.110502	HD Systems CSF-32-50
Maxon 62.110504	HD Systems CSF-32-80
Maxon 62.110506	HD Systems CSF-32-120
Maxon 62.110508	HD Systems CSF-32-160
Maxon 81.110410	HD Systems CSF-40-50
Maxon 81.110411	HD Systems CSF-40-80
Maxon 81.110412	HD Systems CSF-40-120
Maxon 81.110413	HD Systems CSF-40-160
HD Systems CSF-20-50	HD Systems CSF-45-120
HD Systems CSF-20-80	HD Systems CSF-45-160
HD Systems CSF-20-120	HD Systems CSF-50-120
HD Systems CSF-20-160	HD Systems CSF-50-160
HD Systems CSF-25-50	HD Systems CSF-58-120
HD Systems CSF-25-80	HD Systems CSF-58-160
HD Systems CSF-25-120	HD Systems CSF-65-120
HD Systems CSF-25-160	HD Systems CSF-65-160

`inlineRevolute2` module, which is in turn connected to an `MEETool`. Since the `inlineRevolute2` is referenced twice by the `ffBase`, it and the `MEETool` are duplicated thus producing symmetric arms. The `inlineRevolute2` was specified for the first link of the robot's arms because, during initial simulations with manually-generated robots, it was much less prone to self-collision than the other two joint modules and because it allows for a spherical shoulder joint. The two `const` connections to the `inlineRevolute2` module are necessary to enforce symmetry -- if the connections were `var`, they could be altered by the synthesizer thus resulting in asymmetric arms. We also specify some configuration filters to eliminate configurations with undesirable properties: To ensure that the manipulators are kinematically redundant to enhance obstacle avoidance capabilities, we will use the `dofFilter` to cull any configurations with less than 20 or more than 22 total degrees of freedom including the base's 6 unactuated degrees of freedom (i.e. only 7- or 8-DOF arms will be allowed). We will also use the `moduleRedundancyFilter` to eliminate configurations with certain module topologies. For example, attaching two `inlineRevolute2` joints to each other does not provide any extra degrees of freedom since the axes of both joints are colinear. This is also the case for connecting two `rightAngleJoints` via connector 1 on each of them (at the center of the circular surface at the top of the `rightAngleJoint` in Figure 5.3) and for connecting a

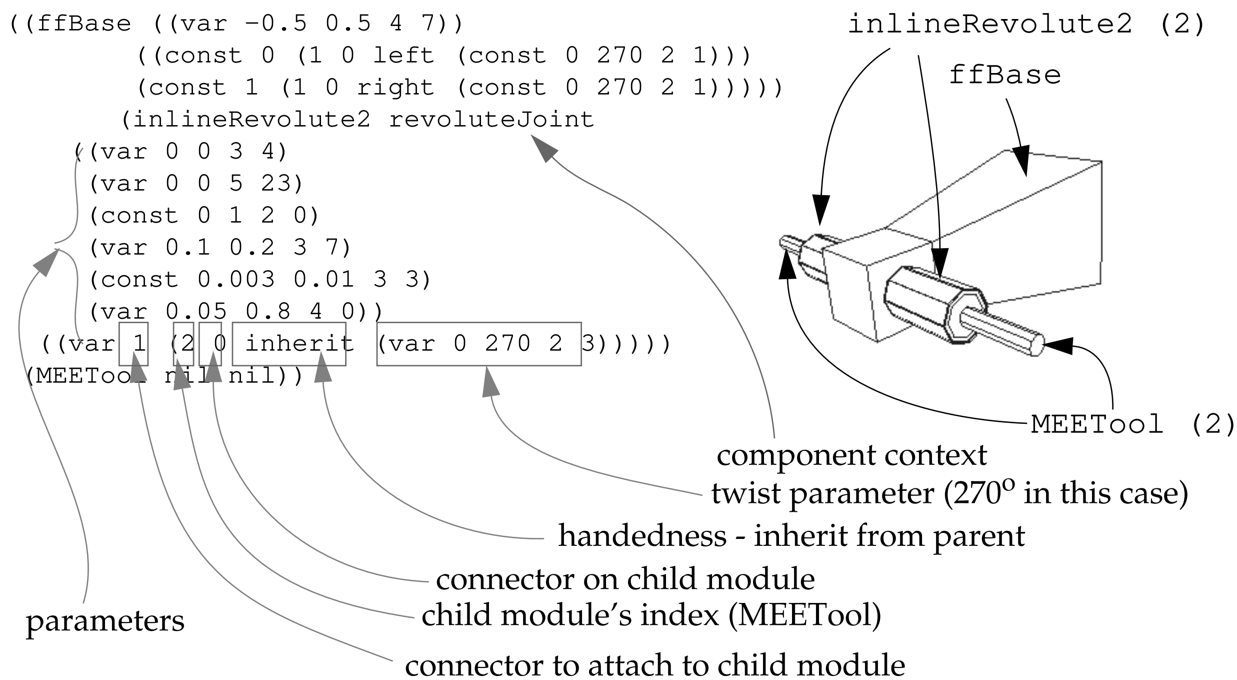


Figure 5.4: Kernel configuration for free-flyer

The `ffBase` has two `const` connections to the `inlineRevolute2`, resulting in two symmetric arms each consisting of an `inlineRevolute2` and `MEETool`. All configurations generated from the kernel will thus have two symmetric arms beginning and ending in `MEETools`. (See Section 2.1.1 and Figure 2.3 for a description of the text format)

`rightAngleJoint` via connector 1 to an `inlineRevolute2`.[2] These filters will prevent configurations with known undesirable properties from being evaluated, thus reducing synthesis time. Given the set of modules, the kernel configuration, the filters, and the task parameters, the size of the search space is approximately 3×10^{58}; that is, there are 3×10^{58} unique configurations (including variation in task parameters) that can be made from the given modules and kernel which meet the constraints imposed by the configuration filters.

We have now described everything necessary for the specification and evaluation of robots for the task; the only thing that remains to be done before starting synthesis is to give values for the synthesizer's parameters, such as probabilities of application for each genetic operator, population size, and any configuration filters. We will use a population size of 200, with an initial population of 5000. The rates for commonality-preserving, module, and parameter crossover are 0.3, 0.2, and 0.1, respectively; the base mutation rate for parameter mutation is 0.02, and is 0.01 for all other mutation operators. We will use adaptive mutation, with a time constant of 1000 (5 times the population size). These

2. These settings for the `moduleRedundancyFilter` are reasonable defaults given the module database and can readily be used for other synthesis tasks that use the same joint modules.

values were chosen based on previous synthesis runs on other tasks; I do not believe they are very task dependent, and are reasonable default values. In general, the synthesizer responds gracefully to changes in parameter settings; a difference of 25 or even 50% in most parameters will normally cause a change in the rate of improvement but will not drastically affect the outcome of the synthesis process. In later experiments we will investigate the relative importance of module and commonality-preserving crossover, but detailed investigation of other parameter settings is not warranted as reasonable default values are known and the system is not particularly sensitive to changes in them.

5.1.5 Synthesis results

Approximately 30 workstations (SGI R5000 and R10000 machines) were used to run Darwin2K for the free-flyer synthesis problem. Typically, 20 to 25 of these were available and used by Darwin2K. A time limit of 15 hours was used, and during that time the ESE generated 60,153 configurations, 46,564 of which were evaluated in simulation with the remainder discarded by the filters. The process began with the generation of an initial population of 5000 configurations from the kernel configuration. After evaluating these initial configurations, the population was culled to 200 configurations. Subsequently, approximately 100 configurations were evaluated before generating the first configuration that satisfied the first requirement group (path completion, number of collisions, and position error). The number of feasible configurations continued to increase before stabilizing at approximately 140 configurations; Figure 5.5 shows the size of the population and number of feasible and optimal configurations over the course of the synthesis run, including the initial 5000 evaluations. Given that the first feasible configuration was generated after 5,100 evaluations and the minimum number of evaluations between generating a feasible configuration and advancing to the next requirement group was set to 4000, the synthesizer advanced to the second requirement group after a total of 9,100 evaluations. After re-seeding the population from the feasible configurations, the metrics for the second requirement group (dynamic path completion, link deflection, and actuator saturation) were added. Actuator saturation is often the most difficult requirement to satisfy; this observation is reflected here by the fact that after 10,600 only a single feasible configuration was generated. During optimization of the second requirement group, at most 28% of the population had acceptable actuator saturation, compared with 53% having acceptable dynamic path completion and 77% having acceptable link deflection. The maximum limit was reached at 20,600 (10,000 evaluations after generating the first feasible configuration; see Section 3.4.2 for details); at this point the synthesizer essentially gives up on trying to generate more feasible configurations and moves on to the next group. Again, the population was re-seeded from the feasible set -- a single configuration in this case -- before continuing with the optimization. The benefit of re-seeding from a feasible configuration is quite evident in Figure 5.5: after re-seeding, numerous feasible configurations were quickly produced, and the synthesizer repeatedly increased the population size to provide room for the ever-growing set of feasibly optimal solutions. Figure 5.6 shows the best mass, energy, and task completion time of feasible configurations (that is, the extreme values in the feasibly-optimal set) during optimization of the final requirement group, while Figure 5.7 shows the feasible configurations from the final population

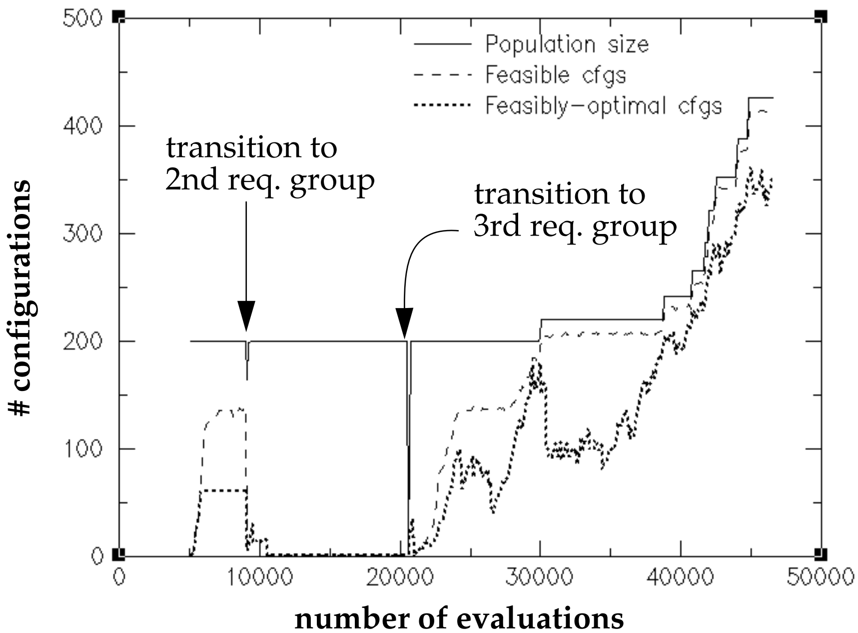


Figure 5.5: Population size vs. number of evaluations for free-flyer
The transitions between requirement groups can be seen on this graph as dips in the total population size, since the population was re-seeded at the beginning of each requirement group. The number of feasibly-optimal configurations approached the population size during the final requirement group (at about 30,000 configurations), so population size was steadily increased to match the growth of the feasibly-optimal set.

---

with the lowest mass, energy, and time (cfg_m, cfg_e, and cfg_t, respectively). (Note that the high mass of the robots -- over 400kg -- is due to the ffBase, which weighs approximately 260kg.) The robots can be seen to have properties that are common among manually-designed robots: their shoulder joints have three degrees of freedom whose rotation axes intersect at a point, and the lengths of the upper arm and forearm are similar so that workspace is maximized. The final degree-of-freedom in each robot's wrist rotates the tool about its axis; this is well-suited for the task, which requires exactly this rotation when removing the ORUs from their mounting points. cfg_e and cfg_t have similar manipulator topologies, with 3-DOF shoulders, 2-DOF elbows, and 3-DOF wrists. cfg_e has low power consumption primarily due to the fact that it moves very slowly between base poses. Because of its initial pose and the location of the `ffBase's` origin, cfg_e does not move much during the first base repositioning. The second base repositioning is a significantly longer motion, and for this motion (`basePath2`) the relevant task parameter specifies a velocity of 0.1m/s--the minimum value for the parameter. In contrast, cfg_t moves at a speed of 0.27m/s between the second and third base poses, thus saving significant time at the expense of higher energy consumption. cfg_m is less massive than the other configurations: it only has one degree of freedom instead of two at its elbow, but its topology is otherwise identical to that of cfg_e and only a few parameter values differ. The lack of the extra actuator at the elbow makes this configuration lighter, though its energy consump-

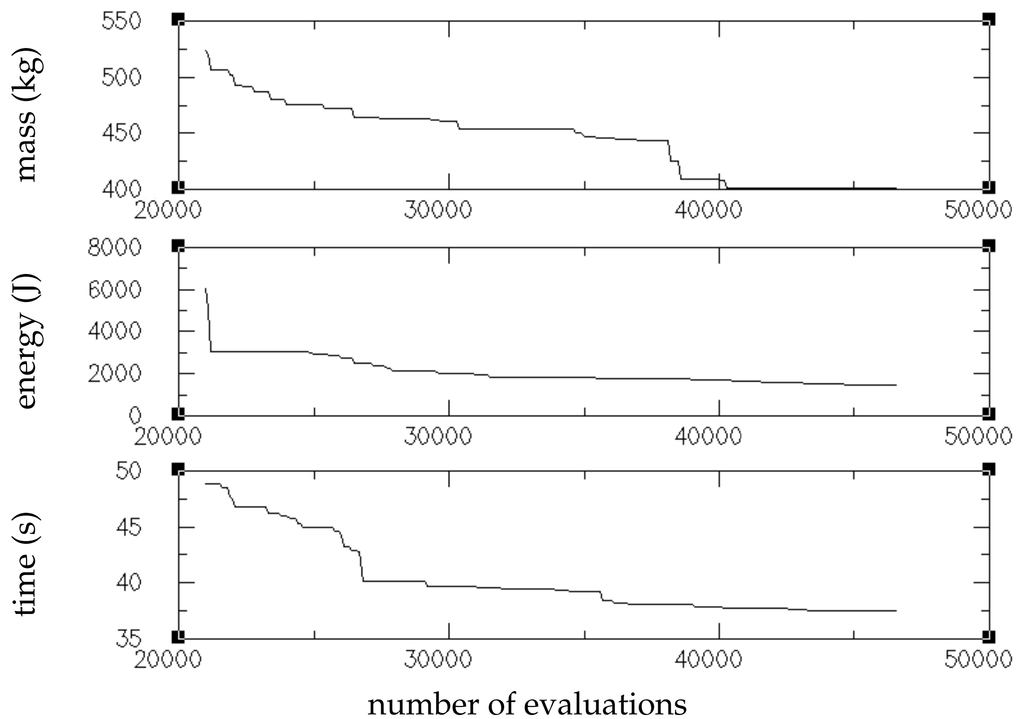
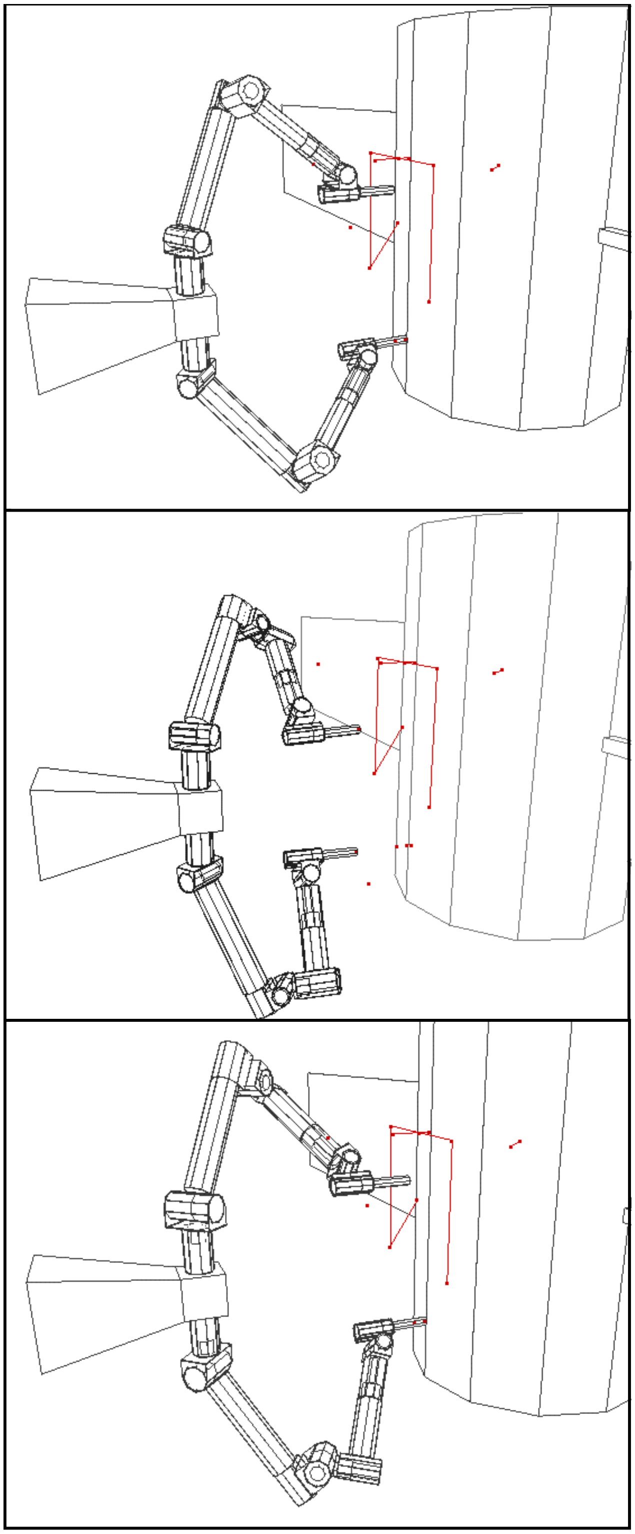


Figure 5.6: Optimization of mass, energy, and time for free-flyer
This graph shows (from top to bottom) mass, energy, and time for the best feasible configuration during optimization of the last requirement group.

tion is significantly higher than that of either cfg_e and cfg_t. Its task parameters specify a velocity of 0.16m/s during the second base motion causing inefficiency and saturation from many of its actuators, which are significantly less powerful than those of cfg_t. cfg_m has a continuous actuator saturation of 0.62, significantly higher than cfg_e's saturation of 0.38 and cfg_t's 0.41. These numbers indicate that cfg_m's actuators are not as over-designed as those of the other two configurations. Figure 5.8 shows cfg_m executing the task in simulation; see Appendix C.1 for a detailed description of this configuration.

These three configurations are the ones with best (lowest) mass, energy, and time; however, there are 347 other feasible Pareto-optimal configurations that make trade-offs between these extremes. If we can accept a slight degradation in one metric from one of the best configurations, significant improvements can be had in the other two metrics (e.g. a slightly heavier robot than cfg_m may have significantly lower task completion time or energy consumption). Figure 5.9 shows scatter plots for the feasibly-optimal configurations that have a 5% or less degradation in performance in time, energy, and mass when compared to cfg_t, cfg_e, and cfg_m, respectively, e.g. every data point in Figure 5.9a shows the energy and time for a configuration with mass less than 421kg (5% more than cfg_m). For example, the configuration closest to the origin in Figure 5.9a has a mass of 416kg, power consumption of 2087J, and task completion time of 42.3s -- a 3.7% increase in mass yields a 70% decrease in energy and a 7.5% decrease in completion time. However, trade-offs against energy are not as useful -- there are only two feasible configurations with energy within 5% of cfg_e, and neither offer much improvement in mass or time. There are



Configuration cfg_m

- 58,952nd configuration
- mass: 401 kg
- energy: 7.1 kJ
- time: 45.9 s
- 7-DOF arms; 3-DOF wrist and shoulder, 1-DOF elbow
- having only 7 DOF leads to reduced mass
- actuator saturation: 0.62

Configuration cfg_e

- 59,861st configuration
- mass: 421 kg
- energy: 1.5 kJ
- time: 49.9 s
- 8-DOF arms; 3-DOF wrist and shoulder, 2-DOF elbow
- minimal base motion and low velocity during base motions lead to reduced energy
- actuator saturation: 0.38

Configuration cfg_t

- 59,172nd configuration
- mass: 473 kg
- energy: 4.3 kJ
- time: 37.5 s
- 8-DOF arms; 3-DOF wrist and shoulder, 2-DOF elbow
- large actuators and fast base motion reduce time but increase mass and energy
- same topology as cfg_e
- actuator saturation: 0.41

Figure 5.7: Best feasible free-flyers

From top to bottom, the free-flyer configurations with best mass, energy, and task completion time, respectively. It is interesting to note that all three configuration have upper- and fore-arms of similar length as well as 3-DOF shoulders and elbows -- features commonly found in manually-designed manipulators.

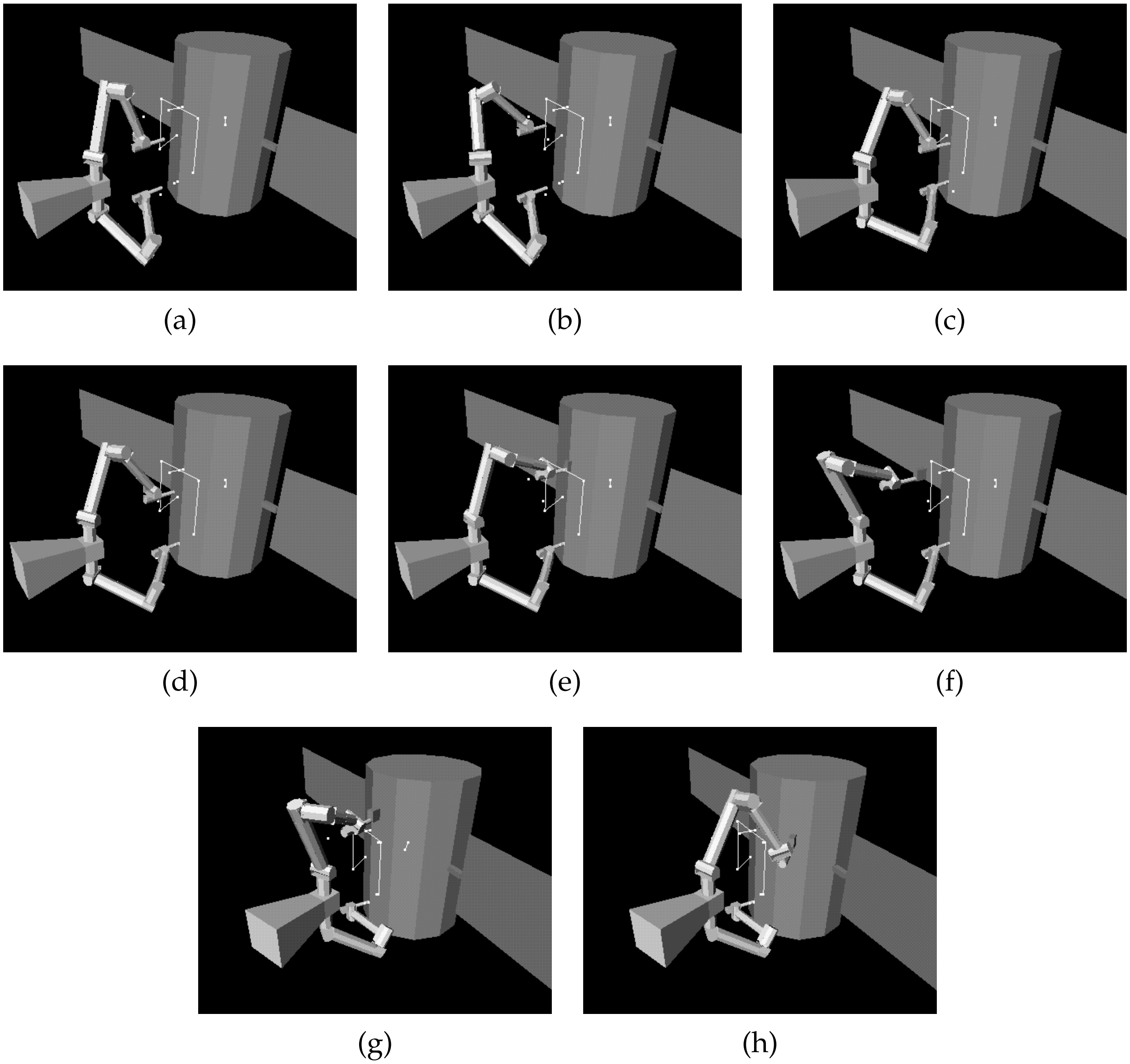
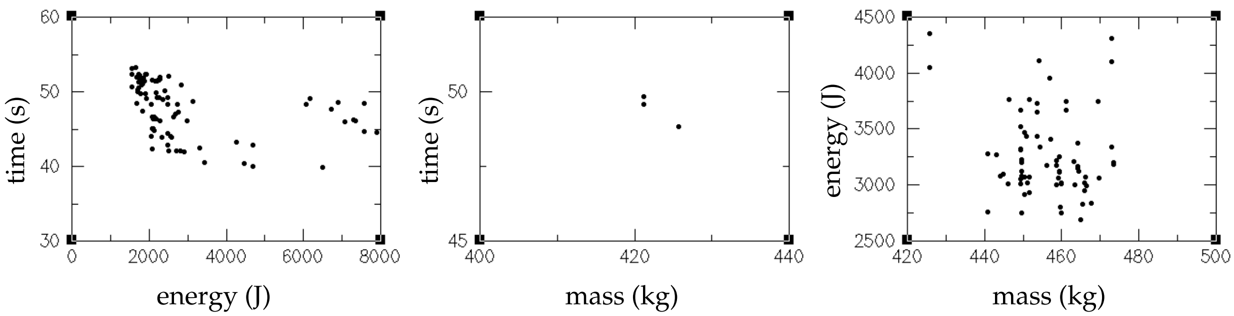


Figure 5.8: Sequence showing free-flyer performing task

This sequence of images depicts a free-flyer synthesized by Darwin2K as it performs the ORU replacement task:

(a) end effectors are moved to their neutral positions
(b) the work manipulator traces a path (white lines) in kinematic simulation
(c) the grapple manipulator approaches the grasp point on the satellite
(d) the robot repositions itself relative to the satellite using the grapple manipulator
(e) the work manipulator removes the ORU
(f) the ORU is moved away from the satellite to avoid collisions
(g) the grapple manipulator again repositions the base
(h) the work manipulator re-inserts the ORU

A detailed description of this configuration is given in Appendix C.1.



(a) energy vs. time (mass < 421kg)

(b) mass vs. time (energy < 1529J)

(c) mass vs. energy (time < 39.4s)

Figure 5.9: Scatter-plots of feasibly optimal configurations
These three figures show the feasible configurations with performance degradations of five percent or less compared to the optimal configurations with best mass, energy, and time, respectively.

numerous trade-off configurations with respect to cfg_t; one has a mass of 441kg (6.7% decrease), energy consumption of 2759J (35% decrease), and completion time of 39.1s (4.2% increase). In general, it will be up to the designer to decide which trade-off is best in terms of the task's requirements; in this case, it may make sense to choose a configuration with low mass and reasonable energy and time because of launch costs. After looking at some of the optimal configurations, the designer might also decide to manually modify one of them to see if further improvement is possible--in this case, perhaps the designer would change cfg_m's task parameter for base motion velocity to see if lower power consumption is possible. The synthesizer does not attempt to extract or exploit any causal relations between robot properties and performance; doing so would reduce its independence of task and might introduce biases in the way configurations are generated which would decrease its ability to explore the design space. However, it is easy for the designer to set many of the parameters and connections in a well-optimized configuration to `const` and then run the synthesizer starting from this configuration so that a focused parametric optimization can be performed.

5.1.6 Summary and discussion

In this experiment, we applied Darwin2K to a complex synthesis problem. The synthesized robots were able to complete the task while meeting several performance constraints, and exhibited some properties that are common in manually-designed robots as well as other properties that are particularly well-suited for the task. After generating the first feasible configurations, the synthesizer was able to produce significant performance improvements in the population and generate configurations spanning a range of trade-offs between objective functions.

This synthesis example demonstrates several new capabilities in automated configuration synthesis: the use of dynamic simulation for evaluation, synthesis of link structural properties, synthesis of motor and gearbox selection for a non-modular robot, and creation of a range of designs which meet multiple performance constraints while providing different trade-offs between several objective functions. These capabilities represent significant advances over previous synthesis systems, which were restricted to kinematic simulation and either performed purely-kinematic or purely-modular synthesis.

The reader should now have a good understanding of the process of specifying a task for Darwin2K: choosing metrics, modules, and evaluation components; coding task-specific simulation methods; and specifying a kernel configuration. However, there may be some remaining questions about the impact of these choices: How does the selection of modules in the module database affect the synthesis results? What effect does changing the selection probabilities have? How repeatable are the synthesis results? These questions and others are important in characterizing the robustness and limitations of Darwin2K, and will be addressed in the other experiments presented in this chapter.

5.2 Task 2: A fixed-base manipulator

An example of automated task-based kinematic synthesis from the literature is the design of a manipulator for waterproofing the tiles on the underside of the Space Shuttle [Kim93], based on the requirements of the Tesselator project [Dowling92]. The underside of the Space Shuttle is tessellated into a number of rectangular regions that will be serviced sequentially by a manipulator mounted on a mobile base. Each region measures 3m across its diagonal (2.14m on a side) and contains approximately 180 tiles, each of which must be individually waterproofed by the robot's manipulator after the mobile base is repositioned beneath the region. Using Darwin2K to synthesize manipulators for this task will allow us to compare its results to those of Kim's system (which also only addressed synthesis of the robot's manipulator), and the moderate complexity of this task makes it useful for examining the impact of different starting conditions such as modules and kernel configuration on synthesis results. The synthesis problem for this task is complex enough to observe interesting behavior in the synthesizer, yet simple enough to make repeated runs under different conditions feasible.

Simulating the robot as it moves to each tile in a region, and over all regions on the Space Shuttle, is currently too computationally expensive for use in an automated synthesis method. For this reason, both Kim's work and this experiment simulate the robot over a single representative region that contains the range of heights and orientations that will be encountered on the underside of the Shuttle. The representative region is 2.14m on a side, and ranges from a horizontal surface 3m above the ground to a 45 degree surface 4m above the ground (Figure 5.10) The number of tiles reached by the manipulator during simulation is also reduced from 180 to 7 (in Kim's work) and 16 (in this experiment) tiles which span the representative region so that the entire necessary workspace is covered. The waterproofing mechanism at the robot's end effector is modeled as a 1.5kg payload, and to ensure accurate placement of the payload a position error tolerance of 2mm is used at each via point in the trajectory -- that is, the end effector must stop within 2mm of each via point before moving to the next one.

Many of the metrics used for this synthesis problem (listed in Table 5.6) were also

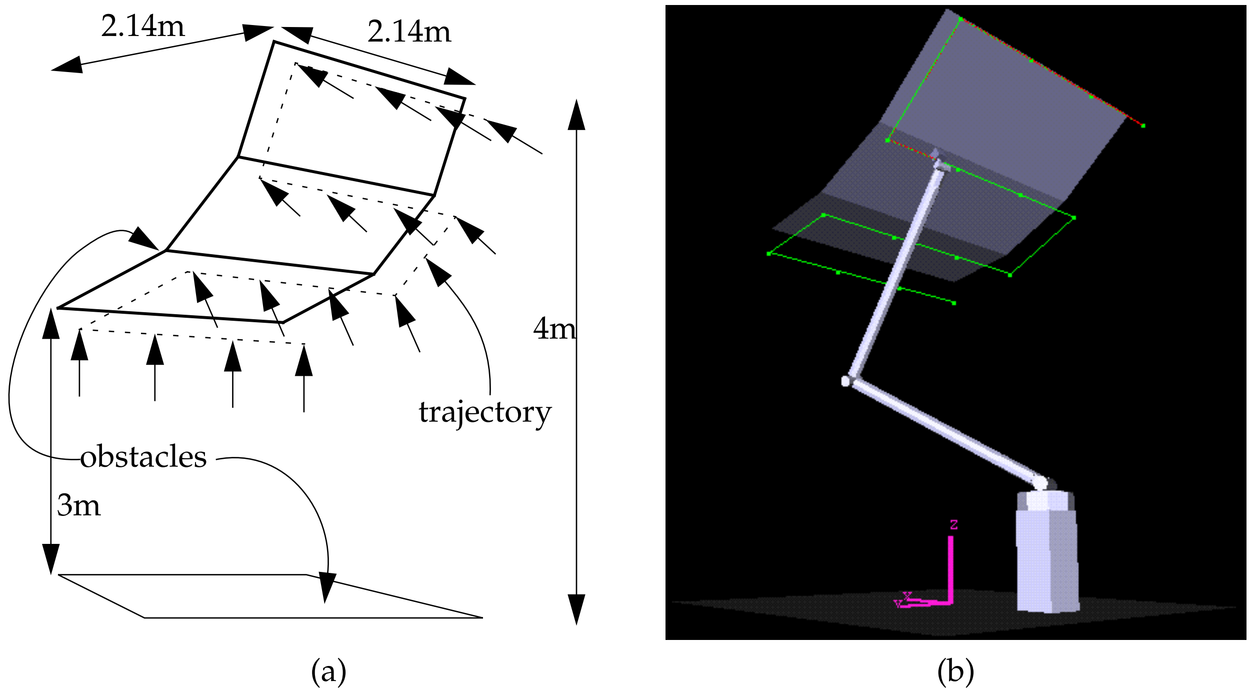


Figure 5.10: Trajectory and obstacles for Space Shuttle waterproofing manipulator

(a) shows a schematic view of the obstacles and trajectory with normals denoting orientation at each via point. (b) shows a simulation view of a synthesized manipulator executing the task.

Metric name	Acceptance threshold	Min	Max	Scale	AFDI
pathCompletionMetric	= 1.0 (100%)	0	1	4	0.17 (17%)
collisionMetric	integral = 0	0	10	1	0.693
positionErrorMetric	max < 0.03m	0.005m	0.3m	200	3.4mm
linkDeflectionMetric	max < 0.002m	0.0019m	0.01m	1000	0.7mm
actuatorSaturationMetric	max < 1.0 (100%)	0.3	50	0.2	3.4
massMetric	none	0 kg	100kg	0.1	6.9kg
timeMetric	none	18s	60s	0.5	1.4s

Table 5.6: Metrics for Space Shuttle waterproofing manipulator

The scale values for this task were set before the AFDI was formulated, resulting in less-intuitive values.

used in the free-flyer, though their ranges, scales, and acceptability thresholds are different here. The first requirement group is composed of the `pathCompletionMetric`, `collisionMetric`, and `positionErrorMetric`, which together ensure that the en-

	1-DOF revolute joints	`prismatic-Tube`	`scaraElbow`	# kernels	kernels
Baseline	●			1	- simple
Prismatic	●	●		1	- simple
SCARA	●	●	●	1	- simple
Const	●			1	- first joint axis vertical; final 3 axes intersect
High-Level	●	●	●	3	- simple - 2-DOF shoulder - 3-DOF wrist

Table 5.7: Modules and kernels for manipulator experiments

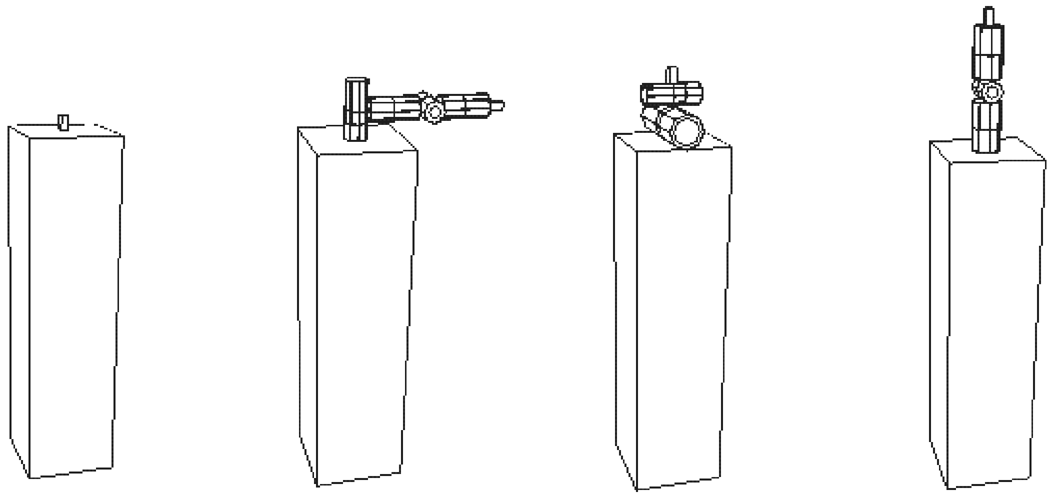
The simple kernel (Figure 5.11a) was used in all experiments except the Const trials, which duplicated the constraints in Kim's experiment (Figure 5.11b). The High-Level trials used two additional kernels, one with a 2-DOF shoulder and the other with a 3-DOF wrist (Figures 5.11a and 5.11b).

---

tire trajectory is accurately followed with no collisions. The second requirement group contains the `linkDeflectionMetric` and `actuatorSaturationMetric` so that link structure and joint actuators may be appropriately sized, and finally the third requirement group consists of the `massMetric` and `timeMetric` so that an efficient, lightweight robot can be created.

Since the manipulator's base is not free-flying and we are requiring actuator saturation to be less than one, we can use kinematic rather than dynamic simulation for this task. (On the other hand, if we allowed actuator saturation we would have to use dynamic simulation to determine if saturation prevented a robot from completing the task.) No task-specific modules or components are needed for this task; we can use the `pathEvaluator` for overall simulation control, the `SRIController` to generate joint commands that follow the trajectory (represented by a `path` object), the `collisionDetector` to monitor collisions as needed by the `collisionMetric`, the `rungeKutta4` method to provide numerical integration of robot state, and a `payload` to model the waterproofing payload. We will include the velocity, acceleration, angular velocity, and angular acceleration variables of the `path` as task parameters so that they may be optimized to improve performance.

To investigate the impact of module and kernel choice on the synthesizer's performance, five different sets of experiments (with five runs each) were performed: Baseline, Prismatic, SCARA, Const, and High-Level. The same task and simulator were used for all five sets of experiments; the differences were in the modules contained in the module database, and in the topologies supplied in the kernel configuration. A unique initial population was generated for each run of the synthesizer by using different seed values to initialize the random number generator. Table 5.7 summarizes the different module and kernel properties for each of the experiments, while Figure 5.11 depicts the kernel configurations used in each of the experiments. The Baseline, Prismatic, and SCARA experi-



(a) (b) (c) (d)

Figure 5.11: Kernels used in the manipulator experiments

(a) shows the simple kernel used in all experiments except for Const, which used (b) instead. The High-Level experiment also included one kernel with a shoulder assembly (c) and another with a wrist assembly (d).

ments all used a single simple kernel consisting of a `simpleTool` mounted directly on a `fixedBase`. The Const experiment used a kernel duplicating the constraints used in Kim's experiment: the first joint axis was vertical, and the last three (wrist) axes intersected in a point. These constraints were enforced by setting the `const` flags for the attachments between the base and first joint, and between the last three joints. The High-Level experiment used the same simple kernel as in the Baseline, Prismatic, and SCARA experiments and also included two other kernels: one had two `elbowJoints` forming a shoulder, and the other had a wrist assembly identical to that in the Const experiment. While the joint modules in these kernels were attached with `const` attachments, the fact that there were multiple kernels (including the simple kernel) in the High-Level experiment meant that these features were optional and could be combined or eliminated when creating configurations.

The `dofFilter` was used to limit the number of degrees of freedom for the Const experiment to 7 (as was used in Kim's experiment), and between 5 and 7 for the other experiments. The `moduleRedundancyFilter` had not yet been implemented at the time these experiments were run, so there was no filtering of redundantly-connected modules. The terminating conditions for each experiment were a maximum run time of 6 hours and a minimum of 80,000 configurations; between 80,000 (for most of the High-Level experiments) and 140,000 configurations (for the unsuccessful runs in other experiments) were evaluated during each run.

A summary of the best time and mass of any feasible configuration synthesized during each five-run experiment is shown in Figure 5.12. The High-Level experiments consistently produced the best results -- the average of the best mass and time over the five trials is significantly lower than for the other experiments, and the standard deviations of mass and time are lower than for the other experiments indicating more consistent performance. A rough ordering of the other experiments in terms of the average of

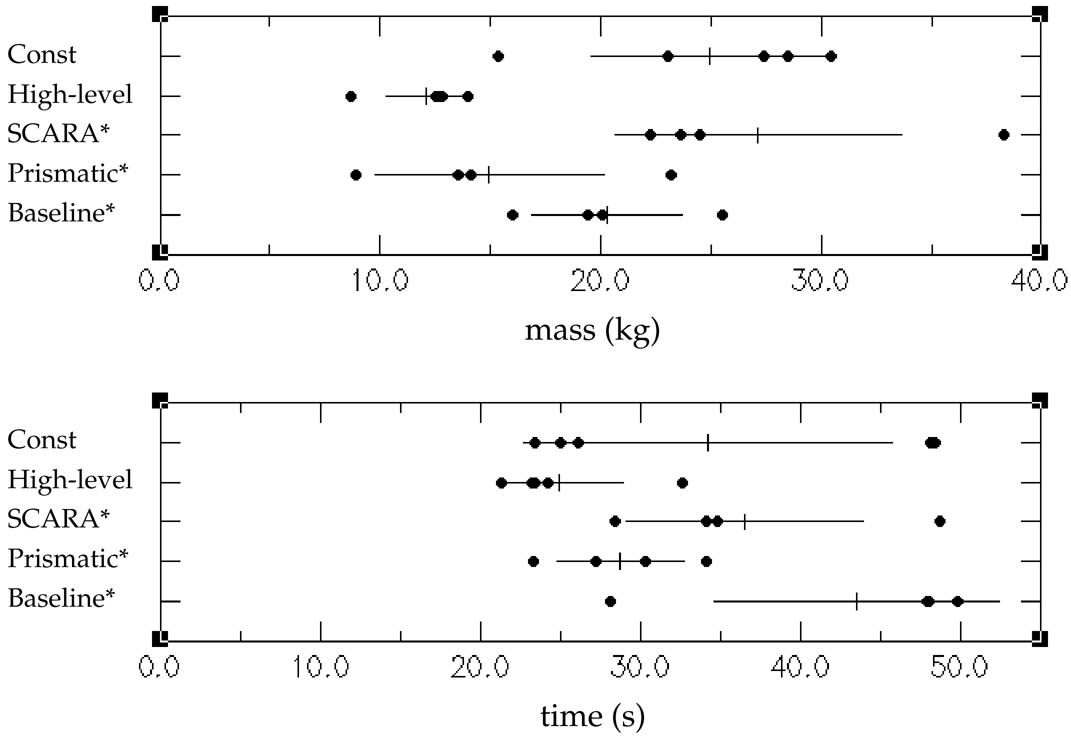
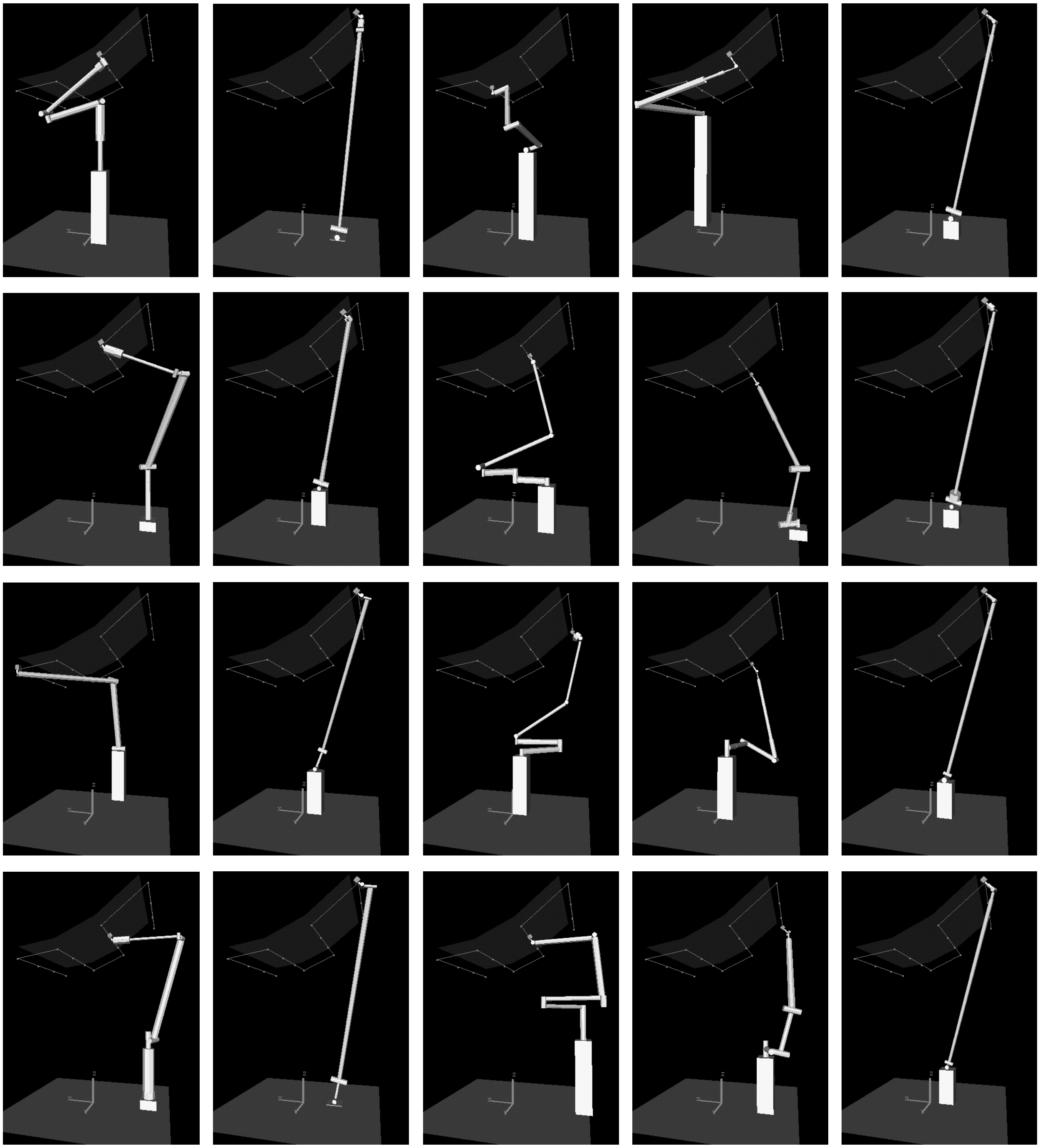


Figure 5.12: Mass and time for varying starting conditions

In the upper graph, each dot indicates the mass of the lightest configuration generated for each successful run of the five different sets of experiments. The vertical line shows the mean mass over the successful experiments, and the horizontal line indicates +/- one standard deviation. The lower graph is similar, but for task completion time. The experiments with an asterisk next to their label (SCARA, Prismatic, and Baseline) only had four successful runs out of five; one of each of their runs failed to produce any feasible configurations.

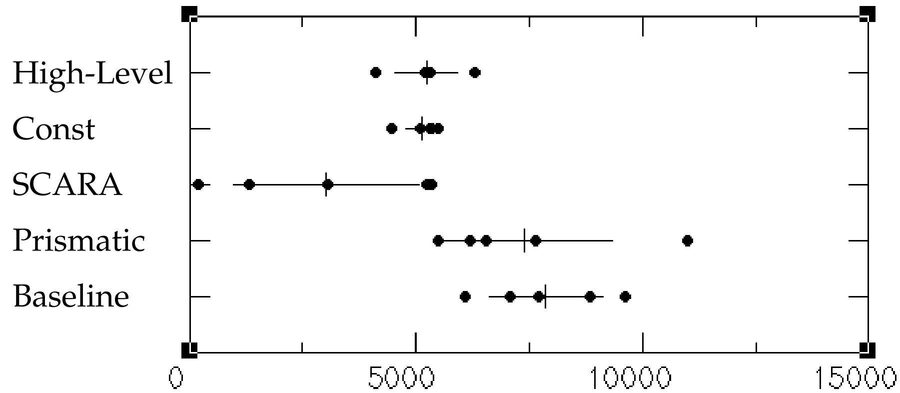
best mass and time over the successful trials is Prismatic, Const, Baseline, and finally SCARA. As we will see, these results can be understood by considering the how the primitives available for each experiment influence the synthesizer's ability to find feasible configurations, and how well-optimized the first feasible configurations are.

Figure 5.13 shows a number of robots synthesized in the five sets of experiments. The topology of the High-Level and Prismatic results are generally similar: the basic form is a 2-DOF shoulder followed by a prismatic joint and ending in a 2- or 3-DOF wrist. In terms of mass, the solutions generated in the High-Level experiments are superior to all other experiments, with the Prismatic results close behind. (See Appendix C.2 for a detailed description of one of the High-Level robots.) Based on the similarity of the topologies from the Prismatic and High-Level experiments, one can surmise that the use of a prismatic joint is necessary for creating lightweight manipulators for this task. The results for task completion time are similar though not as one-sided, primarily because task completion time is heavily influenced by the task parameters for maximum acceleration and velocity along the path. Task completion time is thus less dependent on topology (and therefore on available modules) than on task parameters, though a configuration's actuators must be able to supply the joint velocities necessary to follow the trajectory at the

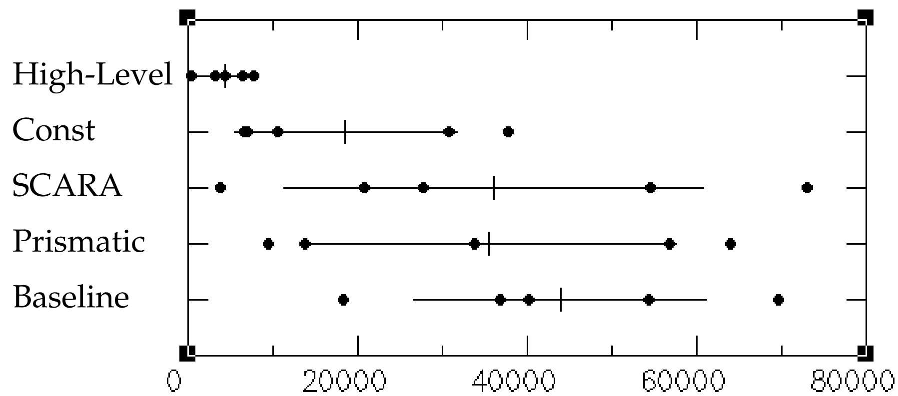


Baseline **Prismatic** **SCARA** **Const** **High-level**

Figure 5.13: Best manipulators for varying synthesizer starting conditions
Each column above shows robots from the final optimal sets of repeated trials of each experiment. The top two images in each column show the lightest robots from two different trials; the bottom two show the fastest robots. Overall, the High-Level and Prismatic experiments generated the best-optimized robots, and also showed less variation between trials than the other experiments in terms of robot topology and performance. See Appendix C.2 for a description of the robot in the upper right corner.



(a) number of evaluations - 1st requirement group



(b) number of evaluations - 2nd requirement group

Figure 5.14: Number of evaluations required to generate first feasible configuration

(a) shows the number of evaluations required in each experiment to generate the first configuration meeting the feasibility criteria for the first requirement group. (b) shows the number of evaluations required to create a feasible configuration after advancing to the second requirement group.

---

speed dictated by the task parameters if maximum performance is to be achieved.

The lack of a prismatic joint can be seen as the major reason that the Const and Baseline experiments were not able to produce configurations as light as those in the High-Level and Prismatic experiments. However, the SCARA experiments also included a `prismaticTube` in the module database and yet the best configurations generated in the SCARA experiments were substantially inferior to those in the High-Level and Prismatic experiments. None of the configurations in the SCARA feasibly-optimal sets contain a prismatic joint, which is at first surprising given that the High-Level and Prismatic experiments made effective use of the `prismaticTube` module. The SCARA experiments did make heavy use of the `scaraElbow` module, though: all of the optimal configurations in four of the runs contained two `scaraElbow` modules, and in the remaining experiment all optimal configurations had one `scaraElbow` module. Figure 5.14, which shows the number of evaluations required to synthesize configurations that satisfy the

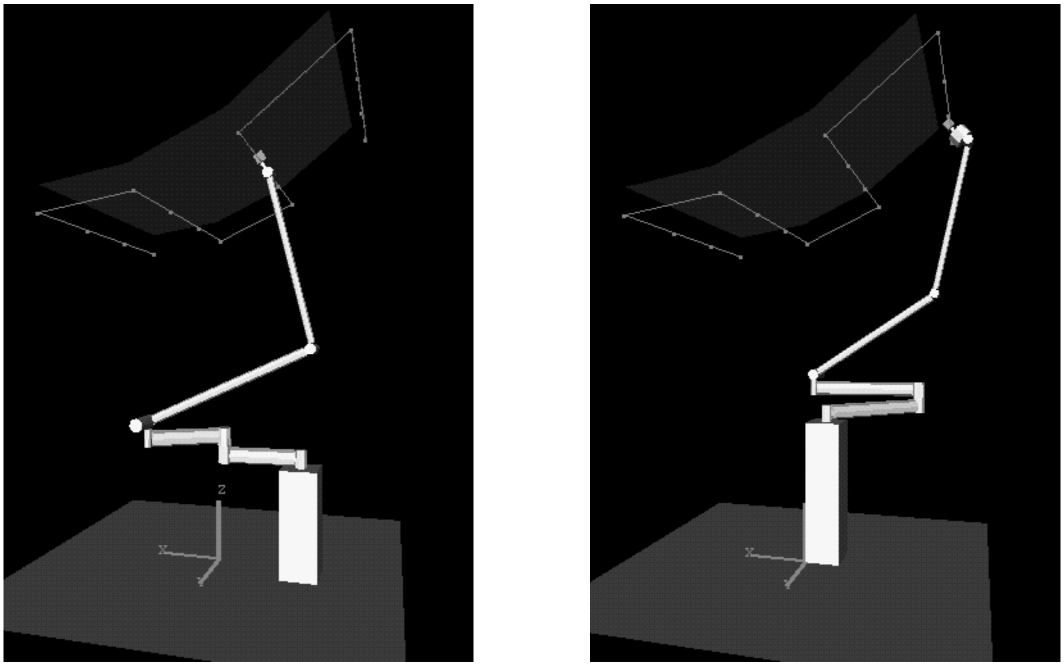


Figure 5.15: Typical configurations from the SCARA experiments
Most of the robots from the SCARA trials consisted of two scaraElbow modules mounted in series, with the first one horizontal and the second vertical. The scaraElbow's high complexity made it easy for the synthesizer to generate feasible configurations, thus leading the synthesizer to focus on these initial topologies rather than explore others that would ultimately lead to better-optimized robots.

first and second requirement groups, provides some insight into how the scaraElbow influences the synthesis process for this task. (Note that for the Baseline, Prismatic, and SCARA experiments, only four out of five runs were successful; for the unsuccessful runs, the number used in the graph is the number of evaluations during optimization of the second requirement group although the second requirement was never satisfied). In the SCARA experiments, the synthesizer was able to quickly generate configurations which satisfied the first set of requirements; the synthesizer quickly focused on these configurations and became trapped in a local minimum. The high complexity (3 joints and two links) of the SCARA module makes it easy for the synthesizer to create feasible configurations: many feasible configurations simply consist of the fixed base, two scaraElbow modules, and the tool (Figure 5.15). When feasible configurations can be created very quickly from complex modules, they can quickly dominate the selection of configurations for reproduction. Any major changes to these configurations -- such as those necessary to create configurations using a prismatic joint as in the High-Level experiments -- will likely result in poorly-performing configurations that have little chance of reproduction and will quickly be culled from the population. It is thus easy for the synthesizer to become trapped in a local minimum when promising designs can be quickly synthesized from complex modules that are not actually useful for well-optimized configurations. On the other hand, when well-suited for a task complex modules with many parameters will likely require less synthesis time compared to simpler modules with few parameters. The designer should thus be willing to use complex modules if there is good reason to believe they will be useful for a particular task.

Complex topological features can likewise improve synthesis speed and quality when appropriate for the task. A closer examination of the best configurations from the High-Level experiments reveals that the subgraphs specified in the kernel configurations were highly useful: the fastest and lightest configurations in every High-Level trial contain both the wrist and shoulder subgraphs, which are easily identified by their `const` attachments. The High-Level experiments were quickly able to generate configurations satisfying the first and second sets of requirements (recall that the third set of requirements had no feasibility thresholds) -- substantially faster than the Prismatic experiments, though the results were similar. The reduced synthesis time and improved performance of the resulting configurations clearly indicate the utility of supplying human knowledge to the synthesis process in the form of useful subgraphs in the kernel configurations.

The Const experiments lie somewhere between the High-Level and SCARA experiments, in terms of both the quality and the phenomena responsible for the results. The Const experiments contained complex primitives (in this case, the kernel configurations) which, like the scaraElbow module, turn out to be detrimental to performance. The Const experiments were constrained to use only revolute joints, have 7 degrees of freedom, and use the vertically-oriented shoulder axis and 3-DOF wrist specified in the kernel configuration. These constraints prevent solutions similar to those in the High-Level and Prismatic experiments from being discovered, though they do seem to be somewhat helpful in that the task completion times of the best configurations are better for the Const experiments than for the Baseline experiments even though both used the same set of modules. The number of evaluations required to generate configurations satisfying the first and second requirement groups is also lower for the Const experiment than for the Baseline experiment, indicating that the subgraphs (wrist and shoulder) in the Const kernel reduced the amount of exploration required by the synthesizer when generating feasible configurations. Similarly, the High-Level experiments required less search than the Prismatic experiments, even though they arrived at similar results.

One factor that was not investigated in these experiments is how synthesis results are affected by the range and resolution of parameters. It seems likely that increasing the number of bits for the parameters would have little affect on convergence time or quality, as the additional bits would have a relatively small impact on robot features. Increasing the range of parameter values (e.g. 0.01-10m instead of 0.01-1m for a link length) would probably have a more significant affect on synthesis quality and convergence, since many parameter settings would lead to poor performance. In general, it is probably best for the designer to set the range for parameter values such that obviously nonsensical parameter values are avoided.

5.2.1 Summary

These experiments demonstrated synthesis of a manipulator's kinematics, dynamics, actuator selection, structural dimensions, and controller parameters using kinematic simulation for evaluation. Repeated trials with a range of starting conditions give some insight into the repeatability of synthesis results and the impact of module and kernel choices on the properties of the synthesized robots. Including human knowledge in the form of specific modules or kernels is a double-edged sword: it can be quite effective if that knowledge is accurate, as in the High-Level experiments, or it can be detrimental if that knowledge is not useful for well-optimized solutions, as in the Const and SCARA experiments. In addition to potentially improving the quality of synthesis results, incorporating human knowledge significantly reduces the time required to synthesize feasible configurations -- the High-Level, Const, and SCARA experiments were significantly faster than the Prismatic and Baseline experiments in synthesizing feasible configurations for the first requirement group, and the High-Level and Const experiments were faster than the others in synthesizing feasible configurations for the second requirement group. Repeatability varied significantly between different experiments and seemed correlated with the optimality of the results: the High-Level and Prismatic experiments generated the best-optimized configurations and had less variation in robot topology and performance than other experiments. The fraction of runs resulting in feasible configurations (22 out of 25, or 88%) was similar to the only other figure reported in the literature, which was 15 out of 20 (75%) in [Paredis96]. Throughout these experiments, the synthesizer's parameters were held constant - the same selection method, genetic operators, and operator probabilities were used in all experiments. The next set of experiments, which use a simplified version of the task addressed in the five experiments we have just seen, investigate the effectiveness of the synthesizer's different selection algorithms and genetic operators.

5.3 Task 3: Simplified manipulation task for characterizing synthesizer performance

Chapter 3 presented several methods for enhancing the synthesizer's performance. Some of these methods, specifically requirement prioritization and configuration decision functions, are ways of selecting configurations for reproduction and deletion; others such as the commonality-preserving crossover operator and subgraph preservation are methods of generating new configurations from those that were selected for reproduction. To assess the impact of these methods, they are applied to the synthesis of a manipulator for a simplified version of the Space Shuttle waterproofing task used in the previous set of experiments. The first set of experiments involves commonality-preserving crossover and subgraph preservation, and involves multiple complete runs of the synthesizer and additional runs on the final requirement group. The second set of experiments compares the performance of three different selection methods: the weighted sum, the Configuration Decision Function, and Requirement Prioritization.

To decrease evaluation time and synthesis complexity, several changes were made to the previous experiment's task description. Instead of 16 via points (tile locations) in the manipulator's trajectory, there are only four; this greatly reduces task execution time (and thus the time required for each evaluation) since the robot does not have to decelerate and stop at as many points as in the previous task. The four points are at the corners of the representative region and the surface representing the Shuttle's underside is now

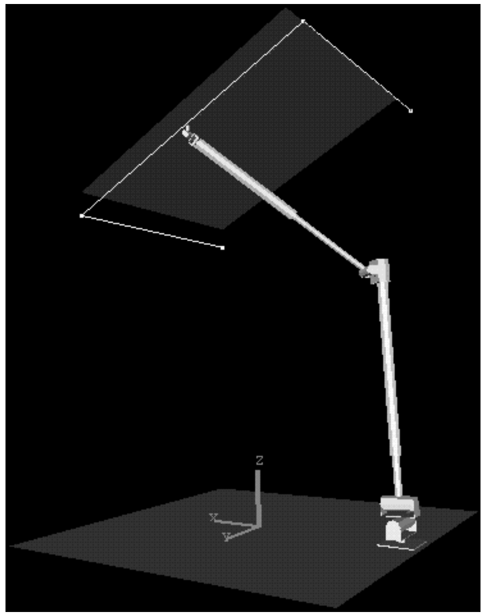


Figure 5.16: Simplified Space Shuttle waterproofing task
The simplified Space Shuttle waterproofing task uses a trajectory with only four via points (rather than 16), and the surface representing the Shuttle's underside is flat since the trajectories between via points are straight lines.

Metric name	Acceptance threshold	Min	Max	Scale	AFDI
pathCompletionMetric	= 1.0 (100%)	0	1	6.93	0.1 (10%)
collisionMetric	integral = 0	0	200	1	
positionErrorMetric	max < 0.03m	0.005m	0.3m	200	
linkDeflectionMetric	max < 0.002m	0.0019m	0.01m	693	1mm
actuatorSaturationMetric	max < 1.0 (100%)	0.7	200	0.69	1 (100%)
massMetric	none	0 kg	100kg	0.139	5kg
timeMetric	none	0s	60s	0.34	2s

Table 5.8: Metrics for simplified Space Shuttle waterproofing task
The shaded values are those that were changed from the previous Space Shuttle waterproofing experiment to give more intuitive AFDI values and to reflect the fact that the previous minimum values for mass and time were not necessary.

flat, since the manipulator will follow a straight-line trajectory between via points (Figure 5.16). Some minor changes were also made to the scale values for the task's metrics (see Table 5.8) since the AFDI interpretation of scale values had not yet been formulated at the time the previous experiment was run. The minimum values for mass and time were changed to zero, since no clipping of mass or time is necessary. The minimum for the `actuatorSaturationMetric` was increased from 0.3 to 0.7, indicating that actuator sat-

	Module crossover	Commonality-Preserving Crossover	Parameter Crossover	Subgraph preservation
Baseline	0.65	0	0.25	no
CPCO	0.4	0.25	0.25	no
SUBP	0.4	0.25	0.25	yes
SUBP-2	0	0.65	0.25	yes

Table 5.9: Crossover rates for CPCO and subgraph preservation experiments

This table shows the rates for each type of crossover operator used in the crossover experiments. The far right column (titled "Subgraph preservation") indicates whether or not the common subgraphs of two feasible parents was fixed (as described in Section 3.4.4) in their offspring. Note that the probabilities for module crossover and CPCO sum to 0.65 in all experiments, and the probability for parameter crossover is 0.25 in all experiments.

uration values less than 0.7 should be considered equivalent by the synthesizer. The simple kernel configuration and module database with 1-DOF revolute joints and `prismaticTube` module duplicate the Prismatic conditions in the previous experiment.

5.3.1 Commonality-preserving crossover and subgraph preservation

In Chapter 3, the Commonality-Preserving Crossover Operator (CPCO) and the concept of subgraph preservation were presented and were expected to have beneficial impact on the synthesis process. To test this assertion, we can compare the performance of the robots generated by the synthesizer for several cases: Baseline (no CPCO or subgraph preservation), CPCO (no subgraph preservation), SUBP (using CPCO and subgraph preservation) and SUBP-2 (as SUBP but with higher frequencies of CPCO and subgraph preservation). For each of the tests the mutation, duplication, and parameter crossover rates were identical, and the CPCO and module crossover probabilities summed to 0.65 as shown in Table 5.9. Requirement Prioritization was used as the selection method for the two groups of experiments: the first group compared the performance of the crossover operators over an entire synthesis run, while the second group was limited to optimizing the final requirement group (mass and time) starting from a set of feasible configurations.

For the first set of experiments, an initial population was randomly generated and then decimated to 100 configurations; this decimated population was used as a common starting point for each of the thirty-six trials (nine each for Baseline, CPCO, SUBP, and SUBP-2) so that the effects of a varying initial population were eliminated. Even so, there was still considerable variation in the best time and mass of the synthesized robots, in the improvement in mass and time achieved during optimization of the final requirement group, and in the number of evaluations required to synthesize configurations meeting the acceptance criteria for each group. Each experiment was run for 30,000 evaluations,

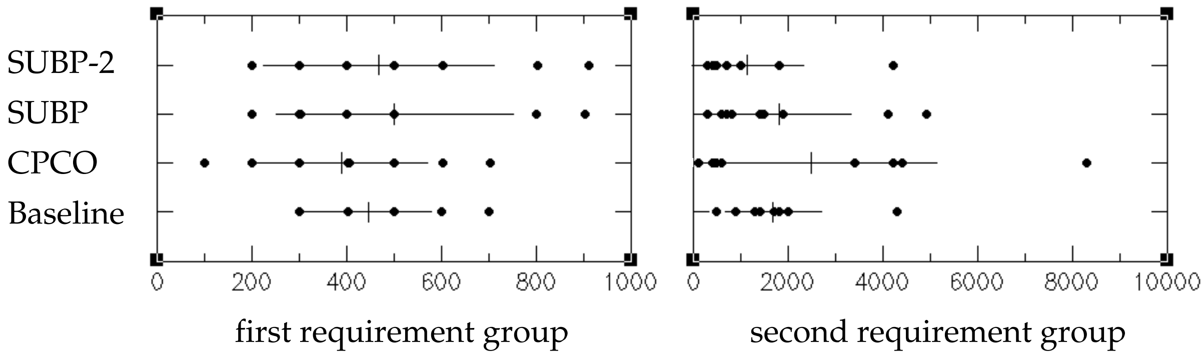


Figure 5.17: Number of evaluations before generating first feasible configuration

The graph on the left shows the number of evaluations required before generating the first configuration satisfying the first requirement group's criteria, while the graph on the right shows the number of evaluations to generate a feasible configuration after starting the second requirement group.

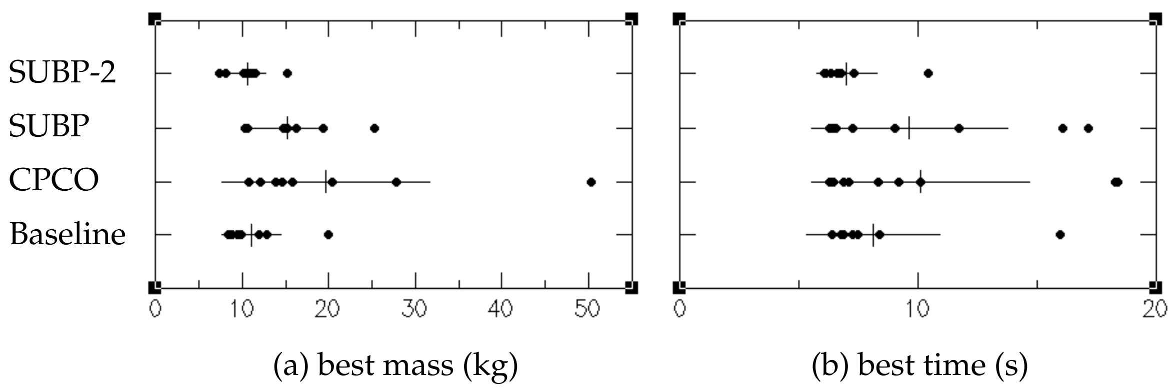


Figure 5.18: Mass and time for full synthesis runs with varying crossover probabilities

The left graph shows the mass of the lightest feasible configuration for each trial of the four experiments; the graph on the right is similar but for task completion time. Each trial began with the same initial population and proceeded through optimization of all three requirement groups until a total of 30,000 configurations had been evaluated.

---

requiring approximately 1 to 2 hours depending on workstation availability.

Figure 5.17 shows number of evaluations required to generate configurations that satisfied the first and second requirement groups for each trial. There was a wide variation in the number of evaluations required to generate feasible configurations for the second requirement group, with the SUBP-2 being slightly better than the Baseline and SUBP experiments. Examining the performance of the best configurations generated in each run (Figure 5.18) there is also a large amount of variation, though the SUBP-2 trials showed less variation than the others. On average, the SUBP-2 trials generated the lightest configurations while the Baseline trials generated the fastest configurations; however, the SUBP trials were close behind in terms of both mass and time and may indicate better general-purpose performance. The Baseline and SUBP-2 experiments were fairly consistent (and close to each other) in terms of the mass of the lightest configuration, but the SUBP-2 trials generated configurations with slightly better task completion times than the Baseline and other experiments. The SUBP trial generated robots with mass about 50% higher than the Baseline and SUBP-2 trials; time for the SUBP robots was about 40% higher than for SUBP-2. CPCO performed the worst out of all the methods.

Though the SUBP-2 trials generated the best results overall, they were only slightly better than the Baseline trials. This seems surprising, since the SUBP-2 and Baseline trials represent the extreme cases, as it were: the Baseline experiments used no commonality-preserving crossover or subgraph preservation, while the SUBP-2 trials used no module crossover and performed subgraph preservation. The CPCO and SUBP trials were in between the two other methods, and did significantly worse.

To try to understand the reasons for these outcomes, another series of experiments was conducted in which the synthesizer started with an initial population of feasible configurations and optimized only the final requirement group (mass and time) over 20,000 evaluations. This eliminates much of the variation between different trials, as the feasible and less diverse initial population focuses the search in a much smaller region of the search space and the synthesizer can sharply limit exploration based on feasibility of solutions (infeasible solutions cannot be in the elite set and are therefore likely to be quickly deleted from the population). Additionally, the time spent optimizing the final requirement group is constant in these experiments, whereas in the previous set of experiments it depended on how much time was spent optimizing the first two requirement groups.

As shown in Figure 5.19, the best mass and time of robots generated in each trial vary much less and than in the previous experiment, indicating that the optimization of the first two requirement groups can have a significant effect on the final outcome of the synthesizer. The Baseline experiments were fairly consistent in terms of the mass of the lightest configuration, and also had the best (lowest) average mass over the twelve trials. SUBP-2, while occasionally able to generate very well-optimized configurations, had average mass and time higher than each of the other three methods. SUBP had average mass and time slightly better than CPCO and was more repeatable with respect to time, but the difference in average performance was a small fraction of the standard deviation of both mass and time and for both experiments, so a conclusive judgement on the basis of these numbers cannot be made. The Baseline trials had average time slightly worse than CPCO, but again the difference was a small fraction of the standard deviation of either method.

Based on these results, we might conclude that the Baseline method -- without subgraph preservation or the commonality-preserving crossover operator -- should be used, as it generated the single best result in terms of both mass or time, had the lowest average mass, and was not conclusively better or worse than the CPCO or SUBP trials with respect to task execution time. However, when we look at the average over the twelve trials of the best (lightest and fastest) configurations versus number of evaluations, a slightly different picture emerges (Figure 5.20). In the figure, it can be seen that the SUBP trials are able to consistently optimize mass and time more quickly than the Baseline method; for example, had the experiments been stopped at 10,000 evaluations the SUBP method would have appeared significantly better with average mass about 2.5kg less than that of the Baseline

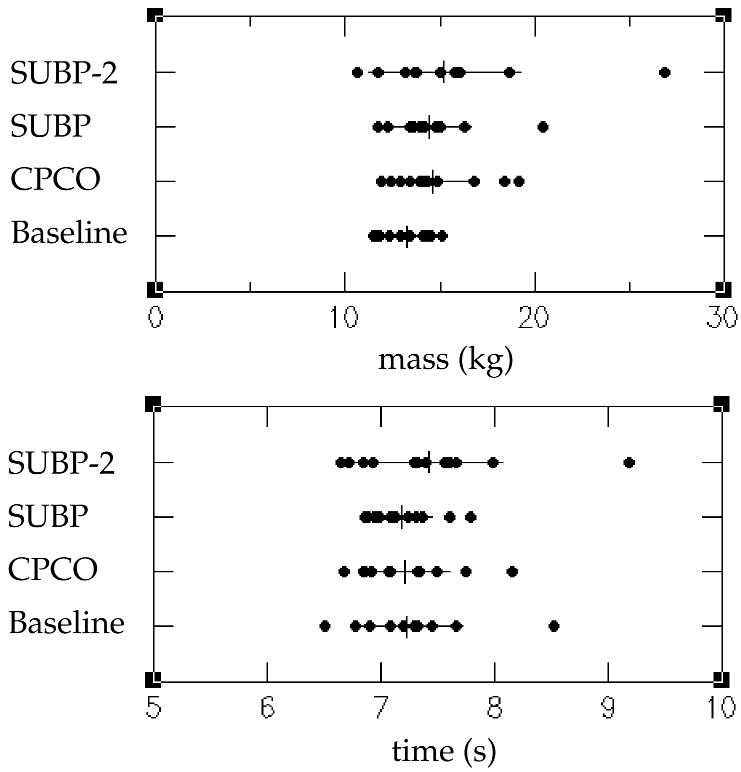


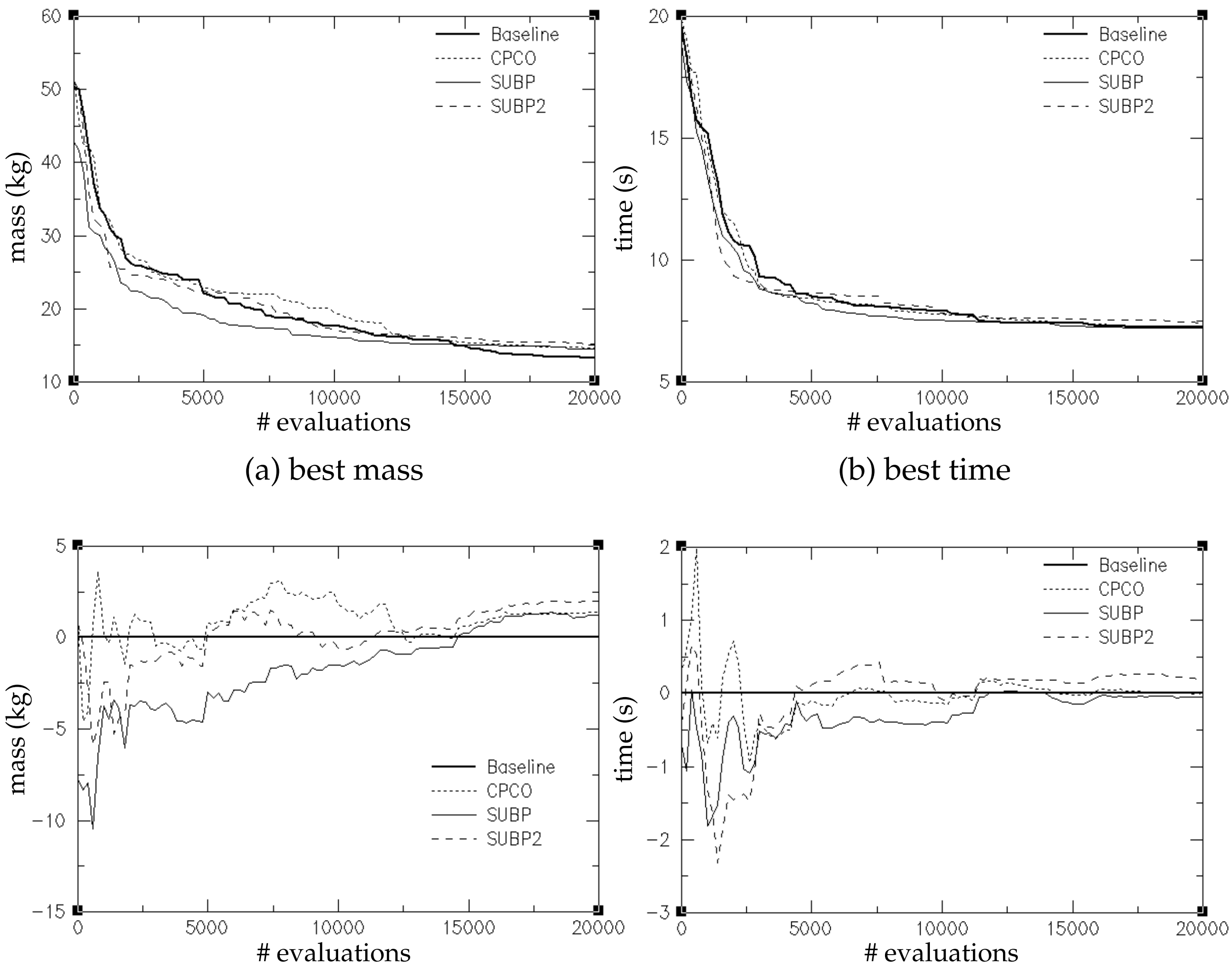
Figure 5.19: Best mass and time for final requirement group with varying crossover operators

These graphs show the best mass and best time over 12 trials of each experiment. All of the trials started from the same set of configurations, which satisfied the first two requirement groups. The Baseline experiments were significantly better than the others with respect to robot mass, while the SUPB generated slightly better robots with significantly less deviation in performance between trials. The SUBP-2 results were slightly worse than the others in terms of both time and mass, and had a higher standard deviation as well.

---

trials and average time about 0.5s faster. Ultimately, the SUBP, CPCO, and SUBP-2 reach their respective local optima sooner than the Baseline method, and in this case the Baseline method generated the lightest configurations in the steady state. In retrospect, this is not surprising: the CPCO, SUBP, and SUBP-2 methods all try to exploit solutions that are known to perform well by restricting the form of the solutions' offspring so that they contain the largest subgraph common to the two parents, while the Baseline method has no such restrictions and thus performs more exploration of the design space. It is worth noting that there is not as much variation between trials in task completion time as there is in robot mass. This makes sense since task completion time is largely determined by task parameters which are independent of robot topology, and the only difference between the different experiments is in how they change robot topology. (Note that the probabilities for parameter crossover and for all mutation operators are the same in all experiments.)

What recommendations can be made on the basis of these results? The most con-



(a) best mass (b) best time

(c) best mass relative to baseline (d) best time relative to baseline

Figure 5.20: Best mass and time versus number of evaluations (average of 12 trials)

The SUBP trials exhibited faster improvement in mass and time than the other trials, though given time the Baseline trials generated robots that performed slightly better than those from the other methods.

servative is that if computation time is not an important limitation, then the Baseline method should be used. On the other hand, if rapid optimization is desired, then subgraph preservation and the Commonality-Preserving Crossover should be used. Exclusive use of the CPCO and subgraph preservation (as in the SUBP-2 trials) was useful in the full synthesis runs but was detrimental in the trials that optimized only the final requirement group. It is not clear where the “sweet spot” is with respect to the relative frequency of the CPCO and normal module crossover operators; a small (0.05-0.1) frequency for the CPCO and much larger (0.6-0.7) frequency for module crossover may harness some of the benefits of subgraph preservation while retaining the ability to make significant changes to robot topology. Currently, Darwin2K will preserve the largest common subgraph in two parent configurations whenever all of the following four conditions are true:

- subgraph preservation is enabled by the user; and
- the probability for the CPCO operator is non-zero; and
- both parent configurations are feasible; and
- either CPCO was selected in the normal manner (i.e. based on the specified weight), *or* at least one of the parents was in the feasibly-optimal set.

Enabling subgraph preservation but specifying a very low probability of CPCO application would allow subgraph preservation to be performed when two feasibly optimal parents are selected for reproduction, while still retaining a large degree of exploration capability by not applying the CPCO too often. It may also be useful to change Darwin2K's behavior so that subgraph preservation and CPCO are performed only until a steady state is reached with respect to robot performance, at which point subgraph preservation and CPCO are disabled so that more exploration of configuration topology can be performed to aid in escaping a local minima.

5.3.2 Comparison of selection methods

All of the experiments presented thus far have used Requirement Prioritization (RP) to select configurations for reproduction and deletion. In addition to RP, Chapter 3 presented several other methods for selection: the weighted sum, which was used in initial experiments with moderate success; the Metric Decision Function (MDF), which RP is based on; and the Configuration Decision Function (CDF), which uses tournament selection rather than fitness-proportionate selection and which requires the designer to specify a simple function for deciding which of two configurations is better. It was stated in Chapter 3 that the MDF, CDF, and RP were all significant improvements over the weighted sum, but backing of this claim has been deferred until now. One major problem with the weighted sum was that it did not account for the dependencies between metrics, e.g. comparing two configurations on the basis of power consumption is not useful unless both configurations can complete the task. Another problem was that since multiple metrics are combined into a single number, the weights must be carefully chosen by the designer based on both the perceived relative importance of the metrics, and the expected range of value in the metrics. The desire to avoid making a scalar value from multiple metrics with vastly different ranges (which caused some metrics to get "lost in the noise") and the need to account for task-specific dependencies between metrics led to the development of the CDF and MDF. While both of these methods were promising, it became apparent that it was possible to formulate a much simpler yet still effective version of the MDF (which was fairly unintuitive for the designer to specify), which led to Requirement Prioritization. In this section, we compare the quality of synthesis results for the weighted sum, CDF, and RP methods. The MDF is not tested as Requirement Prioritization is internally similar to the MDF yet is much simpler for the designer to specify.

In each of the experiments, the synthesizer used a population size of 100 and was limited to 50,000 evaluations. All synthesizer parameters except those specific to the selection method (weighted sum, CDF, or RP) were identical in all trials. Subgraph preservation was enabled and the CPCO, module, and parameter crossover rates were 0.05,

Metric	Priority for RP	Weight for Weighted Sum
pathCompletionMetric	0	10
collisionMetric	0	10
positionErrorMetric	0	10
actuatorSaturationMetric	1	5
linkDeflectionMetric	1	5
massMetric	2	2.6
timeMetric	2	7.7

Table 5.10: Parameters for Requirement Prioritization and Weighted Sum experiments

The priorities used for RP are integer-valued and do not depend on the relative dynamic ranges of metrics since multiple metric values are never combined. On the other hand, the weights used for the weighted sum must account for the difference in magnitude between the best achievable values in different metrics, as well as the relative importance of metrics.

0.65, and 0.25, respectively. (These rates were set based on the experiments detailed in the previous section.) Mutation probabilities were the same as in previous experiments. Seven trials were conducted for each selection method, each of which succeeded in generating feasible configurations. Figure 5.21 shows the decision function used for the CDF trials, while Table 5.10 lists the priorities used for Requirement Prioritization and the weights used for the weighted sum. All other values for each metric (scale, minimum, maximum, and acceptance threshold) are the same as in the previous experiments (see Table 5.8).

In the case of the weighted sum, the weights for the first five metrics were chosen purely based on relative importance since the acceptance thresholds dictated that the optimal value must be achieved (or very nearly so) for a configuration to be considered feasible. Thus, for these metrics it could be expected that an adjusted fitness of 1 would be achieved. The weights for mass and time were more problematic since the best achievable values have to be guessed. For example, if we simply assign a weight of 1 to mass and it turns out that the best possible mass for a feasible configuration is 15kg, then given the scale value for mass of 0.139 the best adjusted fitness for mass is $e^{-(10*0.139)} = 0.124$. Recall that the adjusted fitness is computed as $a(i) = e^{-s(i)}$, where

$$s(i) = scale \times \begin{cases} max - raw\ fitness & \text{positive sense} \\ raw\ fitness - min & \text{negative sense} \end{cases} \tag{5.2}$$

Thus, if we decide that mass should have an importance of 1 relative to an importance of 10 for path completion, we need to assign a weight of 1/0.124 = 8 to mass to account for the fact that the adjusted fitness for mass will never reach 1. Alternatively, we could set the minimum value for mass to 10kg so that an adjusted fitness of 1 could be achieved,

```
// first, decide on percentage of task completed (metric 0)
if (cfg1->metric[0] > cfg2->metric[0]) return cfg1;
else if (cfg2->metric[0] > cfg1->metric[0]) return cfg2;

// choose cfg w/ fewer collisions (metric 1)
if (cfg1->metric[1] < cfg2->metric[1]) return cfg1;
else if (cfg2->metric[1] < cfg1->metric[1]) return cfg2;

// decide on error (metric 2) only if > 3cm
if (cfg1->metric[2] > 0.03 || cfg2->metric[2] > 0.03) {
  if (cfg1->metric[2] < cfg2->metric[2]) return cfg1;
  else if (cfg2->metric[2] < cfg1->metric[2]) return cfg2;
}

// decide on actuator saturation (metric 3) only if > 1.0
if (cfg1->metric[3] > 1.0 || cfg2->metric[3] > 1.0) {
  if (cfg1->metric[3] < cfg2->metric[3]) return cfg1;
  else if (cfg2->metric[3] < cfg1->metric[3]) return cfg2;
}

// decide on link deflection (metric 4) only if > 2mm
if (cfg1->metric[4] > 0.002 || cfg2->metric[4] > 0.002) {
  if (cfg1->metric[4] < cfg2->metric[4]) return cfg1;
  else if (cfg2->metric[4] < cfg1->metric[4]) return cfg2;
}

// decide on time (metric 5) and mass (metric 6)
if (cfg1->metric[5] < cfg2->metric[5] &&
    cfg1->metric[6] < cfg2->metric[6]) return cfg1;
else if (cfg2->metric[5] < cfg1->metric[5] &&
         cfg2->metric[6] < cfg1->metric[6]) return cfg2;
// cfg1 & cfg2 are in a non-dominating relationship
else if (cfg1->metric[5] > cfg2->metric[5]) {
  if (cfg1->metric[5]/cfg2->metric[5] >
      cfg2->metric[6]/cfg1->metric[6]) {
    // cfg1's mass is proportionally better than cfg2's time
    return cfg1;
  }
} else {
  if (cfg1->metric[6]/cfg2->metric[6] >
      cfg2->metric[5]/cfg1->metric[5]) {
    // cfg1's time is proportionally better than cfg2's mass
    return cfg1;
  }
}
return cfg2;
```

Figure 5.21: Configuration Decision Function

The CDF used in experiments comparing selection methods sequentially compared different metrics until a meaningful (in terms of the task) difference in performance was discerned.

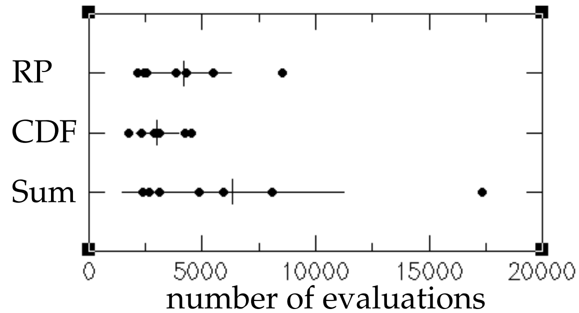


Figure 5.22: Number of evaluations required to generate first feasible configuration using different selection algorithms
Using the Configuration Decision Function (CDF) for selection, the synthesizer was able to generate feasible configurations slightly faster than when using Requirement Prioritization (RP). Both methods were faster than using a weighted sum of metrics.

---

but this approach is flawed since that we don't actually know a priori what the best value is. In contrast, neither Requirement Prioritization or CDF ever have to combine metrics and thus do not require the minimum value to be accurately chosen. In this case, the RP and CDF experiments were run before the weighted sum trials, so reasonably accurate values for the best achievable mass (7kg) and time (6s) were available and were used to give the weighted sum trials a better chance of success. When accounting for these lower bounds, weights of 2.6 and 7.7 give effective weights of 1 to mass and time, respectively.

The initial comparison experiments did not use the `prismaticTube` module, resulting in a more difficult synthesis task. In these experiments the synthesizer was unable to produce any feasible configurations using the weighted sum within the specified time limit of 12 hours, during which 46,000 configurations were evaluated. In contrast, when using the CDF only 10,420 evaluations were required to produce the first feasible configurations, and only 7,262 evaluations were required when using RP. Based on these initial results, the `prismaticTube` was added to the module database to make the synthesis problem easier, thus allowing comparison of the performance of feasible robots generated using all three methods.

Figure 5.22 shows the number of evaluations necessary to generate the first feasible configuration for each trial. To generate the first feasible configuration, on average the CDF trials required about 25% fewer evaluations than the RP trials and about 50% fewer evaluations than the weighted sum, and was also more consistent in the number of evaluations required. Both the Requirement Prioritization and the Configuration Decision Function trials consistently generated better-optimized configurations than the weighted sum trials (Figure 5.23). While the CDF trials generally resulted in lighter robots than the RP trials, RP was able to produce slightly faster robots than the CDF. Given the small number of trials conducted and the possibility that the results are somewhat dependent on the synthesis task, it is difficult to make statistically-significant conclusions comparing the three methods; however, from a designer's perspective, CDF and RP are easier and more intuitive to use than the weighted sum, and seem to produce better results. The improved performance of the CDF and RP methods over the weighted sum can be attributed to the fact that they enable the synthesizer to make better choices about which configura-

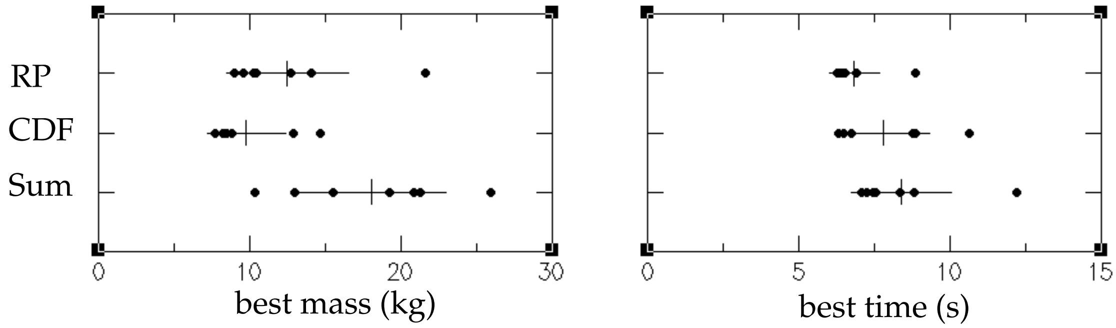


Figure 5.23: Best mass and time of synthesized robots using different selection algorithms

Both Requirement Prioritization and the Configuration Decision Function generated robots with markedly better performance than those generated using the Weighted Sum. Though the performance of robots generated using RP and CDF are generally competitive, RP's parameters are significantly easier for the designer to specify than a Configuration Decision Function.

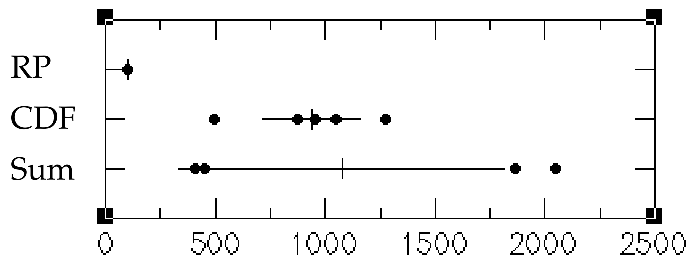


Figure 5.24: Peak population size for weighted sum, CDF, and RP

When determining membership in the feasibly-optimal set, Requirement Prioritization uses fewer metrics than the CDF and the weighted sum methods thus greatly limiting the size of the set. In contrast, the CDF and weighted sum must frequently increase the population size to keep pace with the ever-growing optimal set.

tions should be selected for optimization. The CDF concretely encodes the designer's thought process for deciding which of two configurations is best, so the synthesizer essentially asks the designer to do the comparisons for every decision it makes. Requirement Prioritization improves the synthesizer's ability to make task-relevant choices by focusing on those metrics that are not well-optimized with respect to the task's requirements, and by optimizing metrics in order of their relative importance.

One advantage of Requirement Prioritization relates to the fact that only a subset of the metrics are used to determine membership in the feasibly-optimal set: the feasibly-optimal set is typically much smaller with Requirement Prioritization than with the weighted sum or CDF since fewer metrics are used (Figure 5.24). Since Darwin2K increases the population size to make room for all configurations in the feasibly-optimal set, trials using the weighted sum and CDF frequently required many increases in population size resulting in drastically increased memory use by the synthesizer. In some cases this caused many evaluator processes to idle (thus increasing synthesis time), as the synthesizer required more time to select configurations for reproduction and deletion and to sort the feasibly-optimal population for data logging purposes.

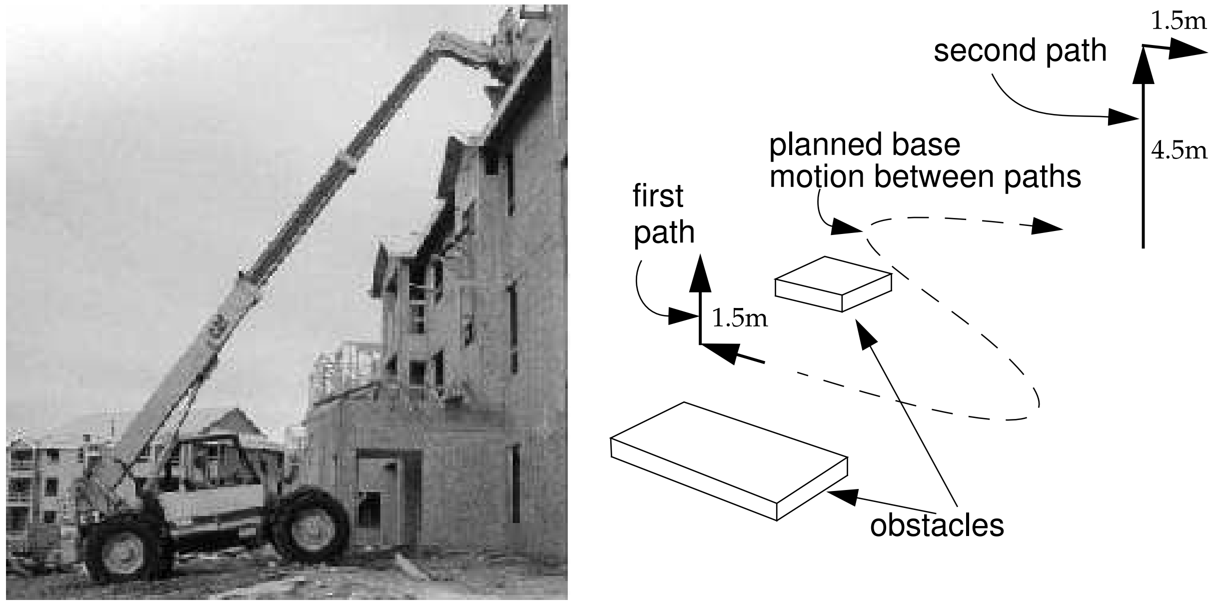
It should also be emphasized that Requirement Prioritization is much simpler for the designer to specify than the CDF: contrast the 7 integers given in Table 5.10 for RP to the non-trivial configuration decision function in Figure 5.21. Both of these methods, however, are much more intuitive than the weighted sum: Requirement Prioritization simply indicates a ranking for metrics and the CDF compares the performance of two robots in a predictable and meaningful way. With the weighted sum, on the other hand, the meaning -- in terms of significance to the task-- is not clear and it is hard to get a feeling for how changing the weights will affect the synthesizer's selections.

In summary, the Configuration Decision Function and Requirement Prioritization methods are both easier to specify (requiring less guessing of "magic numbers") and are more effective at optimizing multiple metrics than the weighted sum, though the expense of gathering test results makes it difficult to make statistically significant conclusions. The ability to encode non-linear, task-specific dependencies between metrics as well as avoid numerical problems based on differing (and initially unknown) dynamic ranges between metrics make these methods more effective than the weighted sum. Though not relevant for this task, Requirement Prioritization's successive use of metrics allows computationally-intensive evaluation methods (such as dynamic simulation) to be avoided until required by an active metric, as was done in the free-flyer experiment. Because of this, overall synthesizer runtime can be less for Requirement Prioritization than for the CDF or weighted sum methods. There is no clear winner between Requirement Prioritization and the Configuration Decision Function: Requirement Prioritization is used with fitness-proportionate selection, while the CDF is used with tournament or rank selection. Selecting parameters for Requirement Prioritization is much easier than specifying a CDF, although the CDF exhibits slightly better performance (at least on this task). Both methods are effective at multi-objective optimization and are likely to be applicable to other synthesis domains.

5.4 Task 4: A Material-handling robot

Many heavy equipment manufacturers produce material handling vehicles that lift and carry heavy loads over uneven terrain. The smallest have maximum payloads of about 2500kg, and a vertical reach of 6m; larger versions can lift substantially more, and to a greater height. Most material handlers have a similar configuration: a telescoping boom with pallet forks is mounted on a four-wheel steer chassis suited for uneven terrain. chassis. Figure 5.25a shows a typical material handler in operation.

Applying Darwin2K to a material handling task characteristic of those addressed by real material handlers allows us to compare Darwin2K's results with manually-designed configurations, and demonstrates the synthesis of a mobile manipulator. As this was the first synthesis problem performed by Darwin2K, many of the synthesis and simulation capabilities now available in the system had not yet been implemented, specifically dynamic simulation, component selection, measurement of actuator saturation,



(a) (b)

Figure 5.25: Actual material handler and experimental task

(a) shows an actual material handler placing a load, while (b) shows the task used in simulation. The robot first picks up a 2500kg payload (the first path), then drives to the second path via a trajectory planned by the motion planner, and then raises the payload to a height of 6m.

requirement prioritization, commonality-preserving crossover, elitism, link deflection estimation, and collision detection. The task used for the synthesis problem (Figure 5.8b) contains two endpoint trajectories and two rectangular obstacles. The first trajectory represents approaching and lifting the payload (a 2500kg mass), and the second trajectory represents raising the payload to a height of 6 meters and then moving forward to place it. The velocity along each path was specified to be 1 meter per second.

Configurations were evaluated in kinematic simulation, with the SRI controller generating joint velocity commands. The simulator for this task later served as the basis for the `pathEvaluator`: the first step was to determine the initial base pose for each path by allowing the base to move freely in the plane as the end effector moved towards the start of each path, and using the bases's degrees of freedom in preference to the manipulator's. The robot then followed the first trajectory, and then drove along a path (generated by the motion planner) to the second base pose that avoided the two obstacles in the workspace. Finally, the robot followed the second trajectory. Stability is a primary concern for material handlers; to emphasize this, the simulation was terminated if the robot's energy stability margin (see Section 4.7.5) fell below zero at any point. Path completion, energy stability, peak actuator torque, and task completion time were used as the metrics for this task. Since velocity is constant along each path, differences in task completion time were mainly due to the time required for the base motion.

The constraints and modules used for this problem were based on a rough knowledge of the task. Each trajectory only requires 3 degrees of freedom, so the number of de-

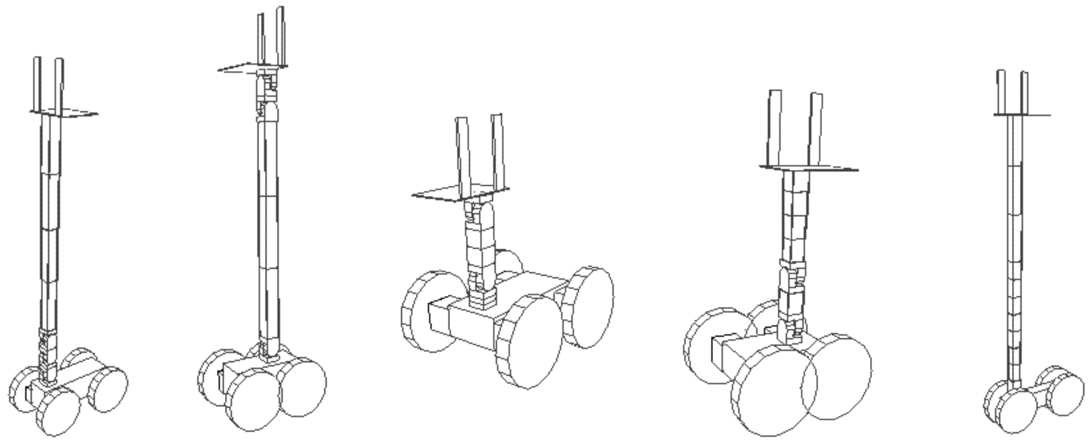


Figure 5.26: Sample material handlers from initial population
The initial population was quite diverse, and poorly-suited for the material handling task: the two leftmost robots above could reach only the first point on the first trajectory, though they were not stable, while the others could not reach the first point at all.

grees of freedom allowed in the manipulator was constrained (via the `dofFilter`) to be either 3 or 4 (the extra DOF was allowed to see if the synthesis program could take advantage of it). The permissible modules consisted of three base modules, four joints, one link, and one tool. The bases were an Ackerman-steer base, a four-wheel steer base, and a Mecanum-wheeled base [Muir88]. Each of the bases have the same parameters: wheelbase, wheel separation, wheel diameter, height, and front-to-back location of the connector. Since the simulation does not account for the effects terrain on the motion of the bases, wheel diameter and base height were set to reasonable values (based on estimated requirements for clearance, traversability, chassis stiffness, and the need for adequate volume for an engine, and hydraulic pumps); in this case, human expertise must be substituted for simulation capability. Since none of these factors were modeled by the simulator, the synthesizer cannot meaningfully optimize them.

The link and joint modules also had some constant parameters: the cross-sectional dimensions of the link (a simple box beam) and joints (an elbow joint, an inline revolute joint, and two prismatic joints) were set to 0.4 by 0.4 because the simulator did not yet estimate link deflections. Given the opportunity to modify these parameters, the synthesizer would eventually settle upon designs with minimal cross sections since this would minimize link mass and therefore joint torque--there is no trade-off to be made against link strength or stiffness. This limitation later led to the addition of link deflection calculations (as detailed in Section 4.5) and the `linkDeflectionMetric`.

The kernel configuration consisted of a set of pallet forks mounted on the center of the top surface of an Ackerman-steer base, with random parameter settings. The initial population, generated randomly from the kernel, was quite diverse, and quite unsuited for the task as can be seen from the examples in Figure 5.26. There is significant variation between the robots: some have only prismatic joints, while others have only revolute joints. The type and size of the base modules vary, as does the position of the attachment

Configuration:	56019
Path Completion:	1.0
Minimum Stability:	4308 J
Peak Torque:	119828 Nm
Execution time:	19.04 sec
Evolved features:	3-piece telescoping boom, redundant elbow joint Ackerman-steer base Base dimensions: 4.5m x 1.7m

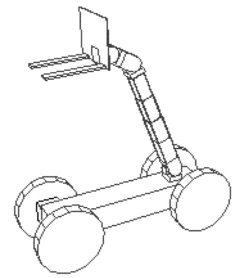


Figure 5.27: First feasible material handler configuration

After over 50,000 evaluations, this configuration was the first that could follow both trajectories without reaching tipover. Significantly, it was one of the few configurations in the population that had a telescoping boom.

of the manipulator to the base. This diversity is important to the genetic optimization process; without such diversity in the initial population, the design space would not be well represented and the process would be especially susceptible to getting stuck in a local minimum.

In early experiments, a randomly-selected metric was used each time a configuration was selected for reproduction or deletion; later, the average of the metrics was used. It quickly became obvious that path completion was more important than the other metrics: many robots could not even reach the first point on the path, but had good values for most metrics. These robots quickly dominated the population. To avoid this, a simple modification was made to the computation of adjusted fitness for each metric: the adjusted fitness was scaled by the square of path completion. This modification emphasizes configurations which complete most or all of the task. Configurations that can only complete a small part of the task are rarely selected for reproduction. Using this modification, the synthesis process generated many configurations which could complete the task. (This problem was an early example of the non-linear dependencies between metrics which later resulted in the development of the Configuration Decision Function and Requirement Prioritization methods for selection.)

In the first trial to incorporate the fitness scaling mentioned above, evaluators were run on approximately 15 Silicon Graphics workstations. Over the course of 9 hours, approximately 70,000 configurations were generated. The initial population was 5,000 and was decimated to 500. Progress was very slow; after more than 50,000 evaluations, a robot that could successfully follow the trajectories without tipover was created (Figure 5.27). Significantly, it was one of the few robots in the population with a prismatic beam. After another 12,000 evaluations, topologies that include the prismatic beam were widespread among the best robots. 28 out of 500 configurations could complete the task, and all had the same general topology of a mobile base followed by an elbow joint, a prismatic beam (2 or 3 piece), 1 or 2 elbow joints, and the pallet forks. The Pareto-optimal set (manually extracted from the population file, as the synthesizer did not yet compute the set) is shown in Figure 5.28. Several differences are noticeable between the three configurations: the first configuration has a 3-piece telescoping boom, which is shorter and slightly lighter than the 2-piece boom used in the other configurations, thus decreasing peak joint torque. The other two configurations differ only in that the lower configuration has a wid-

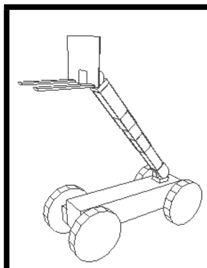
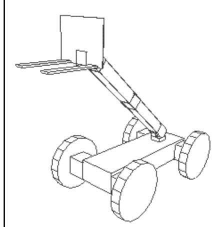
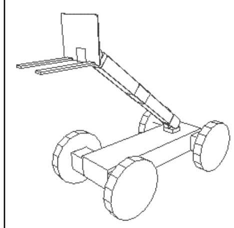
			
Path completion	1.0	1.0	1.0
Min. stability	4937 J	1698 J	7077 J
Peak torque	111443 Nm	126259 Nm	126259 Nm
Time	18.7 sec	12.7 sec	12.6
Features	3-piece telescoping boom Ackerman base Base dimensions: 4.14m x 1.7m	2-piece telescoping boom Mecanum-wheeled base Base dimensions: 4.14m x 2.5m	2-piece telescoping boom Mecanum-wheeled base Base dimensions: 4.5m x 2.5m

Figure 5.28: Pareto-optimal material handlers after 62,000 evaluations

---

er base, and so is more stable. These two robots have a Mecanum base, giving them significantly better task completion time; the Ackerman-steer base must do much more maneuvering when moving between base poses.

After another 3,300 evaluations, the population had nearly converged on an optimum. The average adjusted fitness for path completion, energy stability, and execution time had all approached 1. There were many pareto-optimal configurations, all of which are very similar or identical (duplicate configurations were not eliminated when this experiment was run). The most stable of the Pareto-optimal configurations is shown in Figure 5.29. The subtle differences between the configurations in Figure 5.28 have been resolved, as the robot in Figure 5.29 incorporates the best features of the three: a large Mecanum base; a three-piece telescoping boom; and the rearmost possible manipulator mounting. In fact, the best 24 configurations all have this configuration (Mecanum base and three-piece boom), but with slightly varying base dimensions. Except for 2 of the 3 robots that did not complete the task, all of the robots in the population had prismatic beams. Some had an extra elbow joint, and only 4 configurations did not have the Mecanum base.

To test the repeatability of these results, 5 additional synthesis runs were performed. Figure 5.30 shows the average path completion for each of the six runs. Two of the six experiments (corresponding to the two lowest graphs in the figure) failed to produce feasible results. While one of the unsuccessful experiments was terminated after only 25,000 evaluations, it did not appear very likely that further runtime would have produced a feasible result. There is substantial variation between the time required to

Configuration: 68054
Path Completion: 1.0
Minimum Stability: 14030 J
Peak Torque: 111207 Nm
Execution time: 12.64
Features: 3-piece telescoping boom
Mecanum-wheeled base
Base dimensions: 4.5m x 2.4m

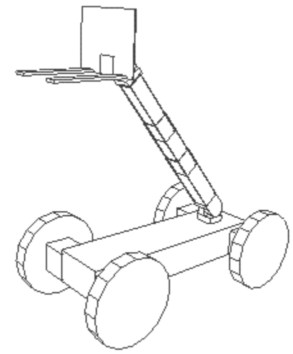


Figure 5.29: Most stable material handler
After approximately 3,300 more configurations, many configurations have been produced which have the 3 piece boom (reducing joint torque) and large, Mecanum base (reducing execution time and increasing stability)

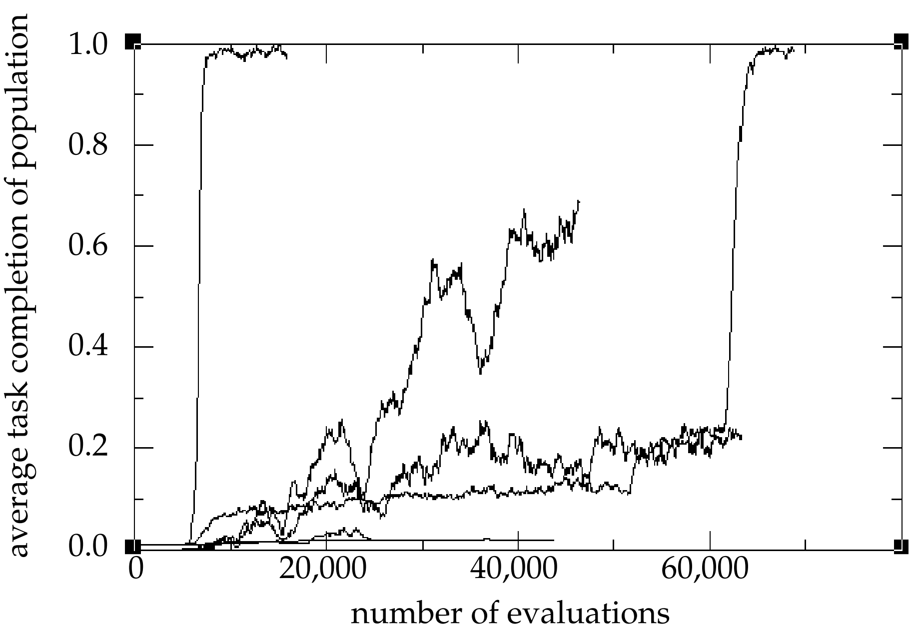
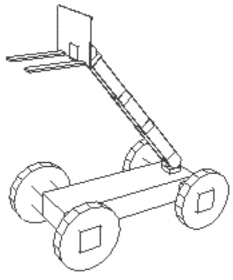


Figure 5.30: Path completion for six synthesis runs
The number of evaluation required to generate successful material handlers varied widely across the six runs, and two did not succeed in producing feasible configurations at all. These results were early indications that the weighted sum was limited in its ability to optimize multiple metrics.

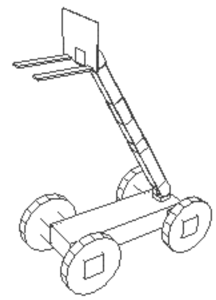
produce feasible configurations in the four successful runs, though the resulting configurations (Figure 5.31) are all quite similar.

5.4.1 Summary and discussion

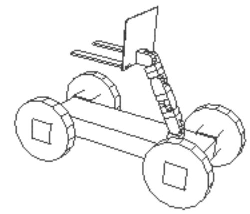
These initial experiments were useful in confirming Darwin2K's feasibility and in



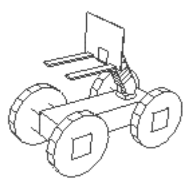
Iterations: **73618**
Completion: **100 %**
Stability:**15917 J**
Torque: **111207 Nm**
Time: **12.6 s**



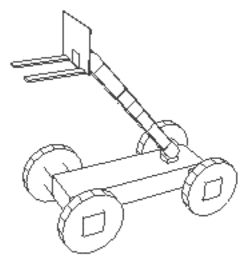
Iterations: **20470**
Completion: **100 %**
Stability: **6470 J**
Torque: **123624 Nm**
Time: **12.7 s**



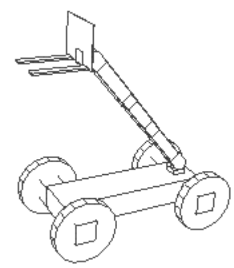
Iterations: **15450**
Completion: **52.9 %**
Stability:**11882 J**
Torque: **75922 Nm**
Time: **-**



Iterations: **47876**
Completion: **52.9 %**
Stability: **10612 J**
Torque: **79013 Nm**
Execution Time: **-**



Iterations: **54085**
Completion: **100 %**
Stability:**13498 J**
Torque: **111215 Nm**
Time: **12.7 s**



Iterations: **66262**
Completion: **100 %**
Stability: **15876 J**
Torque: **111214 Nm**
Time: **12.8 s**

Figure 5.31: Results for six material handler synthesis runs
Out of six runs, four resulted in material handler configurations that could successfully complete the task. The four successful configurations all had three-piece booms and large Mecanum bases.

identifying areas for further work. The configurations from four of the experiments meet the design specifications: approaching and lifting a 2500kg payload, carrying it to another location, and raising it to a height of 6 meters. The synthesis process was able to simultaneously synthesize the robots' type and dimensions, including the selection of an appropriate base module. Furthermore, the final configurations are well-optimized in terms of peak joint torque and task execution time, and three out of the four are highly stable as well. The similarity of the results from the four successful runs are an indication that repeatability can be achieved.

The experiment also illustrates several limitations of the current system. The Mecanum base is not suited for natural terrain; a human designer would know this, but since the simulation does not model terrain interaction there is no way for the synthesizer to properly judge the suitability of configurations with the Mecanum base. In the simulation, the Mecanum base was more maneuverable than the other two bases due to its holonomic nature, and was repeatedly chosen as it reduces task execution time. This points to the need for some human expertise when choosing the modules to be used in the synthesis process, as some modules may have limitations that are not exposed by the simulation.

One difference between the synthesized material handlers and the real one shown in Figure 5.25a is that the real one has a bend in the boom near the forks to allow the boom to clear the chassis when the forks are on the ground. At the time these experiments were performed, the simulator did not check robots for self-intersection, so nearly all configurations had self intersections while following the first path. Additionally, the modules used in the experiment cannot be connected to create a bent boom without including an extra joint. It is easy enough to create a new link module that would allow the bent boom to be synthesized, but it is doubtful that the bent boom would be found optimal unless collision detection was included as a metric. While a human could view the results, attach a new link to the pallet forks with a non-variable attachment, and re-run or continue the synthesis process, this experiment pointed to the need for using collision detection as a metric for evaluation. An alternative would be to extend the motion planner to avoid self-intersecting poses, but the added computational complexity of motion planning for the entire robot (rather than just the base) would likely increase evaluation time to unacceptable levels.

While the material handler synthesis experiments had several limitations, they were important in validating the general synthesis approach used in Darwin2K and in helping guide further improvements to the system. Perhaps most importantly, they motivated the inclusion of collision detection, actuator modeling, and estimation of link deflections in the evaluation process, and motivated further work in optimizing multiple metrics. Finally, the demonstration of synthesis and optimization of type and kinematics via the PMCG was a key contribution of these experiments.

5.5 Task 5: An antenna-pointing mechanism

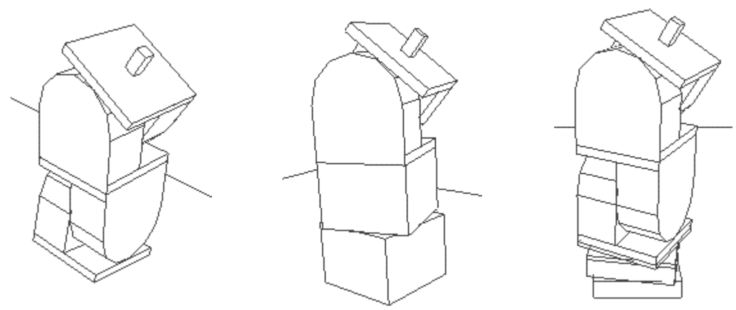
Another early application of Darwin2K was the kinematic synthesis of a rover with an antenna pointing mechanism, motivated by [Bapna98]. Though many of the details of the synthesizer and simulation have since changed, the synthesis results for this task are illuminating in that they can be intuitively seen to be well-suited for the task while simultaneously indicating some pitfalls of automated synthesis. The goal for this task is to synthesize a configuration consisting of a pointing mechanism mounted on a wheeled base that is able to keep an antenna pointed at a distant target as the robot drives over terrain for a variety of headings. In keeping with [Bapna98], terrain was modeled as a pair of sinusoidal elevation functions (one for the wheels on the robot's left side and one for the wheels on the right), with different amplitudes and phases. The rover's pitch and roll are computed from this terrain model by way of a simple kinematic suspension model, and the rover's heading rate is constant so that after 30 seconds the rover has completed a circle. The `antennaEvaluator` was written to encapsulate this task-specific simulation code, and interfaces with a task-specific mobile base module, the `antennaChassis`. The source code for vehicle pose computations based on the sinusoidal terrain model was supplied by Bapna and incorporated into the `antennaEvaluator`. An `antennaPath` object generates velocity commands for the endpoint of a serial chain mounted on the chassis, trying to keep the endpoint (which is the antenna) at a constant azimuth and elevation as the chassis changes orientation; these velocity commands are then translated to joint velocity commands by the `SRIcontroller`.

Five metrics captured the requirements of the task: path completion (the ability to achieve the desired pointing direction), peak joint velocity (rad/s), peak joint torque (Nm), maximum pointing error (rad), and power consumption (W) for the pointing mechanism. Since this experiment was conducted before component selection and link deflection capabilities existed in Darwin2K, the measurements computed for peak joint torque and power consumption were not very meaningful; their main impact was to reduce link lengths to the minimum. The other metrics (peak joint velocity, error, and path completion) are purely kinematic, and were more relevant to the robot's performance and configuration.

Darwin2K synthesized robots for several different cases: one target at a low elevation, one target at a high elevation, and two subsequent targets at differing elevations. Aside from the `antennaChassis` mentioned above, `elbowJoint`, `inlineRevolute`, and `simpleTool` modules were included in the module database. The synthesis runs for the first two cases used a weighted sum for selection, and required some experimentation to determine the best set of weights to use to ensure convergence. For the first two tasks the pointing mechanism was restricted to 2 degrees of freedom and the search space contained about 17 million configurations, many of which were functionally identical. Each experiment ran until 20,000 configurations had been evaluated (approximately 0.1% of the search space) and run time was approximately 15 minutes on 20 SGI workstations, considerably shorter than previous experiments due to the limited size of the search space, simplicity of simulation (control and simulation for only 2 degrees of freedom is significantly faster than for 6 or more), and the fact that this was an inherently easier synthesis problem than those requiring optimization of actuator selection and structural geometry to satisfy actuator saturation and link deflection constraints.

For higher receiver elevations (45 through 90 degrees), the x-y pointing mechanism shown in Figure 5.32a consistently had the best performance. In this case, the geometry of the pointing mechanism is intuitively well-suited for tracking targets at high elevations as the mechanism's singularities only come into play at low elevation angles. For a lower elevation (20 degrees), the azimuth-elevation pointing mechanism shown in Figure 5.32b was consistently best. The azimuth-elevation mechanism's singularity only comes into play for high elevation angles, so it is well-optimized for tracking targets at low elevations. When either of these mechanisms is used for tracking targets near singularities (i.e. at low elevations for the x-y mechanism and at high elevations for the azimuth-elevation mechanism), very high joint velocities are required as the robot's changing heading brings the mechanisms close to their singularities.

For the third synthesis task, the robot was required to track targets at both high and low elevation angles. Based on the results of the previous runs it seemed unlikely that



(a) (b) (c)

Figure 5.32: Antenna pointing mechanisms

The x-y pointing mechanism (a) has better performance at high target elevation angles, while the azimuth-elevation mechanism (b) has better performance at low target elevations. The redundant azimuth-x-y mechanism (c) performs equally well at high and low elevations, since it is free of singularities throughout its range of motion.

any 2-DOF mechanism would be sufficient for tracking targets at high and low elevation angles with high accuracy and low joint velocities, so we allowed up to 4 degrees of freedom for this task. As might be expected, the synthesizer settled on a hybrid design, consisting of an x-y mechanism on top of an azimuthal mechanism. In this case, the azimuth pointing mechanism consisted of two `inlineRevolute` joints with coincident joint axes. This design, while not one that a human would create, performed well with respect to the peak joint velocity metric since any azimuthal velocity was split equally between the two `inlineRevolute` joints. The `moduleRedundancyFilter` had not yet been implemented at this point, so the synthesizer had no reason to rule out designs with coincident joint axes.

5.5.1 Summary and discussion

This synthesis example demonstrates that Darwin2K can generate intuitively understandable, well-optimized configurations for tasks that a human designer can easily grasp and design for. It is easy for a human designer to see that the three pointing mechanisms generated by Darwin2K are well-suited for their assigned tasks, and a human designer would likely choose similar kinematic structures when manually designing the antenna pointing mechanism for each of the tasks. For the Nomad robot [Wettergreen99], which was required to keep its antenna pointed at a low-elevation base station, the designers chose an azimuth-elevation configuration [Bapna98], validating Darwin2K's solution for low-elevation pointing.

Though the pointing mechanisms created by Darwin2K were well-optimized, this task also demonstrates a potential pitfalls of Darwin2K and of genetically-based synthesis in general: evolutionary methods will frequently find unanticipated loopholes in the problem specification. This is illustrated by the redundant azimuthal joints of the azimuth-x-y pointing mechanism: the synthesizer was not prevented from using subsequent joint modules with coincident axes -- which a human designer would probably not consider using in the first place -- and discovered that this configuration reduced peak joint velocity. These lessons are important to keep in mind when specifying the synthesizer's parameters (particularly the configuration filters and metrics), and when developing a task-specific simulator.

5.6 Task 6: A walking robot for space trusses

Recently, there has been a renewed interest in Space Solar Power (SSP)-- orbiting solar power satellites that beam back energy in the form of microwaves, to which the earth's atmosphere is largely transparent. This technology holds the promise of sustained zero-emission power generation (neglecting the impact of numerous rocket launches) while providing more continuous power output than ground-based solar due to its independence from weather conditions and the limited time spent in darkness each day. The geosynchronous orbits in which the satellites would lie are far beyond the Space Shuttle's range, thus greatly restricting human access. Automated assembly and servicing are therefore important enabling technologies for SSP and are currently being investigated; one example is the Skyworker project at CMU. (No references for Skyworker are available at this time, as the project is in its early stages.)

The task we will address is the synthesis of a lightweight walker for visual inspection of the satellite's components. Robotic walkers are preferred over free-flyers for operations on the power satellites, since electrical power for walkers can be drawn from the satellite itself via inductive pickups or mechanical connections whereas free-flyers rely on expendable fuel, implying either extra mass for a large fuel supply, occasional re-supply from Earth, or limited robot lifespan. While an assembly or servicing task (rather than visual inspection) would be a more challenging synthesis problem, it would require dynamic simulation of closed kinematic chains which has not yet been addressed by Darwin2K. We will thus address the synthesis of an inspection robot while retaining as many of the task constraints and properties (such as satellite geometry) as possible.

By current thinking, each power satellite will have a modular construction and will be quite large, on the order of kilometers. Much of the satellite's structure will be composed of trusses, upon which the antenna array (for transmitting power) and solar collectors will be mounted. According to one model, the antenna array will be assembled from triangular sections each consisting of a panel backed by a truss to provide rigidity (Figure 5.33). The inspection robot should be able to move to any point on the truss: it should be able to walk along truss rods, transition between them, and change its orientation with respect to a truss rod so that it can move from one side of the truss to the other. The robot should also be able to position its end effectors so that it can aim a camera at the object being inspected (we will assume there is a camera mounted on each effector, though the cameras will not be modeled in the simulation).

The representative task for the inspection robot is shown in Figure 5.34 and encompasses the capabilities listed above. The robot first walks along a truss segment 5.1m in

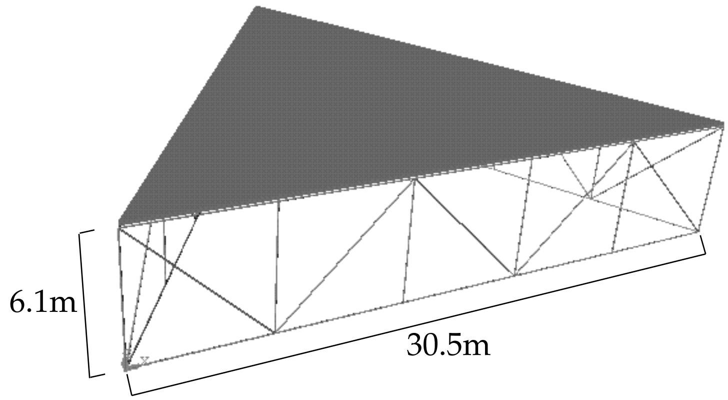


Figure 5.33: SSP satellite panel section
The space solar power satellite's transmitter array is composed of a number of panels, each in the shape of an equilateral triangle and backed by a truss to provide rigidity. During the inspection task, the robot will walk along one of the truss segments, transition to several other segments, and inspect the junction between them.

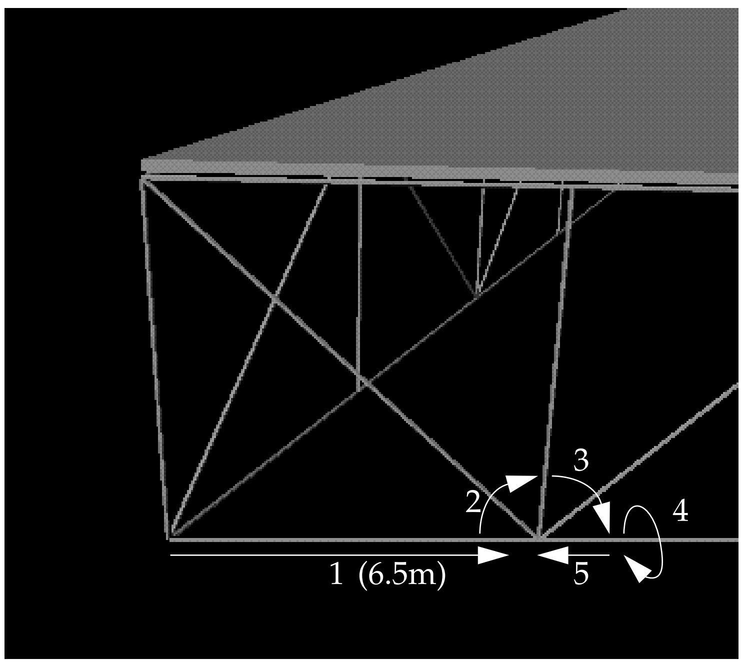


Figure 5.34: Phases representative task for SSP inspection robot
The robot first walks along one of the truss rods (1), transitions to another rod perpendicular to the first (2), move again to another rod (3), moves around to the back side of the truss (4), and positions one end-effector near the truss joint for inspection (5).

length, moves to a perpendicular segment (gripping the segment 1.0m from the truss joint), moves to a third segment, rotates around to the back side of the truss, and then moves one end effector to an inspection location under the truss joint. This task ensures that the robot can reasonably maneuver about the truss in addition to being able to walk

Metric name	Req. Group	Acceptance threshold	Min	Max	Scale	AFDI
dynamicPathCompletion	0	= 1.0 (100%)	0	1	6.93	0.1 (10%)
collisionMetric	0	integral = 0	0	100	0.4	1.73
linkDeflectionMetric	1	max < 0.001m	0.0009m	0.01m	693	1mm
continuousSaturationMetric	1	max < 0.8 (80%)	0.3	10.0	6.9	0.1 (10%)
massMetric	2	none	5kg	50kg	0.139	5kg
powerMetric	2	none	0J	20kJ	0.002	100J
timeMetric	2	none	20s	100s	0.34	2s

Table 5.11: Metrics for SSP inspection robot

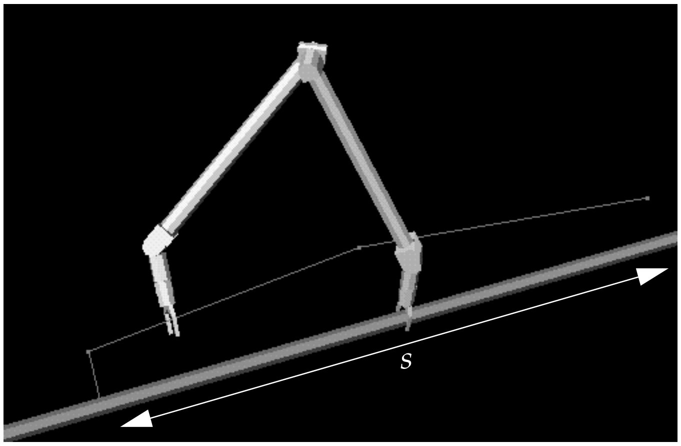
The metrics used for the SSP inspection task are similar to those used in the free-flyer task, though without the `pathCompletionMetric` (since dynamic simulation is always used in this task) or `positionErrorMetric` (since accurate trajectory following is not required).

---

along truss segments. The metrics for this task (see Table 5.11) are a subset of those used for the free-flyer, the differences being the absence of the `pathCompletionMetric` and the `positionErrorMetric`. A dynamic simulation will always be used for this task, so the `pathCompletionMetric` is not needed. The `positionErrorMetric` is not used for this task because accurate trajectory following is not required: as long as the robot can accurately reach the via points of the task without collisions (as measured by the `collisionMetric`), it is not important if the robot deviates significantly from the straight-line trajectory between via points. The `linkDeflectionMetric` and `continuousSaturationMetric` allow synthesis of link structure and actuator selection, while the `massMetric`, `powerMetric`, and `timeMetric` guide the optimizer in producing lightweight, efficient robots.

While the simulator for this task makes use of many of the general-purpose evaluation components (including the `rungeKutta4`, `collisionDetector`, `relativePath`, and `ffController`), some task-specific objects are required for simulation and control. The `panelSection` is derived from the `payload` class and creates the truss geometry shown in Figure 5.33 based on parameters for the length, height, and number of truss segments for each side of the panel. The `panelSection` also provides coordinate information for several classes which aid in trajectory generation for the walker. The `walkerEvaluator` performs high-level simulation control such as querying the trajectory generation object for the next part of the robot's gait, passing the trajectory information (as a `relativePath`) to the `ffController`, and using the `rungeKutta4` object to compute the robot's behavior. There is also a task-specific module, the `walkerGripper`, which is appropriately sized for the diameter of the truss segments. All told, there are approximately 1500 lines of C++ code for this task (including source and header files).

The task-specific simulation and control code assumes a two-limbed robot, as this is the minimum number of limbs to provide continuous contact with the truss during lo-



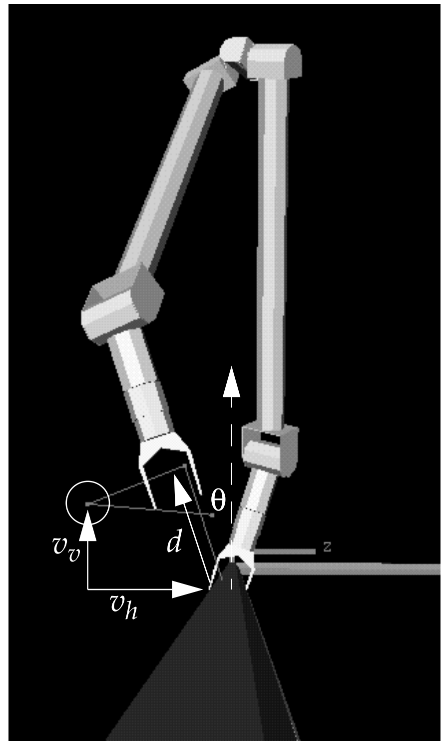


Figure 5.35: Gait parameters

These two images illustrate the parameters for each stride of the walker's gait:

- *s* - distance between successive grasp points for an end-effector
- *d* - distance from segment center to approach point
- θ - angle from normal to approach point
- v_v - offset of via point (white circle) in direction of the normal (the dotted line in the right image)
- v_h - offset of via point perpendicular to the normal
- v_{tol} (not shown here) is an additional gait parameter for how close the end effector must come to the via point.

---

comotion and the robot will not need more limbs for manipulation or other tasks. The robot must have at least six degrees of freedom between its two end effectors so that it can grasp truss segments in arbitrary orientations and in three dimensions; however, it does not need six degrees of freedom in each limb as there is no need for body positioning in this task. When moving along a truss segment, the control code uses a hand-over-hand gait, since this has a longer stride and few accelerations and decelerations than an inchworm-like gait. The gait trajectory generator has 6 parameters (shown in Figure 5.35) all of which will be included in the set of task parameters (Table 5.12) to be optimized. The basic form of each stride in the gait is to release the gripper that is farther back, move it towards a via point specified by horizontal and vertical offsets (v_h and v_v respectively) from the fixed gripper until within a specified distance v_{tol} from the via point. The gripper continues motion until it reaches (and stops at) the approach position at a distance *d* and angle θ above the next grasp location, which is located on the truss segment a distance *s* from the gripper's previous position. The trajectories for gait, for transitioning between truss segments, and for positioning an end effector for inspection are all represented as a `relativePaths`, allowing the `ffController` to be used to for trajectory following. However, there is one difficulty with this approach: the `ffController` controls the motion of one or more end-effectors relative to the robot's base link (which is usually deter-

class	component label	variable name	min	max	# bits
walkerEvaluator	(none)	firstGraspPoint	0	1	1
		hOffset	0.1 m	0.6 m	4
		vOffset	0.1 m	0.6 m	4
		approachAngle	0	$\pi/2$	4
		approachDist	0.1 m	0.5 m	4
		inspectionStandoff	0.2 m	0.8 m	4
		stride	1.0 m	2.0 m	4
relativePath	gaitPath	vel	0.2 m/s	2.0 m/s	5
		maxAcc	0.5 m/s^2	4.0 m/s^2	5
		omega	$\pi/6$ s^{-1}	$\pi/2$ s^{-1}	5
		maxOmegaDot	$\pi/6$ s^{-2}	2π s^{-2}	5
ffController	ffController	ignoreLimitThresh	0	0.5	5
		gradientStepSize	0	0.05	5

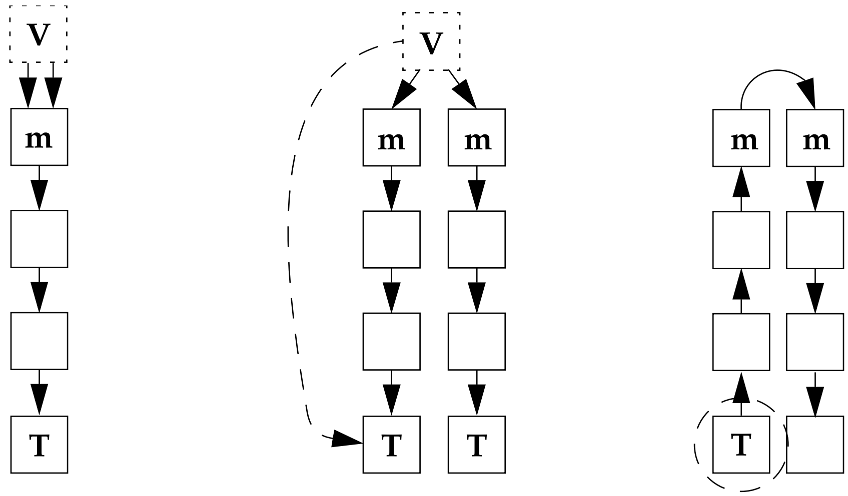
Table 5.12: Task parameters for SSP inspection robot

The first seven parameters are for the walkerEvaluator, which passes the parameters' values to the gait trajectory generator. The next four parameters are for the linear and angular velocity and acceleration used when following trajectories, and the last two determine when and how much the robot's redundant degrees of freedom are used for joint limit avoidance. The last two parameters have significant impact on the robot's ability to accurately stop at via points.

---

mined by the robot's base module), but in this case we want to control the motion of one endpoint relative to another. From a control point of view, it is much simpler to consider the robot to be a single serial chain from one end effector to the other than to think of it as two shorter serial chains from a common base since the latter approach requires computing two trajectories (one for each end effector) relative to the robot's base that are constrained to produce the desired relative end effector motion. Treating the robot as a single serial chain rooted at the gripper that is grasping the truss is simple enough, but requires an additional step since each gripper is alternately fixed or moving. One way would be to re-number the links and joints with the stationary gripper as the robot's base, though this requires recomputing the robot's dynamic equations and re-building all of the data structures based on the robot's kinematic description. Alternatively, we can keep the same kinematic and dynamic description of the robot no matter which gripper is moving and simply change the coordinate system of the trajectory so that the desired relative end effector motion is produced regardless of which gripper is moving. For these experiments I took the latter approach due to its simplicity.

Locating the robot's base link at one of the robot's grippers presents a difficulty if we want the robot to be symmetric: the configuration graph can only preserve symmetry



(a) (b) (c)

Figure 5.36: Effects of the `virtualBase` module
In (a), the `virtualBase` module **V** has two references to module **m**, resulting in duplication of **m** in the instantiated configuration graph (b). **V**'s single parameter selects one of the configuration's endpoints as the robot's base link (dotted arrow in (b)), which in (c) becomes the base link of the robot's kinematic and dynamic descriptions. This allows the robot to have two symmetric limbs while being represented as a single serial chain (including both limbs) for simulation and control purposes.

---

when two subgraphs share a common parent module, which will not be the case if the robot's base module is located at one end of the serial chain. The `virtualBase` module eliminates this problem: it has no geometric description other than two coincident connectors, and has one parameter which selects the link to be designated as the robot's base link. This allows the base link that is used for kinematic, dynamic, and control purposes to be located at one of the grippers rather than at the root of the configuration graph. The robot can thus be symmetric (since the base *module* has two references to a single limb subgraph) while having the base *link* located at the end of the robot's single serial chain. Figure 5.36 shows a schematic depiction of how the virtualBase affects the representation of the robot's links and joints. (Note that for clarity, (c) in the figure is similar to (b) although in general the graph describing the links and joints of the robot is different from the configuration graph.)

At this point it is also useful to introduce the `virtualLink` module which, like the virtualBase, aids in preserving symmetry and has no geometric representation other than a pair of coincident connectors. In the free-flyer experiments the kernel configuration had a base module with two references to an `inlineRevolute2` module, which resulted in two symmetric arms beginning with the `inlineRevolute2`. If we want two symmetric arms but do not want to specify what the first module in the arms should be, we can use the `virtualLink`: `const` incoming connections to the `virtualLink` will not be changed, but the single outgoing connection from the `virtualLink` can be modified by the genetic operators. The `virtualLink` simply acts as a constant reference

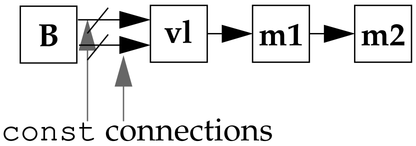


Figure 5.37: virtualLink usage
The `virtualLink` module (**vl** above) is used to allow `const` connections to be made to a varying subgraph. In this example, **m1** and **m2** are free to be replaced; in contrast, if the const connections had been to **m1** then the synthesizer could not change **m1**. The `virtualLink` has no geometry, so while the connections to **vl** cannot be changed, **vl** has no influence on the physical form of the robot.

```
((virtualBase ((const 0 1 3 0))
      ((const 0 (1 0 left (var 0 270 2 0)))
       (const 1 (2 0 right (var 0 270 2 0)))))
 (offsetElbow revoluteJoint ((var 0 1 2 2)
                             (var 0 1 2 8)
                             (var 0 1 2 0)
                             (var -3.14 3.14 4 7)
                             (const 0.003 0.01 3 0)
                             (var 0.005 0.05 2 1))
             ((const 1 (2 0 inherit (var 0 270 2 1)))))
 (virtualLink nil ((var 1 (3 0 inherit (var 0 270 2 1)))))
 (walkerGripper ((const 0.01 0.1 3 1)) nil))
```

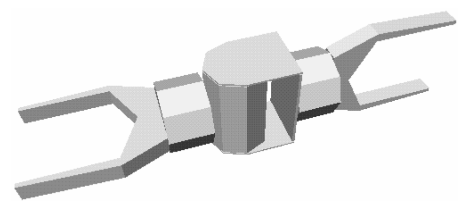


Figure 5.38: Kernel configuration for the SSP inspection robot
The kernel configuration makes use of the `virtualLink` to yield configurations with symmetric limbs ending in `walkerGrippers`, but without specifying any physical characteristics for the first module in each limb. The arrows overlaid on the symbolic descriptions of the kernel show the connections in the two different subgraphs rooted at the base module.

---

point for the arm subgraphs without contributing to the physical description of the arms (Figure 5.37). The kernel used for the experiments (shown in Figure 5.38) specified a joint structure to which the two symmetric limbs should be attached, and used the `virtual-Link` as the reference point for the subgraph shared by the two limbs.

As with previous experiments, the possible configuration topologies were influenced by the module database and configuration filters used by the synthesizer. The `moduleRedundancyFilter` use the same settings as in the previous experiments, disallowing successive joints whose axes were colinear. The `dofFilter` eliminated any configuration with less than 13 or more than 14 DOFs--6 base DOFs and 7 or 8 for the serial chain--though given that the kernels each had 1 DOF and symmetric arms, this restricted the solutions to 7-DOF serial chains. The module database contained the same three revolute joints as in the free-flyer task--the `inlineRevolute2`, `offsetElbow`, and `rightAngleJoint` modules--and also included the `hollowTube` link module. Again, prismatic joints were not used due to the preference for revolute joints in space applications. Given that the walker will not need to manipulate massive payloads, the com-

Table 5.13: List of motors

Maxon RE25.118722
Maxon RE25.118748
Maxon RE25.118755
Maxon RE35.118800
Maxon RE36.118778

Table 5.14: List of gearheads

Maxon 16.134782	Maxon 81.110410
Maxon 16.110325	Maxon 81.110411
Maxon 16.134783	Maxon 81.110412
Maxon 16.118188	Maxon 81.110413
Maxon 26.110396	HD Systems CSF-20-50
Maxon 26.110397	HD Systems CSF-20-80
Maxon 26.110398	HD Systems CSF-20-120
Maxon 32.114489	HD Systems CSF-20-160
Maxon 32.114499	HD Systems CSF-25-50
Maxon 32.114507	HD Systems CSF-25-80
Maxon 32.114513	HD Systems CSF-25-120
Maxon 32.114516	HD Systems CSF-25-160

The component database for the SSP inspection robot is similar to that used for the free-flyer, but contains lighter-duty components since the inspection robot will not manipulating massive payload.

---

ponent database for the walker's modules (Table 5.13 and Table 5.14) contains fewer and less-powerful gearboxes and motors than the database for the free-flyer. Limiting the component database to those that will be most relevant to the task will reduce the search space for the synthesizer, which in turn should lead to decreased synthesis time and improved performance of the synthesized robots.

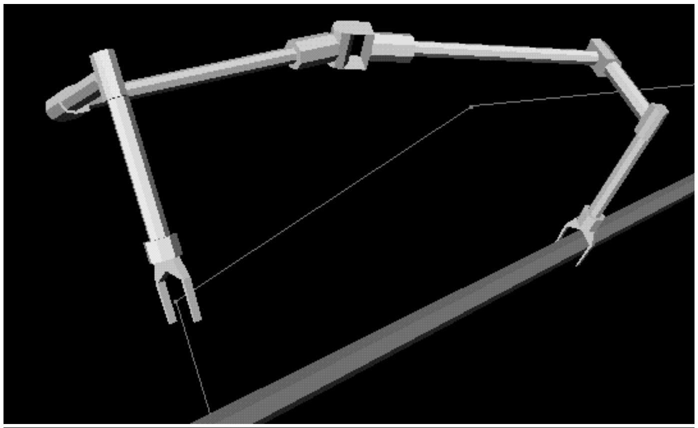
The first experiment used the kernel shown in Figure 5.38a. A population size of 200 (decimated from an initial population of 5000) was used, with a maximum of 80,000 evaluations per stage and a time limit of 13 hours. The first configuration satisfying the first requirement group was generated after performing only 100 evaluations after the initial population was decimated, and after another 4000 evaluations (the minimum per requirement group) 152 out of 200 satisfied the first requirement group. The first configurations satisfying the second requirement group were generated after 1,500 further evaluations (a total of 10,600), and after another 4000 the synthesizer advanced to the third and final requirement group with a total of 70 feasible configurations. At this point, the lightest configuration had a mass of 22.9kg; the most energy-efficient configuration consumed 955J to complete the task, and the fastest configuration completed the task in 85 seconds. The time limit was reached after a total of 52,500 evaluations, including the 5000 for the initial population. At this point the lightest, most energy-efficient, and fastest configurations (cfg_m, cfg_e, and cfg_t, respectively in Figure 5.39) had significantly better performance: the lightest configuration had a mass of 12.2kg, the most efficient used 221J, and the fastest required only 55 seconds to complete the task. Figure 5.40 shows cfg_m at several stages while executing the task. Examining the performance trade-offs of the Pareto-optimal feasible configurations, if we are willing to accept a decrease of up to 5% relative to the best mass, energy, and time, we can have configurations with:

Table 5.15: Trade-off configurations for SSP inspection task

relative to configuration	mass	energy	time
cfg_m	12.25 kg (+0.5%)	311J (+21.9%)	73.8 s (-4.4%)
cfg_e	12.3 kg (0%)	231J (+5%)	83.9 s (-1%)
cfg_t	14.8 kg (-20%)	794J (-28%)	57.7 s (+4.7%)

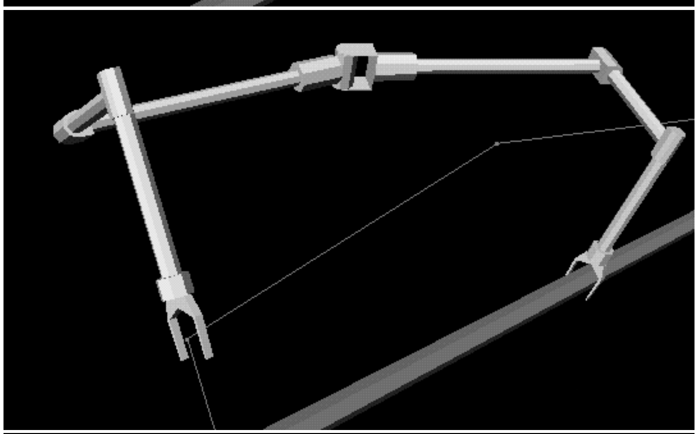
Note that negative percentages are better here, since they indicate a decrease in mass, energy, or time. In this experiment, there were few configurations that represented reasonable trade-offs against cfg_m or cfg_e: for example, a configuration which consumed 5% more energy than cfg_e might only have a task completion time of 1% less, and could have greater mass. There were numerous useful trade-offs against cfg_t, however, as evidenced by the last configuration in the Table 5.15.

The robots in Figure 5.39 seem to have an odd configuration: the three degrees of freedom for each "wrist" are distributed along the length of the arm, rather than being closely-grouped at the end of the arm. The first module of each arm is an `inlineRevolute2`, and the last two are `rightAngleJoints`. 197 of the robots in the final population shared the same configuration graph topology; the remaining three were similar to the majority except that an `inlineRevolute2` module was oriented differently, resulting in slightly different inertial properties. The main drawback of this configuration is that changing the orientation of the end-effector in three dimensions requires large motions, since each of the wrist joints (and corresponding links) must be moved; however, such motions are not required in this task. Are there any benefits to this configuration? After examining the kinematics of the arm, it becomes apparent that this configuration is excellent for avoiding self-collisions: in fact, when considering each half of the arm separately, each of the three wrist joints can be moved through a full revolution without causing any self-collisions (Figure 5.41). This is not generally the case for 3-DOF wrists with multiple intersecting axes, as are commonly found in dextrous manipulators. The robot's inherent lack of self-collisions due to wrist motion make the control problem easier, which is particularly important given that the controller used in the experiment does not try to avoid self-collisions. There has been recent work in identifying ([Full99]) and exploiting ([Zeglin99] and [McGeer90]) "mechanical intelligence" to perform roles normally reserved for active control, specifically for dynamic stability in mechanisms; the kinematic configuration evolved in this experiment has a similar type of mechanical intelligence for avoiding self collisions. This kinematic configuration is also an example of generating novel, unintuitive designs: it seems likely that a human designer would not give high priority to inherent collision avoidance the design process, and would instead



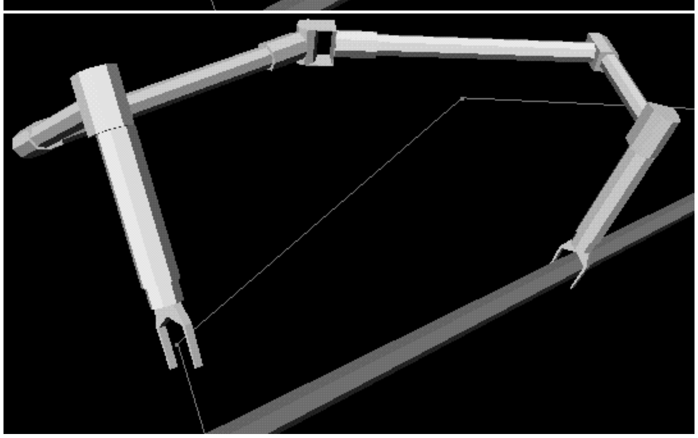
cfg$_m$ - lowest mass
mass - **12.2** kg
energy - **255** J
time - **77.2** s
key differences:

- small-diameter links are lightweight



cfg$_e$ - lowest energy consumption
mass - **12.3** kg
energy - **221** J
time - **84.7** s
key differences:

- lower velocity and acceleration parameters than cfg_m (and cfg_t) lead to lower energy use



cfg$_t$ - lowest completion time
mass - **18.6** kg
energy - **1112** J
time - **55.2** s
key differences:

- high velocity and acceleration parameters
- powerful wrist actuators enable higher accelerations
- larger-diameter links provide stiffness for high accelerations

Figure 5.39: Walkers with best mass, energy consumption, and task completion time

The three feasible configurations with lowest mass, energy consumption, and task completion time are shown from top to bottom, respectively. All three configurations share the same kinematic structure (7 DOFs total) than inherently minimizes self-collisions. For each configuration, its mass, energy, time and the features leading to high performance are shown. The key difference between *cfg*$_m$ and *cfg*$_e$ (which are generally very similar) is that *cfg*$_e$ has a significantly lower angular acceleration parameter and slightly lower linear velocity and acceleration parameters. *cfg*$_t$ has significantly higher velocity and acceleration parameters which lead to lower task completion time, but it pays a price for this performance by having heavier actuators and links as well as much higher energy consumption.

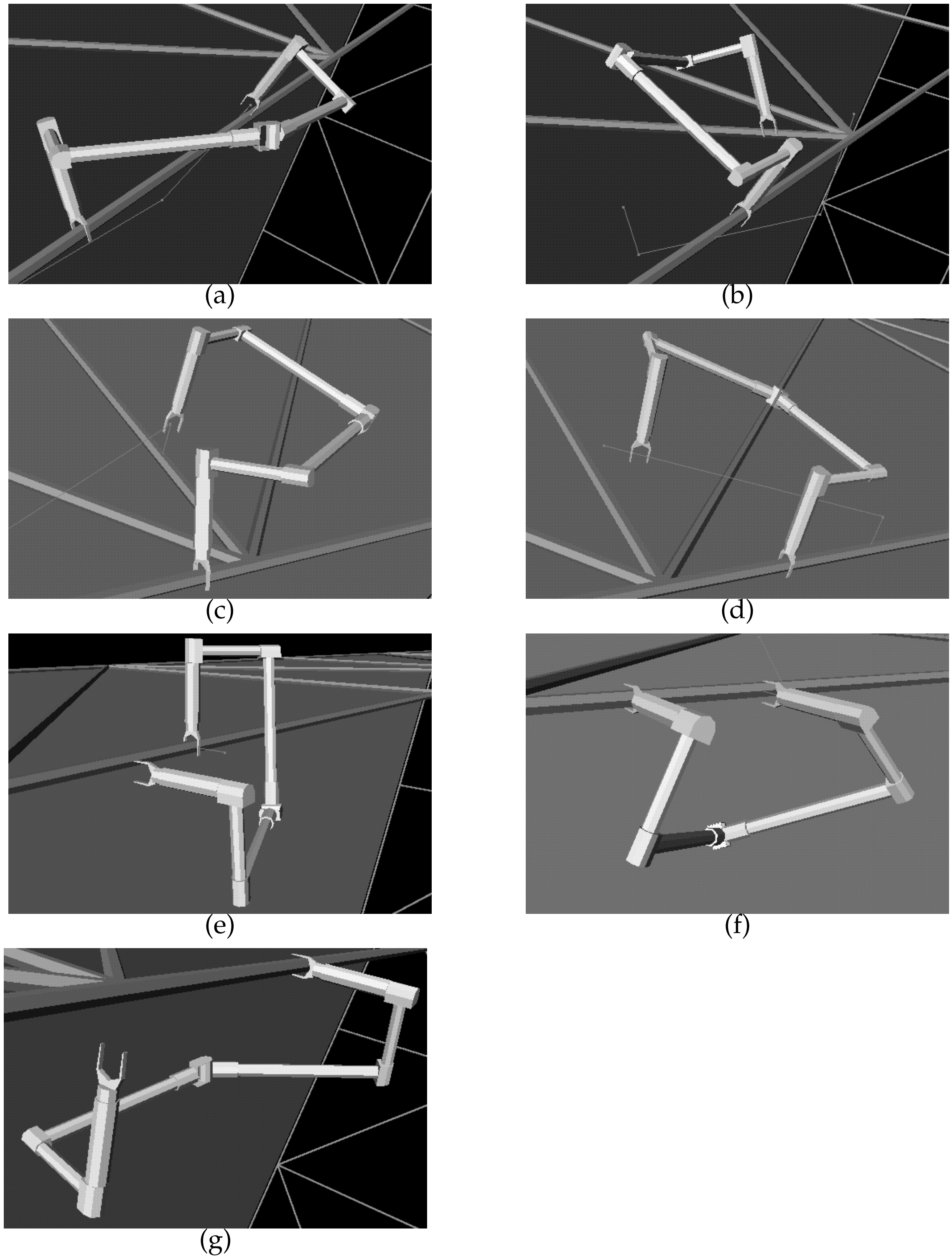


Figure 5.40: Synthesized walker executing inspection task

(a), (b) - walking along the first truss segment

(c) - transitioning to the second segment

(d) - transitioning to the third segment

(e) starting the rotation about the truss segment

(f) completing the rotation

(g) moving the end effector to inspection location

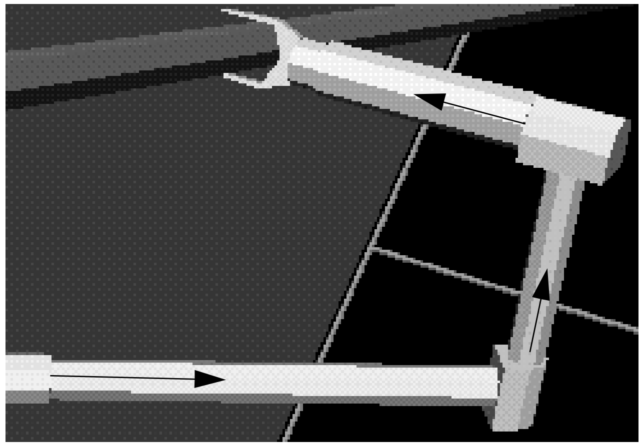


Figure 5.41: Close-up of walker wrist
The wrist structure present in the evolved walkers allows full rotation about each of the three wrist axes without any self-collisions. The joint axes are indicated by the arrows.

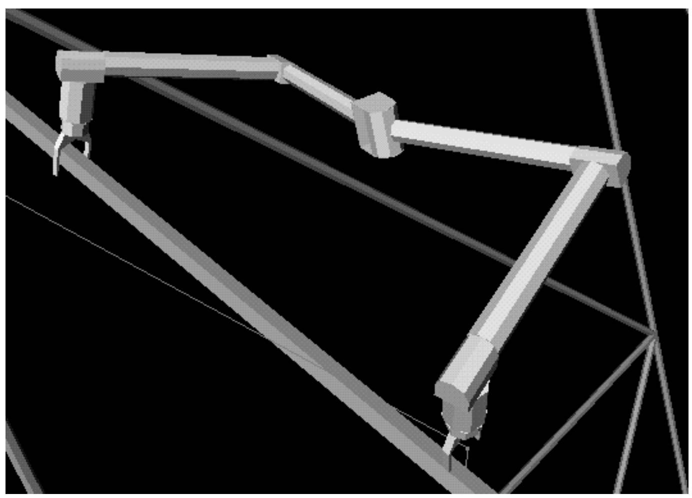


Figure 5.42: Most efficient walker from second trial
The robots generated in the second trial share the same kinematic structure as those from the first, but have a different ordering of joint modules in the configuration graph.

plan to use an appropriate controller to perform collision avoidance.

To see if these results were repeatable, or if any other unanticipated configurations could be synthesized, a second trial was conducted. As can be seen from the configuration in Figure 5.42, the robots synthesized in the second trial are similar to those from the first. While the second set of robots have the same collision-minimizing kinematic configuration as the first, it is worth noting that the configuration graphs are in fact different: whereas in the first experiment each limb consisted of an `inlineRevolute2` followed by two `rightAngleJoints`, the configurations in this experiment reversed the order of the arm modules with the two `rightAngleJoints` coming first followed by the `inlineRevolute2`. The performance measurements for the best feasible configurations are:

Table 5.16: Performance of best walkers from second trial

	mass	energy	time
best mass	13.7 kg	482 J	69.4 s
best energy	15.6 kg	263 J	87.3 s
best time	15.3 kg	917 J	54.9 s

As in the first trial, the synthesizer ran for 13 hours; however only 35,000 evaluations, rather than 52,000, were performed (due to fewer available computers for evaluation) and thus slightly lower robot performance in terms of mass and energy is not surprising. Note that the fastest configuration (at 54.9s) was slightly faster than in the first trial (55.2s) and is both lighter and more energy efficient.

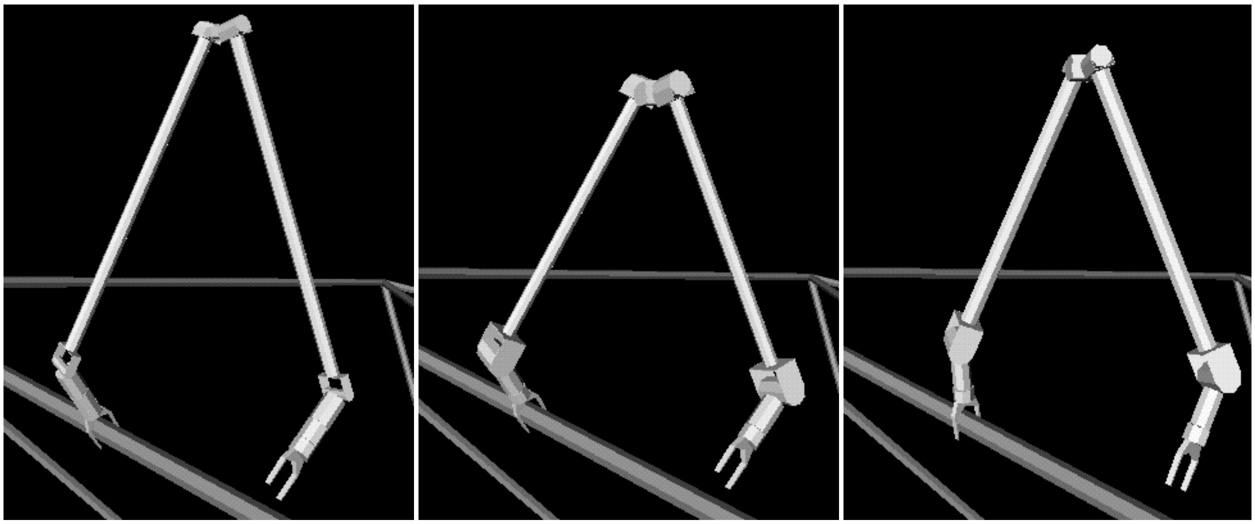
It seems likely that a more efficient walker can be synthesized if the only requirement is to be able to walk along truss segments and not transition to the back side of the truss. We can expect the robots synthesized only for walking to be better at walking than the robots we have seen so far, but perhaps be unable to perform transitions. To test this hypothesis, the synthesizer was re-run with a modified task specification that did not include any transitions between segments. The robots synthesized for this task, shown in Figure 5.43, have a markedly different structure than those evolved for the full inspection task. The three joints at the intersection of the two limbs are similar to human hips, though they have one less degree of freedom and cannot independently lift a leg without changing its orientation, while the wrists each have two intersecting joint axes. As detailed in Table 5.17, the lightest, most energy-efficient, and fastest robots have significantly better performance in at least one metric each than their counterparts evolved for the full inspection task. (The robots evolved for the full inspection task were re-evaluated on the walking-only task to produce the performance measurements in the table.) When the robots evolved only for walking are evaluated on the full task, they all have self-collision prob-

Metric	Task	Mass	Energy	Time
Lowest mass	complete inspection task	12.18 kg	98 J	30.5 s
	walking only	6.3 kg	78 J	40.5 s
Lowest energy	complete inspection task	12.3 kg	82 J	33.7 s
	walking only	12.3 kg	38 J	46.3 s
Lowest time	complete inspection task	18.6 kg	468 J	19.8 s
	walking only	14 kg	479 J	19.6 s

Table 5.17: Comparison of robots for walking-only task

The robots evolved for the walking-only task performed significantly better in one metric (shaded table cells) when compared to those evolved for the full inspection task.

lems at the wrist when rotating about the truss segment (Figure 5.44). This demonstrates a property of task-specific synthesis that can be both a strength and a weakness: by definition, the robots are specialized for the task and, while they may perform quite well at



Lowest mass	Lowest energy	Lowest time
Mass: **6.3** kg	Mass: **12.3** kg	Mass: **14** kg
Energy: **78** J	Energy: **38** J	Energy: **479** J
Time: **40.5** s	Time: **46.3** s	Time: **19.6** s

Figure 5.43: Best robots evolved for walking-only task
The robots that were evolved only for walking (without the transition or inspection motions) do not have the collision-minimizing wrist assembly synthesized in earlier runs

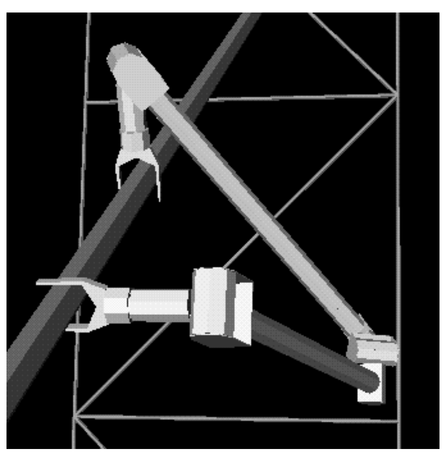


Figure 5.44: Link interference with walking-only robot
When evaluated on the full inspection task, the robots evolved for the walking-only task had self-collisions at the wrist when rotating about the truss segment. (Compare to the configuration in Figure 5.40e, which has the wrist assembly that avoids self-collisions)

Metric	Metrics used	Mass	Energy	Time
Lowest mass	all	12.18 kg	255 J	77.2 s
	mass only	14.4 kg	569 J	73.6 s
Lowest energy	all	12.3 kg	221 J	84.7 s
	energy only	19.6 kg	537 J	87.6 s
Lowest time	all	18.6 kg	1112 J	55.2 s
	time only	24.5 kg	1506 J	60.5 s

Table 5.18: Comparison of walkers optimized for single vs. multiple metrics
Starting from the same set of feasible configurations, the synthesizer produced better-performing robots when simultaneously optimizing mass, energy, and time, than when optimizing each metric independently.

---

the desired task, slightly different tasks may present substantial problems for the robots. For this reason, it is crucial to ensure that the representative task used during the evaluation process is in fact representative of the operations that will be encountered in the actual application.

In the experiments presented thus far (both in this and previous sections), we have examined the impact of many factors on the synthesis process, including the module database, kernel configurations, crossover operator probabilities, selection algorithms, and task description. One other factor that influences the results of the synthesis process is the set of metrics used to measure performance. In all of the experiments presented earlier that used requirement prioritization, there were multiple competing metrics in the final requirement group. Do these work against each other, resulting in configurations that are a compromise between the different metrics? Or do they have a beneficial effect on robot performance by pulling the synthesizer in different directions so that local minima in a single metric can be avoided? To answer these questions, the synthesizer was run three times, once for each metric in the final requirement group (mass, energy, and time). Each trial began with the population from the beginning of the final requirement group of the first experiment and only performed optimization of the final requirement group; however, instead of three metrics, the final requirement group contained only a single metric in each of the three trials (mass, energy, or time).

Even though the single-metric experiments were run for 40,000 evaluations, the robots with lowest mass, energy, and time performed significantly worse than the best from the original, three-metric experiment (Table 5.18). It thus appears that multiple metrics may improve the synthesizer's performance rather than degrading it. One potential reason for this surprising result is that multiple metrics lead the synthesizer in different directions, thus improving the synthesizer's ability to escaping local optima with respect to any single metric. When optimizing multiple metrics, the feasibly-optimal set usually contains many configurations with diverse properties, which prevents many good configurations from being deleted from the population and which provides a broad range of genetic material for creating new configurations. In contrast, when optimizing a single metric the feasibly-optimal set will contain configurations that have equal performance in the single metric and are thus not likely to be very diverse. This restricts the diversity of genetic material in the population, and when coupled with the fact that the synthesizer is always trying to move the search in the same direction (towards better values in the sole metric) this may lead to an increased likelihood of becoming trapped in a local minima. While it would be difficult to experimentally verify whether this hypothesis is correct, it seems to be the most likely mechanism that would produce the substantial and repeated performance differences between the single- and multiple-metric experiments.

5.6.1 Summary and discussion

The synthesis experiments presented in this section demonstrated synthesis of kinematic, dynamic, actuator, structural, and controller properties for a mobile robot tasked with walking and maneuvering on a truss structure in zero-gravity. The robots synthesized for this task possessed an unexpected kinematic structure that inherently minimized self-collisions and allowed each wrist degree of freedom to move through a full revolution. This inherent minimization of self-collisions make the control problem easier and allowed a controller with no explicit collision-avoidance capabilities to control the robots as they executed the task.

When the task was restricted to contain only the walking phase and no other maneuvering, the synthesizer was able to produce robots that were significantly better-optimized for walking than the robots evolved for the full task; however, the walking-only robots had problems with self-collisions when later evaluated on the full task. While this underscored the utility of the robots that had inherent self-collision minimization, it also demonstrated the importance of ensuring that the task used for evaluating configurations is actually representative of the end application, as factors that are not addressed in evaluation may present problems in practice. Additionally, this experiment demonstrated the importance of the controller in determining robot performance: while the collision-minimizing robots were able to complete the full task without collisions, it is likely that with a better controller the walking-only robots could also successfully complete the task. The expected continuation of increasing computing power should make it possible to include better controllers (possibly including deliberative planning and re-planning to account for errors in plan execution) in the synthesis process, giving better predictions of a robot's optimum performance. (Potential improvements in control and planning for use in automated synthesis are discussed in the next chapter.)

Finally, the experiments that compared optimization of metrics independently versus simultaneously indicate that the synthesizer's performance is improved by including multiple competing metrics. Multiple metrics may actually improve the synthesizer's robustness by preserving a greater diversity of genetic material in the population and by guiding the search process to a new region of the design space when a local minima in one metric is reached. This mechanism has not been experimentally verified to be the true cause for better performance when optimizing multiple metrics than when optimizing a single metric, but in any case there seems to be a significant improvement in the quality of synthesis results when multiple competing metrics are used.

5.7 Discussion

5.7.1 Experiment summary and perspective

In this chapter, Darwin2K was applied to the design of six robots: the free-flyer, the Space Shuttle waterproofing manipulator (for two versions of the task), the material handler, the antenna-pointing mechanism, and the truss walker. These experiments demonstrated the breadth of Darwin2K's applicability, showing how task-specific simulation needs can be integrated with the system's existing capabilities to produce relevant evaluation methods for each task. The experiment descriptions also documented the process of task specification including selecting of metrics, constructing a representative task for evaluation, and incorporating human knowledge through the module database and kernel configurations. Several experiments explored the sensitivity of the synthesizer to changes in the module database, kernel configuration, crossover operators, selection algorithms, and evaluation metrics.

The free-flyer and walker experiments used dynamic simulation for the evaluation process, a significant improvement in simulation capability for robotic synthesis. Many of the tasks included independent optimization and synthesis of kinematics, actuator selection, and structural properties as well as optimization of some control parameters. In some experiments, the robots had evolved to include properties that are familiar to a robot designer: the free-flyer had manipulators whose upper- and fore-arms were similar in length, and whose shoulders had three intersecting joint axes; the material handlers were similar to those created by human designers; and the antenna-pointing mechanism evolved for tracking low-elevation targets was kinematically similar to a manually-generated mechanism designed for the same purpose. However, the walker experiments demonstrated synthesis of unexpected yet well-optimized and potentially novel configurations, demonstrating the utility of automated synthesis for tasks that are not well-understood on the basis of previous experience within the design team. Significantly, the same robot representation -- the parameterized module configuration graph -- was used for all experiments despite the varying robot topologies and requirements for different tasks.

Some rules of thumb for applying Darwin2K to future synthesis problems can be learned from the experiments in this chapter. Human knowledge can play an important role in determining the quality of synthesis results. Including useful topologies and modules can greatly improve the performance of the synthesized robots, while including suboptimal modules or topological features can lead to poor performance. The use of the commonality-preserving crossover operator with subgraph preservation can speed convergence towards good solutions by exploiting features that have been beneficial in the past; however, these methods necessarily restrict the search space and increase the synthesizer's susceptibility to local optima. These methods should be used in moderation unless computer time is at a premium, in which case quality of results can be sacrificed to improve synthesis speed by increasing the rate of CPCO and subgraph preservation. The Configuration Decision Function and Requirement Prioritization are both significantly better than the weighted sum for selecting configurations for reproduction and deletion on the basis of multiple performance metrics, in terms of synthesizer performance, quality of the synthesized result, and relevance to the task at hand. Finally, the synthesizer was able to generate a wide range of feasible configurations that make different trade-offs between multiple metrics; the designer must choose among these based on the requirements of the task.

The experiments suggest several areas for future improvement. Changes that could be made to the synthesizer include automatically setting the probabilities for subgraph preservation and for the commonality-preserving crossover operator, and the use of multiple sub-populations to improve repeatability by performing more exploration early in the synthesis process. The synthesis process would also benefit from the use of optimal, or at least deliberative, planning and control, as local controllers inevitably introduce biases into the synthesis process by not making best use of each robot. Improved dynamic simulation that included closed kinematic chains and simple terrain interaction models would extend Darwin2K's utility for mobile robot synthesis problems. Finally, a high-level task scripting language would allow task-specific simulators to be more easily constructed, and could lead the way for including the high-level task specification in the synthesis process.

5.7.2 The role of Darwin2K in the design process

In the experiments presented in this chapter, Darwin2K performed a significant portion of the design process: configuration design. However, there are other phases of design that occur both before and after configuration design: conceptual design, and detailed design. Conceptual design decides the general form of the machine (or machines) that will address a task: Will a mobile robot or fixed-base manipulator be used? Will there be one robot, or several? Should the mobile robot walk, roll, or fly? For each of the experiments, conceptual design was implicitly performed in the process of describing the task, kernel configurations, and configuration filters: in the free-flyer experiments, we limited the search to two-armed, free-flying robots; the truss inspection walker experiments assumed symmetric walkers with 7 degrees of freedom; the material handler experiments assumed a wheeled base; and so on. In each of these cases, the highest-level designs decisions had already been made, thus limiting the design space. For example, the material handler could have used a legged base, and the truss inspection task could have been accomplished by a free-flying camera or some sort of articulated, wheeled robot.

The effort required to specify tasks and implement task-specific planning or control algorithms is the main reason Darwin2K did not address high-level conceptual design. The addition of flexible planners, or of motion plan synthesis performed simultaneously with configuration synthesis, would allow Darwin2K to perform conceptual design, most likely at a substantial increase in computational cost. Improved planning capabilities would also reduce the work required to specify a detailed task description. Even with these proposed improvements, it may not make sense to give Darwin2K an unrestricted (or very loosely-restricted) design space; rather, it would probably be more useful to apply Darwin2K to each conceptual design sequentially. For example, wheeled, flying, and walking robots for the truss inspection task would have drastically different topological and parametric features, and it is not likely that exchanging information between them would prove useful. Furthermore, simultaneous exploration of all three schemes for mobility would likely result in convergence upon a single modality and would not allow each method to be explored and optimized to the fullest extent. In contrast, using Darwin2K to explore, detail, and optimize each modality independently would provide the designer with a realistic assessment of the performance and trade-offs possible with each design. This approach simultaneously allows a more informed conceptual design decision to be made, and results in a complete configuration description that is ready for detailed design.

Partitioning of the design space is also useful at a finer level: for example, the experiments that explored the impact of module and kernel choice indicated that manipulators with prismatic joints were better-suited for the task than manipulators with only revolute joints. Using Darwin2K in a sequential fashion starting from different configuration alternatives allows each alternative to be explored in-depth and optimized. In contrast, manual design methods often make mid-level configuration decisions (e.g. revolute or prismatic joints, number of degrees of freedom, etc.) before each alternative can be extensively evaluated. When a robot is being designed manually, the mid-level configuration decisions may often be made when each alternative is at the "stick figure" level of detail. Darwin2K provides much more information about the pros and cons of each alternative and provides a significantly more detailed description of the configuration, allowing the designer to make more appropriate mid-level configuration decisions and thus avoid surprises down the road as the chosen configuration is fleshed out. This difference between manual and automated design is partially due to an interesting and fundamental difference in design methodology: manual design starts with vague design concepts and proceeds to refine and detail each concept, discarding alternatives before they are fully specified, while automated approaches such as Darwin2K use the same level of detail for every design and seek to improve performance by changing attributes of the design.

The development, testing, and characterization of Darwin2K focused on a highly-automated manner of use that did not take advantage of human interaction during the synthesis process. While this resulted in a capable synthesis system, it is likely that the computational resources required for synthesis can be substantially reduced by including more human interaction during synthesis. One mode of interaction is to use Darwin2K in an iterative manner: the best few configurations from one run of the synthesizer can be used as the kernel configurations for another run. The designer can set the `const`-flags for some of the parameters and attachments of the new kernel configurations in order to limit the search space, thus allowing Darwin2K to be used in a coarse-to-fine manner. Another mode of interaction would be the "Hand of God", which requires some modification to Darwin2K: the designer could interactively view robots during the synthesis process and modify them, so that easily-remedied design flaws could be quickly corrected. This method would likely entail another process that queries the Synthesis Engine for the best configurations in the population (perhaps showing the designer a scatter plot of the feasibly-optimal set, similar to Figure 5.9), then shows the designer a simulation of the chosen robot as it executes the task. The designer could then modify the robot (either by editing the text representation of the PMCG or through an interactive tool) and then return the robot to the Synthesis Engine, which would queue the robot for evaluation and then introduce the configuration into the population. This would allow the designer to influence the exploration of the design space and would probably be most useful in helping the synthesizer escape local minima.

5.7.3 Conclusion

The results presented in this chapter encompass 146 trials of the synthesizer and entailed generating and simulating over five million robot configurations. The range of experiments underscores Darwin2K's applicability, and demonstrates the system's ability to synthesize many key properties including kinematics, dynamics, actuator selection, structural features, and control parameters. While the experiments were predominantly successful, they revealed some limitations and areas for caution and motivated ideas for future work. The next chapter concludes this dissertation and steps back to assess the lessons learned and contributions made during the course of this work, and enumerates key challenges for future work in automated synthesis for robotics.

6 Conclusion

This thesis presented Darwin2K, a practical, widely-applicable, and extensible system for automated robot configuration synthesis. This thesis makes numerous contributions to robot synthesis, including improved methods for multi-objective optimization, an extensible architecture for robot synthesis systems, new simulation and synthesis capabilities, a new representation for robot synthesis, and demonstration of synthesis for a wide range of realistic robot tasks.

Several features of Darwin2K contribute to its wide applicability. Darwin2K's synthesis algorithm is independent of the task and the type of robot, allowing the system to be extended to address a wide range of synthesis tasks. The Parametrized Module Configuration Graph allows representation of a wide range of robots, including modular and monolithic robots, mobile robots, and robots with multiple or bifurcated manipulators. Finally, Darwin2K's extensible system architecture enables novel synthesis tasks to be addressed while maximizing use of existing simulation and synthesis capabilities.

Darwin2K includes a toolkit of simulation and analysis algorithms which are useful for many synthesis tasks including dynamic simulation, estimation of link deflection, PID and Jacobian-based controllers, collision detection, and representations for trajectories and payloads. A method for automatically generating dynamic models of robots is presented, allowing dynamic simulation to be incorporated into the synthesis process. When combined with Darwin2K's synthesis algorithm, these capabilities allow synthesis and optimization of robot kinematics, dynamics, structural properties, actuators, and task parameters. The system's evolutionary algorithm can effectively optimize multiple objective functions in a task-relevant manner, and can generate a range of solutions that make different trade-offs between metrics. Two new selection methods for multi-objective optimization were presented: the Configuration Decision Function and Requirement Prioritization. These methods fill different niches (rank or tournament selection, and fitness-proportionate selection, respectively) and are applicable to other evolutionary optimization domains.

Finally, the capabilities of Darwin2K were demonstrated and characterized through nearly one-hundred and fifty experiments synthesizing fixed-base and mobile robots, with single and multiple manipulators, for a range of tasks. The breadth of experiments and the detail of synthesized results demonstrate a substantial improvement over previous systems for automated robot configuration synthesis.

6.1 Contributions

A new representation for robot configurations

The Parameterized Module Configuration Graph (PMCG) combines the advantages of parametric and modular representations while eliminating their drawbacks. The

PMCG can represent robots that have fixed or mobile bases, that are modular or monolithic in nature, and which have single, multiple, or branching serial chains. The PMCG easily allows human knowledge about robot topology to be incorporated (which can significantly improve synthesis quality while reducing computation time) and allows for preservation of symmetry in configurations.

The use of parameterized rather than fixed modules allows a single representation to be used to efficiently synthesize both modular and non-modular robots. While it is possible to use fixed modules to synthesize robots that will be constructed in a monolithic (rather than modular) manner, the inclusion of parameters in the module representation has several significant advantages over fixed modules:

- **Parameters can be varied independently, allowing a wide range of module properties.** A single parameterized module with five 3-bit parameters can represent the same space of variations as 32,768 fixed modules.
- **When using a parameterized representation, the optimization process can change a specific property of a configuration by changing a single parameter.** In contrast, when using fixed modules, useful information will be lost as the optimizer attempts to make a change to a single property by replacing a module with a different one (which may have drastically different properties and may be of a different type entirely).

A modular representation has important advantages over purely parametric representations, as well. More detailed geometric and component information can be encapsulated within each module, enabling more accurate simulation and detailed specification of the final design. Since modules are self-contained software objects and present a consistent interface to the system, a module's parameters can represent arbitrary properties and new modules types can be added to the system without any impact on the synthesis algorithm. Finally modules can represent subsystems of arbitrary complexity, from a link with no moving parts to an entire mobile robot, and provide a simpler means of varying robot topology than purely parametric representations.

An Extensible Architecture for Configuration Synthesis

The software architecture used in Darwin2K employs a rigorous application of object-oriented programming to enable new capabilities to be added while maximizing re-use of existing software components. The architecture allows task-specific modules, metrics, simulation algorithms, controllers, and other evaluation methods to be added without requiring any modification of the synthesis algorithm while encouraging re-use of Darwin2K's toolkit of software components. The result is a synthesis system that is significantly more extensible and widely-applicable than previous approaches.

Task-Relevant Optimization of Multiple Objective Functions

A significant part of this thesis concentrated on developing an effective optimization algorithm for constrained, multi-objective synthesis problems. A driving factor was the realization that the synthesizer should account for the significance of different metrics in a manner that is relevant to the task at hand. This led to the development of two new selection algorithms: the Configuration Decision Function (CDF) and Requirement Prioritization (RP). Both methods consider the significance of each metric in a task-relevant way, e.g. selecting configurations on the basis of power consumption is only meaningful if the configurations can complete the task at hand. Setting synthesizer parameters for these methods is significantly more intuitive than selecting weights for scalarization-based methods such as the weighted sum, and these methods are effective at satisfying a number of competing performance requirements while simultaneously generating a range of trade-off configurations for multiple open-ended objective functions. Both methods are applicable to other evolutionary optimization domains, and address different niches: the CDF can be used with algorithms based on tournament or rank selection, while RP is formulated for fitness-proportionate selection.

Two other important aspects of the synthesis algorithm are elitism and diversity preservation. The elitist algorithm for multiple metrics augments the traditional notion of the Pareto-optimal set with a task-specific feasibility criteria, resulting in an elitist method that accounts for the task-relevance of different metrics. This algorithm prevents good solutions from being lost due to the probabilistic nature of evolutionary algorithms while limiting the elite set to include only those configurations that are non-inferior with respect to a subset of the metrics and that differ in ways that are significant with respect to the task. The algorithm for diversity preservation ensures that every solution in the population is unique, and is particularly useful for steady-state evolutionary algorithms. Evolutionary approaches require a diverse population in order to avoid local optima in the search space; however, they also necessarily converge on a solution as optimization proceeds. Darwin2K's diversity preservation algorithm eliminates premature convergence of the population while also eliminating wasted computation time spent on redundant evaluations.

Commonality-Preservation for Configuration Graphs

Recent research suggests that it is beneficial to preserve common features when performing crossover between two parent solutions that have been selected for their high fitness. While the crossover operators used for in fixed-length genetic algorithms inherently preserve diversity, those for trees and graphs do not. Two related methods of commonality-preservation used in Darwin2K are the commonality-preserving crossover operator, which preserves the largest common subgraph of two parent configurations in their offspring, and the subgraph-preservation operation, which prevents the common subgraph from being disturbed in future recombinations. These methods trade off exploitation of topological features known to perform well against exploration of potentially better topologies, providing a means of adjusting the synthesizer's behavior to favor rapid generation of feasible configurations or slower but more robust generation of higher-quality configurations.

Forward Dynamic Simulation for Configuration Synthesis

Darwin2K can optionally use a dynamic (rather than kinematic) simulation when evaluating configurations. This is the first time dynamic simulation has been included in the synthesis process of an automated configuration tool. Kinematic simulation is acceptable for applications in which actuators will not be operated at their torque limits, and (in the case of robots with non-fixed bases) when manipulation forces are not large enough to generate significant reaction forces. Dynamic simulation is required for free-flying robots, for robots whose inertial forces are significant enough to cause base motion, and for robots (such as hydraulic machines used in construction) that are controlled in joint-space with actuators operated at their torque limits for large portions of the task -- in short, when the robot's motion can not be computed on the basis of kinematics and controller commands. Forward dynamic simulation of a robot gives the capability of modeling the behavioral effects of actuator saturation and reaction forces due to manipulation. Darwin2K can automatically derive the equations of motion for a robot -- including multiple or branching manipulators and free-flying bases -- and use them to compute the motions a robot due to applied forces from actuators, payloads, and tool reactions. This extends Darwin2K's applicability to include tasks requiring free-flying bases, frequent or constant actuator saturation, and force-based (rather than position- or velocity-based) control.

Synthesis and Optimization of Dynamic, and Structural Parameters

Kinematic parameters are only one aspect of a robot's configuration; structural properties and dynamics are also key properties that define a robot and determine its performance. Both of these properties are included in Darwin2K's robot representation and since Darwin2K's simulation algorithms account for their impact on robot performance, these parameters can be optimized by the synthesis algorithm. Previous synthesis approaches have ignored the effects of link material and cross-section on stiffness, and thus cannot meaningfully optimize the dynamic properties of links (which are also dependent on material and cross-section). Using links with fixed cross-sections without regard to their stiffness implies either that all link and joint modules are overdesigned and thus more massive than necessary, or leads to underestimates of robot mass and actuator torques since some links may require additional mass to increase strength and stiffness to adequate levels. Including the impact of link cross-section and material on both dynamics and stiffness yields more accurate estimation (and better optimization) of total system mass and actuator requirements.

Independent Synthesis and Optimization of Actuators and Components

Darwin2K can vary motor, gearbox, and other component selections independently of other module properties due to the parameterized module representation. Previous approaches have relied on fixed modules; if an actuator was undersized, it could be changed only by replacing the entire module. This is disruptive, since replacement of a module may change many properties of the robot that are already well-optimized (such as joint type and orientation, link length, and module type). Since actuator selection for a module can be an independent parameter in Darwin2K, a joint's actuator can be changed without making disruptive modifications to a configuration. Combined with Darwin2K's dynamic simulation and metrics for actuator saturation and power, this leads to actuators that are well-optimized with respect to a task's mass, power, force, and speed requirements.

Demonstration

This thesis has demonstrated automated synthesis over a significantly broader range of application than previous synthesis systems. This provides a validation of the system's novel aspects, including the parameterized module configuration graph representation, extensible framework, optimization algorithm, dynamic simulation, and evaluation methods. The experiments demonstrated synthesis of fixed-base and mobile manipulators, multiple manipulators, and a walking robot, and included generation of robot kinematics, dynamics, structural properties, actuator selection, and control and task parameters. The breadth and depth of these examples demonstrate a significant advance in the state of the art in automated synthesis of robots.

6.2 Lessons Learned

Many useful and important pieces of knowledge became apparent during the development and application of the system presented in this thesis. These lessons may prove useful to future researchers pursuing automated synthesis.

Task Specification

It is crucial to capture task-specific requirements in a meaningful way. The better a designer can communicate the constraints and needs of a task to the synthesizer, the more relevant the synthesized robots will be. For example, using the weighted sum for selection resulted in many non-feasible configurations being selected for reproduction in preference to feasible configurations because the weighted sum failed to encode the fact that meeting the task constraints (such as task completion or lack of collisions) was more important than task completion time or energy usage. Additionally, it is useful to capture the task objectives as directly as possible: metrics such as peak joint torque or velocity do not actually measure a robot's performance on the task, and are inferior to metrics like task completion time or actuator saturation which have more direct bearing on the robot's performance. Evolutionary algorithms are notorious for their ability to find loopholes in the problem specification; the likelihood of this is reduced when the key requirements of the task are encoded in the synthesizer's goals, constraints, and evaluation methods. One question the designer should ask when selecting metrics or properties to optimize is "What trade-offs do these metrics/properties affect?" The material handler synthesis task in Section 5.4 is a good example: since link deflection was not accounted for, there was no trade-off against minimizing link cross section and thus the synthesizer had no way of meaningfully optimizing link cross section. Fortunately, loopholes in the task description quickly become obvious after several synthesis runs, as the synthesized robots inevitably exploit them.

Evaluation

The controller used during simulation has a substantial impact on robot performance and can induce biases in the synthesis process. For example, the robots evolved for truss walking evolved to include a kinematic structure that inherently minimized self-collisions, but it is unlikely that this feature would have evolved if the robots' controller intentionally avoided collisions. Ideally, a globally-optimal controller would be used in the evaluation process, thus revealing a robot's best possible performance; currently, this is not yet within the reach of most organizations due to the computational resources required. While making individual evaluations more expensive, it is likely that the use of optimal controllers will significantly reduce the number of evaluations required to generate feasible configurations. Until it becomes tractable to use globally-optimal controllers during the synthesis process, it is useful to let the synthesizer optimize any controller parameters (such as velocity, acceleration, or other gains) that are not dictated by the task requirements since different robots are likely to require different values for these parameters. Additionally, the synthesizer can often find useful performance trade-offs (such as speed versus energy) if it is allowed to vary controller parameters.

Evaluation comprehensiveness and accuracy play important roles in determining how much confidence a designer has in the synthesized robot's performance. For example, if a mobile robot's interactions with terrain are not modeled very accurately, it is unlikely that the real robot will perform similarly to the simulated robot, and it is not clear that the synthesized robot will be better in reality than other designs that perform worse in simulation. This is a compelling reason for using simulation rather than heuristics to measure performance, for using dynamic simulation when evaluating free-flying robots, and for further improvements in simulation quality.

An important practical aspect of constructing a task-specific simulation is the cost of iteration when changing simulator or task properties. Darwin2K allows all evaluation parameters (such as trajectories, controller parameters, simulator time steps and error tolerances, payload geometry, and even the choice of controller and other evaluation components) to be specified in text files, thus making iterative changes to the simulator very easy. This is also very useful for robot properties: the text format for robots enables the designer to quickly construct and modify robots after viewing their performance in simulation. These two factors make Darwin2K's simulator valuable as a stand-alone design tool since design iterations can be performed quickly to give the designer a feel for the impact of robot properties on performance.

Finally, the use of flexible algorithms can greatly reduce the effort required to create a simulation while increasing the system's applicability. For example, the SRI controller can work with single, multiple, and branching serial chains of arbitrary degrees of freedom, and with free-flying or planar bases. This allowed the controller to be used for all of the tasks requiring kinematic simulation (since tasks were formulated as trajectory following) and, when augmented with the robot's dynamic model (see Section 4.4.4), to also be used for the free-flyer and truss walker tasks. No specialized controllers were needed for any of these tasks, thus reducing the effort required to construct a task-specific simulator.

System Characteristics

Though not initially a focus of this research, it quickly became apparent that extensibility is a factor that limits the applicability of a synthesis tool. No single task representation short of a well-featured language is adequate for all robot tasks; this realization led to the development Darwin2K's extensible architecture, which isolates task details from the synthesizer and allows new tasks to be simulated while taking advantage of existing software components (e.g. the collision detector, SRI controller, or numerical integrator). This also allows new capabilities to be incorporated into Darwin2K's library of software components in a regular way.

Based on experience with Darwin2K, it seems important for a synthesis system to easily allow the incorporation of human knowledge about potentially beneficial properties of the artifact being designed. In Darwin2K, this is accomplished by specifying partial or complete topologies in the kernel configurations, and selecting appropriate modules (and parameter ranges) for inclusion in the module database. Additionally, the properties of synthesized robots can be manually changed quite easily since each robot can be stored as a human-readable text file. Incorporating human knowledge into the design process by way of both constraints (such as the configuration filters and kernels with `const`-flags set for topologies) and primitives (module selections) can reduce synthesis time and improve the quality of synthesis results. Human guidance can, however, be a two-edged sword: if user-specified design constraints are unnecessary or harmful, the synthesizer will generate poorly-optimized configurations. The inclusion of complex modules (as in the manipulator synthesis trials in Section 5.2 that used the SCARA module) can also make the synthesizer more susceptible to local optima, since feasible but poorly-optimized topologies can be quickly discovered and focused upon. Complex modules should thus be included only when there is a good reason to suspect they will be useful or required for the task being addressed.

6.3 Future Directions

Simulating contact, terrain interaction, and closed chains

Darwin2K's applicability could be expanded by including simulation of contact, terrain interaction, and closed kinematic chains. The simulation of terrain interactions for wheeled vehicles with rigid and non-rigid suspensions is crucial in enabling complete synthesis of wheeled mobile robots. Synthesis of multi-legged walkers or climbers would be improved by the ability to simulate kinematic chains that are closed through contact with external bodies; the related ability to simulate chains that are closed through the robot mechanism itself would expand the types of mechanisms that Darwin2K could synthesize. Simulating contact is a more difficult but still tractable problem and would allow evaluation of running machines, grippers, and other mechanisms that interact with their environment in ways that are not always easily predictable.

Planning and control

The incorporation of optimal or deliberative planning and control will provide better estimates of performance when evaluating configurations in simulation. When using local control, measured robot performance depends heavily on initial conditions such as initial joint angles; more capable control methods would not be as sensitive to these conditions and could yield performance measurements that are closer to a robot's best capabilities. The biases in solution form introduced by local methods would also be greatly reduced by the use of deliberative or non-local planning and control. Another approach is to include plan generation in the synthesis process, probably combining elements of modular plan generation (as in [Farritor98]) and parametric optimization of task and control values (as was done in Darwin2K for controller parameters, and in [Kim93] and [Chocron97] for joint angles). Such an approach would eliminate the need for complex, globally optimal planning and would measure plan optimality according to the task's metrics, but would increase the size and complexity of the search space. This approach would also increase the flexibility of the synthesizer in deciding high-level task and configuration issues since details of task description could be varied. For example, the synthesizer might be able to vary controllers and trajectories, and choose a mobile robot, manipulator or even multiple cooperating robots based on the detailed task description. Both approaches -- better planning algorithms, and including plan generation in the synthesis process -- seem to have unique advantages and costs and warrant further investigation.

Task specification

While Darwin2K provides many useful components for building a relevant, task-specific simulator which can be used for automated and manual synthesis, creating the simulation can take anywhere from an hour to a week depending on how much task-specific coding is required. Much of this time could be eliminated by the development of a task scripting language for specifying how the simulation components interact. In addition to creating the language itself, this would require the interfaces of simulation components to be expanded to provide a regular way of querying states and accessing internal variables. A language for specifying the geometry of parameterized modules would completely eliminate the coding required for most task-specific modules, and the ability to import CAD descriptions for payloads and fixed modules would reduce task-specific coding time as well.

Expanded component modeling

Adding component models beyond motors, gearboxes, and materials will increase the level of detail of synthesized robots and will more accurately model their properties and performance. For mobile robots, power storage and generation subsystems such as batteries, solar panels, and generators can be a significant component of total system mass. With models for these components and appropriate metrics, Darwin2K could include optimization of major power system components in the synthesis process to reduce system mass or improve range and operating time. Another important capability is sensor configuration, which would be enabled by the inclusion of perception sensor models such as cameras and rangefinders. While the human designer would likely know the best type of vision sensor, sensor parameter selection (e.g. field of view) and placement could be optimized by modeling a sensor's visible field and including a metric for measuring sensor coverage of a desired area.

Richer representation for robots

As with expanding the types of components modeled by the synthesizer, improvements to the Parameterized Module Configuration Graph would increase the level of detail of synthesized configurations. For example, the detailed geometry at the junctions between modules could be synthesized if modules could query each other about their geometry; if one module knew that a module it was attached to had a circular connection surface of a certain diameter, the first module could create its geometric representation such that a smooth junction was produced. This could result in more accurate estimates of structural and inertial properties. Additionally, it would be useful to have parameters that were shared by modules, e.g. two different link modules in a manipulator that shared the same length or diameter parameter, or joint modules that shared actuator parameters to simplify the manufacturing process. Labelling the task parameters and allowing the number of task parameters to vary would enable simulation components to perform operations such as constructing joint-space trajectories (with differing numbers of via points) that are optimized along with each configuration.

Improving synthesis repeatability

The probabilistic nature of evolutionary algorithms can make it difficult to achieve repeatable results for some problems. While fairly repeatable results can be had for some problems (such as the material handler, truss walker, and high-level and prismatic Space Shuttle waterproofing manipulator experiments), other problems apparently have many more, or much deeper, local optima in the search space and results differed substantially in topology between different trials. The multi-stage nature of Requirement Prioritization may make multi-population or multiple-restart approaches effective at improving repeatability by exploring many different options early in the search process, before committing to optimization of a limited number of topologies.

Practical issues

Several practical improvements to Darwin2K will help bring it into more general use. A graphical configuration editor would make it easier to assemble configurations for simulator debugging, manual synthesis, or for use as kernels. Similarly, a graphical designer for interactively specifying task properties such as trajectories would reduce task specification time.

Finally, it is important to address the packaging, documentation, porting, and distribution of Darwin2K as a software system. Darwin2K comprises a significant amount of software, on the order of 60,000 lines of C++; it is important to make this publicly-available so that Darwin2K can be put into use, and so that future research in robot synthesis that advances beyond Darwin2K's abilities does not first require duplication of this effort.

These actions should help bring automated synthesis capabilities into the hands of robotics researchers and designers alike.

Future applications

The synthesis tasks addressed in this thesis represent only a small portion of the types of tasks for which robots can be automatically synthesized. The potential improvements and extensions to Darwin2K listed above will provide new opportunities for synthesis such as workcell configuration, cooperative multi-robot systems (including their behaviors), running machines, swimming robots and other submersibles, and multi-legged walkers. These exciting and challenging opportunities will be enabled by improvements to Darwin2K and will themselves provide motivation for new capabilities and directions.

Appendix A: OOP and Class Hierarchy

Darwin2K makes extensive use of object-oriented programming (OOP) to allow new software components to be added and interact with existing system components. Briefly, OOP is a programming methodology that relies heavily on abstraction and interfaces. Abstraction is the process of making a single atomic unit or concept out of numerous properties, so that the entire set of properties can be referred to and used as a unit. This is also the "black box" approach: the internals of each unit are hidden, and all interaction between units is through interfaces. Abstraction is also similar to naming: for example, instead of saying "animal with four legs, whiskers, a tail, and which eats mice and birds, and which has a specific weight and color", we can call the same object "cat" and thus avoid describing the details each time we wish to refer to it. In OOP, the notion of a description of an object (e.g. `cat`) is called a *class*. While `cat` describes a particular kind of animal, it is also a generalization: there are many sub-categories of cat, such as `Persian` or `Siamese` or `Tabby`. In C++, these are said to be *derived classes* of the *base class* `cat`, since they are specific types of `cat`. Similarly, the `cat` class might be derived from `mammal`, which might in turn be derived from `animal`, and so on. Each C++ class can contain a description (i.e. data, such as the cat's weight or color) as well as procedures (such as `eat` or `nap`). The descriptions are called *members* or *data members*; the procedures are called *member functions* or *methods*. When referring to a specific example of a class, such as your own pet cat named "Grouchy", you are referring to an *instance* of the class, or an object. Each instance contains specific values for the data members: the `cat` class description contains a member for weight, and the instance of cat named Grouchy has a weight of 9 lbs.

The "black box" nature OOP allows objects of various types to interact with each other, without having to know about the internals of all objects involved. In Darwin2K, there are classes for modules, for simulation components, for metrics, for evaluators, and for configuration filters, and so on. Each of these base classes defines a specific interface, though which all interactions take place. For example, when creating the physical representation of a robot, the configuration calls each `module`'s `createGeometry` method. The configuration does not need to know what the specific class of each module is or how each module's geometry is created. This allows the configuration to work with any type derived from the `module` base class. Similarly, the `metric` class defines a standard interface so that the synthesizer can work with any metric while remaining independent of the metric's internals.

To make it possible to specify which modules, metrics, evaluators, and other simulation components should be used for a particular synthesis run, all of these objects are derived from the `synObject` class. `synObject` is the base class for objects that are included in Darwin2K's object database, which allows objects of specific classes to be dynamically created by their name. Normally, when an object of a specific class is required, the class is specified in the source code and is thus fixed compile time:

```
massMetric *m = new massMetric;
```

This statement creates an object of type `massMetric` that will be referenced by `m`. How-

```
synObject ──┬── module ──────────┬── dofModule ──┬── baseModule ──── nonComponentBase
            ├── component        ├── linkModule  └── jointModule ──┬─ componentJoint
            ├── obstacle3        └── toolModule                    └─ nonComponentJoint
            ├── evComponent ─────┬── DEsolver
            ├── cdObject         ├── controller
            ├── evaluator        └── genericPath
            ├── metric ──────────── stateDependentMetric
            └── pmComponent ─────── cfgFilter
```

Figure A.1: Hierarchy of base classes for database objects
Shown above are the base classes for objects that can be added to Darwin2K's object database. Any objects (including task-specific objects) that are derived from these objects can be included and configured at runtime through Darwin2K's initialization files. Note that all of these classes only serve as templates, and cannot be instantiated as objects.

---

ever, in practice it is inconvenient to require all objects to be specified at compile time: in this case, if the designer decided that a different metric was required then the source code would have to be changed, and the libraries and executables would have to be rebuilt. To make it easier for the designer to select which objects are used and initialize them, classes derived from the `synObject` class can be instantiated and initialized through text files:

```
char objectType[80] = "massMetric";
float min = 10.0;
float max = 20.0;
```

These statements, while similar in syntax to C, are included in the synthesizer's initialization file and indicate that a `massMetric` object should be created, and that the metric's minimum value should be 10.0 and its maximum value should be 20.0. The modules in the module database are specified similarly: the modules desired for a given synthesis run are listed and values for their properties are given. Simulation components, evaluators, and other classes can similarly by instantiated and initialized. The main method (member function) defined by the `synObject` class is `className`, which returns a string containing the name of the class. The object database contains a list of all classes, and by comparing the string returned by each class's `className` function to the string specified by the user, objects of appropriate type can be dynamically created. Figure A.1 shows the class hierarchy for base classes that can be used with Darwin2K's object database; all are derived from the `synObject` class and some will be briefly described in the following sections.

The module class is used to represent parameterized modules. Some of the important methods for modules are:

- `createGeometry` - creates polyhedral description of module
- `numParams - returns the number of parameters`
- `numConnectors - returns the number of connectors`
- `requiresContext` - returns 1 if the module requires a component context
- `computeDeflections` - computes the deflection of the module's

Figure A.2: `module` class hierarchy
Modules listed in bold can be instantiated, while those in plain type are only base classes. Not all of the modules were used in the experiments presented.

---

parts given applied forces and moments

Each specific module type can define its own replacements for these functions; these replacements will be called instead of the generic version. Figure A.2 shows the class hierarchy for Darwin2K's modules. Appendix B gives the parameters and brief descriptions of the modules that were used in the experiments. The `component` class is used by modules which require descriptions of motors, gearboxes, or materials.

The `evComponent` class is used to encapsulate modular simulation capabilities; the class hierarchy is shown in Figure A.3. The class's main functions include:

```
evComponent ───┬── collisionDetector
               ├── motionPlanner
               ├── reactionForceCalculator
               ├── payload ──────────────── panelSection
               ├── controller ───────────── sriController ──── ffController
               │                            pidController
               ├── genericPath ──────────── path ─────────┬─── relativePath
               │                                          ├─── antennaPath
               │                                          └─── stackerPath
               └── DEsolver ───────────┬─── euler
                                       ├─── midpoint
                                       └─── rungeKutta4
```

Figure A.3: `evComponent` class hierarchy

`evComponents` encapsulate simulation capabilities so that they may be used as needed for a task-specific simulation. Objects of each of these types can be created and initialized through text files, and objects derived from the same type can be used interchangeably with other `evComponents`.

- `readParams` - initializes variables from a parsed text file
- `setVariables` - sets the value of one or more variables from task parameters included in a configuration
- `evInit` - one-time initialization function
- `init` - initialization function called every time a new configuration is evaluated
- `cleanup` - deallocates any data structures allocated by `init`
- `update` - called every simulation time step

`evComponents` can thus be created and initialized through text files, and provide a regular interface for including variables such as controller gains or trajectory via point locations as task parameters to be optimized.

The `evaluator` class provides high-level, task-specific simulation control. Derived `evaluators` typically do not contain much, if any, simulation details; these are left to the `evComponents`. The primary methods are:

- `evaluateConfiguration` - the main simulation loop
- `readParams` - parses initialization file, sets variables, and passes parsed information to `evComponents`
- `init` - initializes a configuration before simulation, and calls each `evComponent`'s `init` method
- `cleanup` - deletes configuration after simulation, and calls each `evComponent's` `cleanup` method
- `setVariables` - sets the value of one or more variables from a configuration's task parameters, and passes the remaining task parameters to the appropriate `evComponents`

The `evaluator` class handles parsing of initialization files and passes the parsed infor-

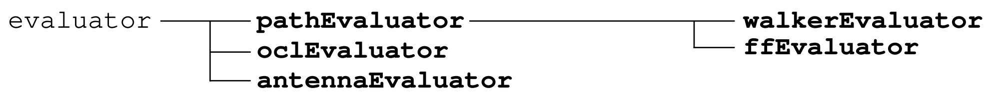


Figure A.4: `evaluator` class hierarchy
The pathEvaluator is a general-purpose evaluator; the others are application-specific. All except for the `oclEvaluator` were used in the experiments.

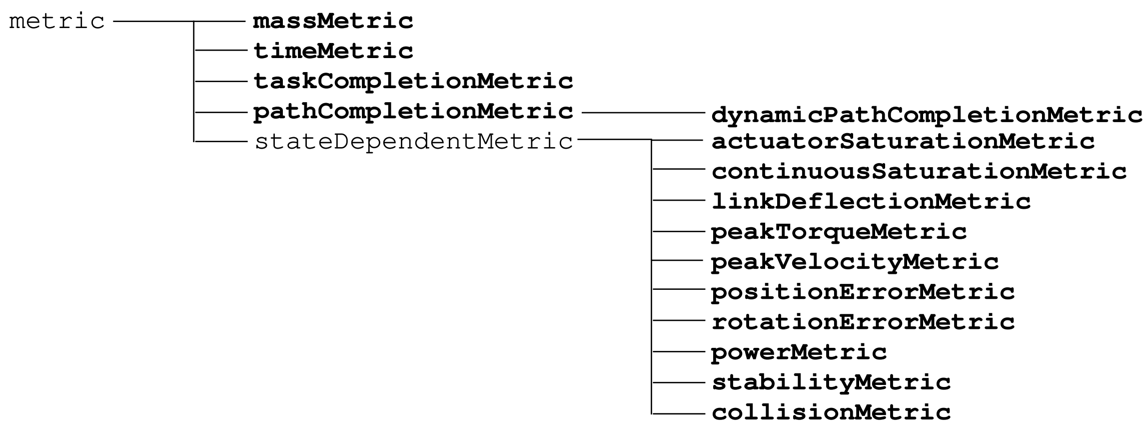


Figure A.5: `metric` class hierarchy
By default, `metrics` are state-independent--they measure a property of the robot or aspect of performance at the end of simulation. The `stateDependentMetric` and derived classes are state-dependent, and measure the robot's performance at each simulation time step.

---

mation to the appropriate `evComponents`. Figure A.4 shows the class hierarchy for evaluators.

The `metric` class measures the performance of configurations in simulation. Figure A.5 shows the class hierarchy; descriptions of each metric are given in Section 4.8. The most important methods for metrics are:

- `evaluate` - records the fitness of a configuration at the current simulation time step
- `biggerIsBetter` - returns 1 if larger values of the metric indicate better performance, or 0 otherwise
- `getVal` - returns the raw fitness measured for the configuration
- `getStdVal` - returns the standardized fitness computed from the raw fitness
- `setToWorst` - sets the raw fitness to the worst possible value; useful when a configuration fails catastrophically (e.g. tipover for a mobile robot)
- `isStateDependent` - returns 1 if the metric is state-dependent (i.e. should be measured at every time step.)

```
cfgFilter ──┬── dofFilter
            ├── endPointFilter
            └── moduleRedundancyFilter
```

Figure A.6: `cfgFilter` class hierarchy
The configuration filters are used by the synthesizer to cull configurations that are known to be unfavorable before they are evaluated.

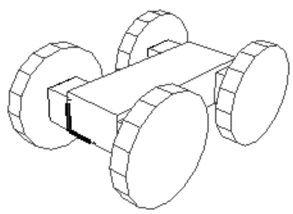
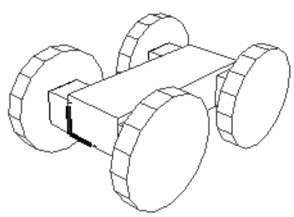
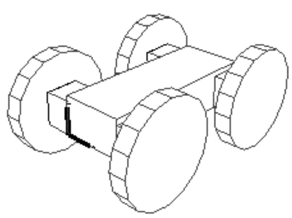
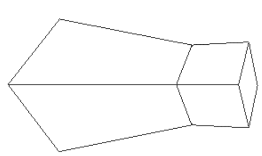
Each metric class also contains variables for scale and range. In addition, the `stateDependentMetric` class contains the `computeStats` method, which computes the minimum, maximum, average, integral, and root-mean-square values from a time series of recorded performance data.

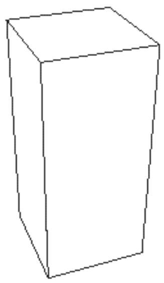
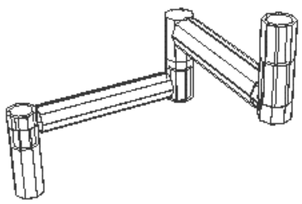
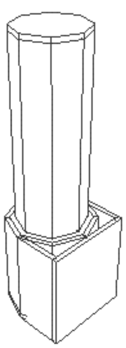
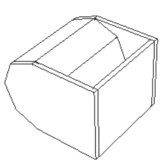
The remaining class that is likely to be useful to the designer is the `cfgFilter`, shown in Figure A.6 and described in Section 3.4.1. This class filters configurations that are determined to be unfavorable before they are evaluated. The cfgFilter has only two methods:

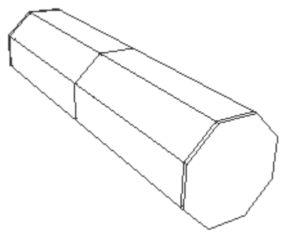
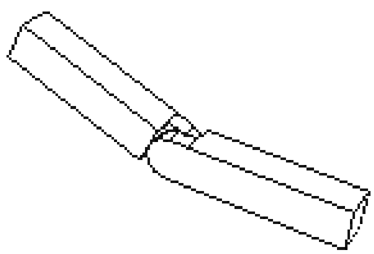
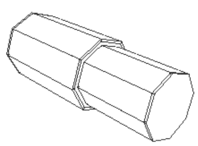
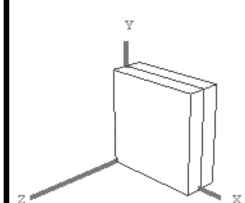
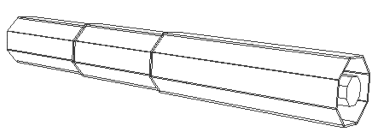
- `acceptable` - returns 1 if the configuration is acceptable according to the filter, or 0 if not
- `readParams` - reads variables from initialization file

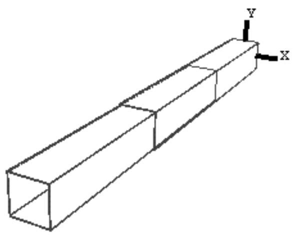
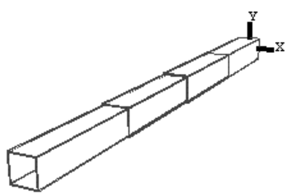
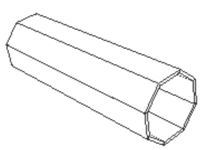
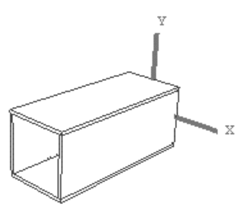
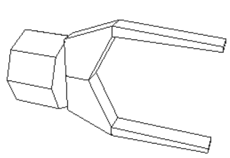
Appendix B: Module descriptions

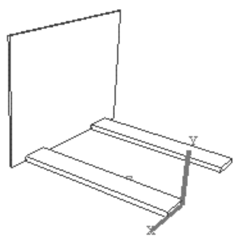
This section lists the modules used in the experiments and gives a brief description and list of parameters for each.

Class	Description	Parameters
`offroadChassis` 	Ackerman-steer mobile base	length width height connector position wheel diameter
`offroadChassis4WS` 	Four-wheel steer mobile base	same as offroadChassis
`mecanumBase` 	Mecanum base	same as offroadChassis
`ffBase` 	Base used for the free-flyer task	Front-to-back location of the base's origin
`virtualBase`	No geometric representation; allows root link of mechanism to be part of a module other than the base (see Section 5.6).	index of the serial chain whose distal link should be designated the root link

Class	Description	Parameters
`fixedBase` 	Fixed base for manipulators	x location y location z location x rotation (not used) y rotation (not used) z rotation (not used) x size y size z size
`scaraElbow` 	A 3-DOF module for planar positioning and orientation, with structural and actuator properties.	motor selection (joint 0) gearbox selection (joint 0) motor selection (joint 1) gearbox selection (joint 1) motor selection (joint 2) gearbox selection (joint 2) material selection distance between joint axes link 0 outer diameter link 0 wall thickness link 1 outer diameter link 1 wall thickness final joint orientation (can be up or down)
`rightAngleJoint` 	A joint module with a 90^{o} angle between connectors, with actuator and structural properties.	motor selection gearbox selection material selection outer diameter wall thickness overall length
`offsetElbow` 	An elbow joint with actuator, but not structural, models. The actuator housing is sized to contain the motor and gearhead. More realistic geometry than the `elbowJoint2` below.	motor selection gearbox selection material selection initial joint angle wall thickness clearance

Class	Description	Parameters
`elbowJoint2` 	An elbow joint with actuator, but not structural, models. The actuator housing is sized to contain the motor and gearhead.	motor selection gearbox selection material selection initial joint angle
`elbowJoint` 	An elbow joint without structural or actuator properties.	cross-section width cross-section height length
`inlineRevolute2` 	An inline joint with actuator and structural models. The inner diameter of the tube section containing the motor and gearhead is automatically enlarged if too small to house the actuator.	motor selection gearbox selection material selection outer diameter wall thickness overall length
`inlineRevolute` 	An inline joint module without structural or actuator properties.	cross-section width cross-section height length
`prismaticTube` 	A three-segment prismatic (telescoping) tube with actuator and structural properties. Actuated by a motor, gearhead, and lead screws.	motor selection gearbox selection lead screw selection material selection outer diameter wall thickness segment length

Class	Description	Parameters
`prismaticBeam2` 	A two-segment prismatic beam with square cross-section. No actuator or structural properties	length cross-section width/height
`prismaticBeam3` 	A three-segment prismatic beam with square cross-section. No actuator or structural properties	length cross-section width/height
`hollowTube` 	A hollow link with circular cross section. Includes structural properties.	material selection length outer diameter wall thickness
`squarePrism` 	A link module with no structural description; suitable only for purely kinematic synthesis.	cross-section width/height length
`virtualLink`	No geometric representation. Effectively allows multiple const attachments to a subgraph that can vary arbitrarily (see Section 5.6).	none
`walkerGripper` 	Gripper for the truss-walker experiment	1 parameter, not currently used

Class	Description	Parameters
`MEETool` 	Represents Ranger's Microconical End Effector (MEE); used in the Free-Flyer experiment.	none
`palletFork` 	Pallet forks for the material handler experiment.	none

Appendix C: Detailed robot descriptions

C.1 Free-flyer

This section presents all of the information generated by the synthesizer for the free-flyer with lowest mass from the experiment in Section 5.1. Figure C.1 shows hidden-line and shaded renderings of the robot, while the text description is given in Figure C.2. The values for module and task parameters are given in Tables C.1 and C.2, respectively. Finally, this configuration's performance is summarized in Table C.3

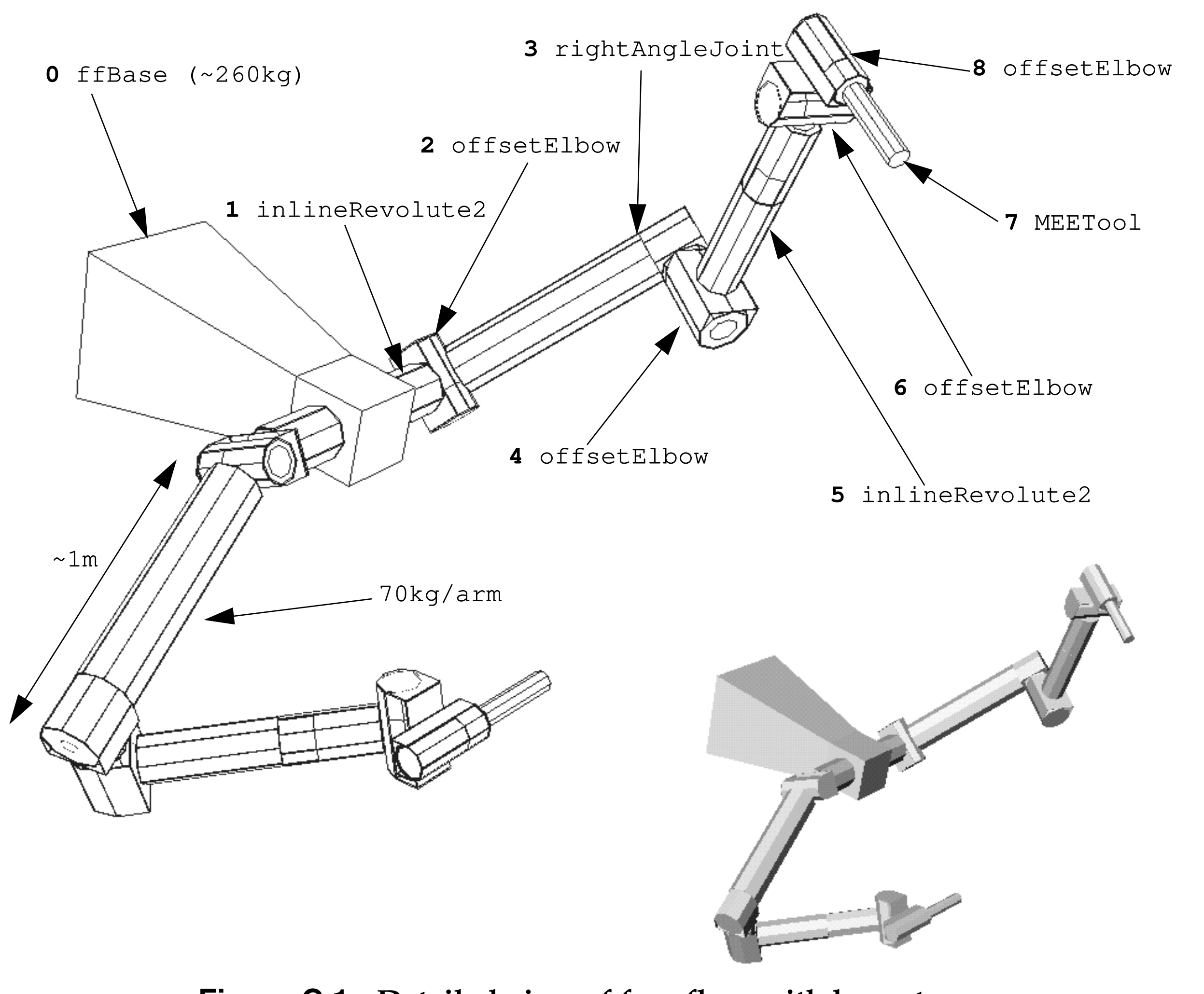


Figure C.1: Detailed view of free-flyer with lowest mass
Shown here is the free-flyer with lightest mass from the experiment in Section 5.1. The hidden-line rendering is annotated with the index and type of each module. The text description for the robot is given in Figure C.2

```
((ffBase ((var -0.5 0.5 4 6))
         ((const 0 (1 0 right (const 0 270 2 2)))
          (const 1 (1 0 left (const 0 270 2 3)))))
 (inlineRevolute2 revoluteJoint ((var 0 1 3 5)
                                 (var 0 1 5 9)
                                 (const 0 1 2 0)
                                 (var 0.1 0.2 3 7)
                                 (const 0.003 0.01 3 3)
                                 (var 0.05 0.8 4 0))
                                ((var 1 (2 1 inherit (var 0 270 2 1)))))
 (offsetElbow revoluteJoint ((var 0 1 3 6)
                             (var 0 1 5 21)
                             (const 0 1 2 0)
                             (var -90 90 4 10)
                             (const 0.003 0.01 3 3)
                             (var 0.005 0.03 2 2))
                            ((var 0 (3 1 inherit (var 0 270 2 1)))))
 (rightAngleJoint revoluteJoint ((var 0 1 3 2)
                                 (var 0 1 5 11)
                                 (const 0 1 2 0)
                                 (var 0.1 0.2 3 7)
                                 (var 0.003 0.01 3 0)
                                 (var 0.05 1 4 14))
                                ((var 0 (4 1 inherit (var 0 270 2 1)))))
 (rightAngleJoint revoluteJoint ((var 0 1 3 7)
                                 (var 0 1 5 18)
                                 (const 0 1 2 0)
                                 (var 0.1 0.2 3 7)
                                 (var 0.003 0.01 3 0)
                                 (var 0.05 1 4 2))
                                ((var 0 (5 1 inherit (var 0 270 2 2)))))
(inlineRevolute2 revoluteJoint ((var 0 1 3 7)
                                 (var 0 1 5 21)
                                 (const 0 1 2 0)
                                 (var 0.1 0.2 3 4)
                                 (var 0.003 0.01 3 0)
                                 (var 0.05 1 4 11))
                                ((var 0 (6 0 inherit (var 0 270 2 3)))))
(offsetElbow revoluteJoint ((var 0 1 3 6)
                             (var 0 1 5 18)
                             (const 0 1 2 0)
                             (var -90 90 4 7)
                             (const 0.003 0.01 3 3)
                             (var 0.005 0.03 2 3))
                            ((var 1 (7 0 inherit (var 0 270 2 3)))))
 (rightAngleJoint revoluteJoint ((var 0 1 3 5)
                                 (var 0 1 5 13)
                                 (const 0 1 2 0)
                                 (var 0.1 0.2 3 0)
                                 (var 0.003 0.01 3 1)
                                 (var 0.05 1 4 0))
                                ((var 1 (8 0 inherit (var 0 270 2 1)))))
 (MEETool () ()))
```

Figure C.2: Free-flyer text description

ID	Module/parameter/ attachments	value	ID	Module/parameter/ attachments	value
0	`ffBase` origin location	 -0.1m along Z	5	`inlineRevolute2` motor gearbox material (`const`) tube diameter tube wall thickness tube length	 Maxon RE75.118825 HDS CSF-45-160 aluminum 15 cm 3 mm 75 cm
	`0 -> 1,0 right` `1 -> 1,0 left`	180^{o} 270^{o}		`0 -> 6,0 inherit`	270^{o}
1	`inlineRevolute2` motor gearbox material (`const`) tube diameter tube wall thickness tube length	 Maxon RE75.118825 HDS CSF-25-80 aluminum 20 cm 6mm 5 cm	6	offsetElbow motor gearbox material (`const`) initial angle (unused) wall thickness clearance	 Maxon RE75.118825 HDS CSF-40-120 aluminum -6 6 mm 3 cm
	`1 -> 2,1 inherit`	90^{o}		`1 -> 7,0 inherit`	270^{o}
2	`offsetElbow` motor gearbox material (`const`) initial angle (unused) wall thickness clearance	 MaxonRE75.118825 HDS CSF-45-160 aluminum 30^{o} 6 mm 2.2 cm	7	`rightAngleJoint` motor gearbox material (`const`) tube diameter tube wall thickness tube length	 Maxon RE75.118825 HDS CSF-32-80 aluminum 10 cm 4 mm 5 cm
	`0 -> 3,1 inherit`	90^{o}		`1 -> 8,0 inherit`	90^{o}
3	`rightAngleJoint` motor gearbox material (`const`) tube diameter tube wall thickness tube length	 Maxon2260.889 HDS CSF-25-160 aluminum 20 cm 3 mm 94 cm	8	`MEETool`	
	`0 -> 4,1 inherit`	90^{o}			
4	`rightAngleJoint` motor gearbox material (`const`) tube diameter tube wall thickness tube length	 MaxonRE75.118825 HDS CSF-40-120 aluminum 20 cm 3 mm 18 cm			
	`0 -> 5,1 inherit`	180^{o}			

Table C.1: Parameter values for free-flyer modules

All motors are from Maxon; gearboxes are actually harmonic drives from HD Systems. Connections are listed as (connector -> child ID, child connector, handedness, twist).

Object	Variable	Value
ffEvaluator	originPosY	2.4 m
	originPosZ	1.0 m
basePath1	vel	0.64 m/s
	maxAcc	1.23 m/s^2
basePath2	vel	0.16 m/s
	maxAcc	1.18 m/s^2

Table C.2: Task parameters for lightest free-flyer

Priority	Metric	Value
0	pathCompletionMetric	1.0
	collisionMetric	0.0
	positionErrorMetric	0.8 mm
1	dynamicPathCompletionMetric	1.0
	linkDeflectionMetric	0.5 mm
	continuousSaturationMetric	62 %
2	massMetric	401.7 kg
	powerMetric	7.1 kJ
	timeMetric	45.9 s

Table C.3: Performance measurements for lightest free-flyer

C.2 Space-Shuttle Waterproofing Manipulator

This section gives details on the properties of the lightest manipulator from the High-Level experiments in Section 5.2. Figure C.3 shows two renderings of the manipulator with labels indicating modules. (This manipulator is also shown in the upper right corner of Figure 5.13.) Figure C.4 gives the text description for the manipulator, while Table C.4 lists its module properties and connections in a more easily readable format. Table C.5 lists the task parameters for this robot, and Table C.6 summarizes the robot's performance.

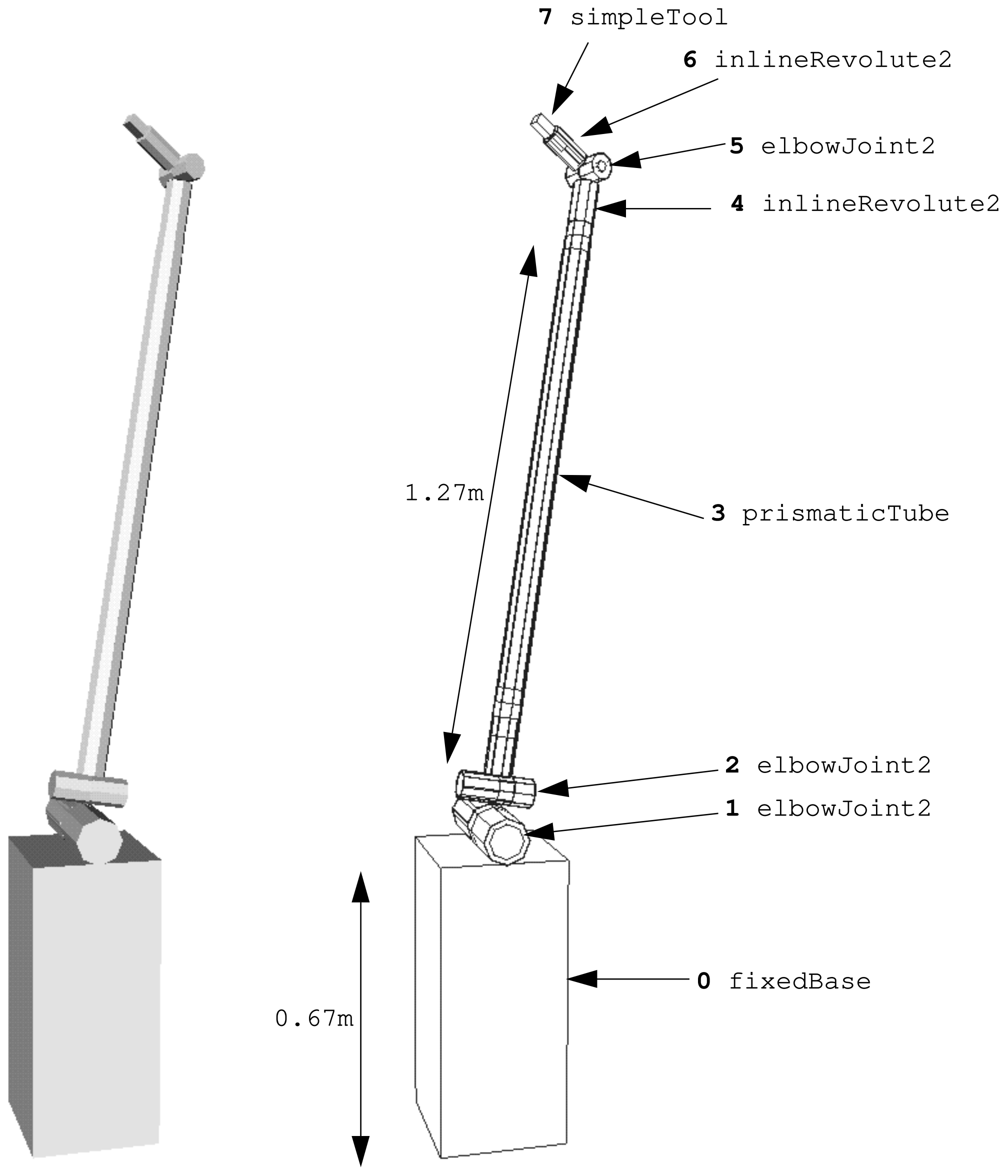


Figure C.3: Space shuttle servicing manipulator

This figure shows the lightest manipulator synthesized for the High-Level experiments in Section 5.2. Note that the prismatic joint is almost fully contracted and the length shown next to the `prismaticTube` is for the length of a single segment.

```
((fixedBase ((var -1.5 1.5 5 12)
(var -1.5 1.5 5 16)
(const 0 1 1 0)
(const 0 1 1 0)
(const 0 1 1 0)
(const 0 1 1 0)
(const 0.1 0.8 3 2)
(const 0.1 0.8 3 2)
(var 0.01 2.5 4 4))
 ((const 0 (1 1 left (subp 0 270 2 3)))))
(elbowJoint2 revoluteJoint ((var 0 1 3 7)
(var 0 1 5 29)
(const 0 1 2 0)
(var -180 135 3 2))
 ((const 0 (2 0 left (const 0 270 2 1)))))
(elbowJoint2 revoluteJoint ((var 0 1 3 3)
(var 0 1 5 13)
(const 0 1 2 0)
(var -180 135 3 5))
 ((subp 1 (3 0 left (subp 0 270 2 1)))))
(prismaticTube prismaticTube ((var 0 1 3 1)
(var 0 1 5 4)
(var 0 1 1 1)
(const 0 1 2 0)
(var 0.05 0.15 3 0)
(var 0.0015 0.005 3 0)
(var 0.3 2 3 4))
 ((subp 1 (4 0 left (subp 0 270 2 3)))))
(inlineRevolute2 smallRevoluteJoint ((var 0 1 2 1)
(var 0 1 2 2)
(const 0 1 2 0)
(var 0.05 0.15 3 1)
(var 0.0015 0.005 3 0)
(var 0.05 0.3 3 1))
 ((const 1 (5 0 left (var 0 270 2 2)))))
(elbowJoint2 smallRevoluteJoint ((var 0 1 3 2)
(var 0 1 5 8)
(const 0 1 2 0)
(var -180 135 3 6))
 ((const 1 (6 0 left (subp 0 270 2 2)))))
(inlineRevolute2 smallRevoluteJoint ((var 0 1 2 1)
(var 0 1 2 0)
(const 0 1 2 0)
(var 0.05 0.15 3 0)
(var 0.0015 0.005 3 0)
(var 0.05 0.3 3 0))
 ((const 1 (7 0 left (subp 0 270 2 0)))))
(simpleTool ((const 0.05 0.1 3 0)) ()))
```

Figure C.4: Text description for Space Shuttle servicing manipulator

ID	Module/parameter/ attachments	value	ID	Module/parameter/ attachments	value
0	fixedBase x origin y origin height	 -0.33871 0.0483871 0.674	4	inlineRevolute2 motor gearbox material (const) tube diameter tube wall thickness tube length	 maxonRE25.118748 maxon26.110398 aluminum 0.0642857 0.0015 0.0857143
	0 -> 1,1 left	270°		**1 -> 5,0 left**	180°
1	elbowJoint2 motor gearbox material (const) initial angle (unused)	 maxonRE75.118825 maxon81.110413 aluminum -90	5	elbowJoint2 motor gearbox material (const) initial angle (unused)	 maxonRE25.118755 hds.csf-14-50 aluminum 90
	0 -> 2,0 left	90°		**1 -> 6,0 left**	180°
2	elbowJoint2 motor gearbox material (const) initial angle (unused)	 maxon2260.889 hds.csf-20-160 aluminum 45	6	inlineRevolute2 motor gearbox material (const) tube diameter tube wall thickness tube length	 maxonRE25.118748 maxon26.110396 aluminum 0.05 0.0015 0.05
	1 -> 3,0 left	90°		**1 -> 7,0 left**	0°
3	prismaticTube motor gearbox lead screw material (const) diameter wall thickness segment length	 maxon2260.815 hds.csf-14-50 screw2 aluminum 5 cm 1.5mm 1.27m	7	simpleTool size (const)	 5cm
	1 -> 4,0 left	270°			

Table C.4: Manipulator properties

This table lists the parameter values for the manipulator shown in Figure C.3. Connections are described as (parent connector ID -> child module ID, child connector ID, handedness, twist). Bold connections are const connections that were specified in a kernel configuration. In each of the High-Level manipulator trials, the best configurations contained both of the const subgraphs shown above, although they were optional since some kernels did not contain the subgraphs. Only the variable parameters of the fixedBase are shown. The lead screw used in the prismaticTube (screw2) has a 1cm diameter, a pitch of 45 degrees, and mass of 600g per meter length.

Object	Variable	Value
`path`	`vel`	1.03 m/s
	`maxAcc`	4.57 m/s^2
	`omega`	2.24 rad/s
	`maxOmegaDot`	6.47 rad/s^2

Table C.5: Manipulator task parameter values

Listed here are the values of the task parameters for the manipulator shown in Figure C.3. The parameters describe the linear and angular velocity and acceleration to be used to when following the task's trajectory.

Priority	Metric name	value
0	`pathCompletionMetric`	1.0
	`collisionMetric`	0.0
	`positionErrorMetric`	1.2 cm
1	`linkDeflectionMetric`	1.7 mm
	`actuatorSaturationMetric`	96 %
2	`massMetric`	12.5 kg
	`timeMetric`	26.4 s

Table C.6: Manipulator performance summary

Glossary

actuator saturation - the ratio of applied torque (or force) to an actuator's maximum available torque (or force). An actuator saturation greater than one indicates that the controller commanded a torque beyond the actuator's capability.

class - in object-oriented programming, a class is a description of a data type and a set of related functions.

component context - a set of component lists, with one list for each of a module's component-selection parameters. Each component list contains one or more components of similar type, e.g. motors or actuators.

configuration - the general form of the robot, including kinematics and other geometry at the bare minimum and usually including descriptions of inertial properties, actuator and material selection, and structural geometry.

configuration graph - see Parameterized Module Configuration Graph (PMCG)

configuration optimization - the process of improving the performance of a robot configuration (or small number of configurations) through parametric variation

configuration synthesis - the process of generating a high-level description of a robot and improving its performance through parametric and topological variation.

`const` flag - a flag associated with parameters and attachments in the Parameterized Module Configuration Graph that can be set by the designer to indicate that the parameter or attachment should not be changed by the synthesizer.

degree of freedom (DOF) - an independent variable describing part of the state of a robot. The state of an n-DOF robot can be uniquely and completely described by n variables.

Denavit-Hartenburg (D-H) parameters - a commonly-used set of parameters describing the kinematics of a robot, in which each pair of successive joints is characterized by a distance between joint axes a, a twist between joint axes α, an offset d, and a joint angle θ. See e.g. [Denavit55] for details.

directed acyclic graph (DAG) - a graph in which each edge has a direction and in which no path along the edges passes through a node more than once.

dynamic simulation - computing the motion of a mechanical system (e.g. a robot) based on the forces and torques applied to the system. (Compare to kinematic simulation.)

elitism - in an evolutionary algorithm, elitism refers to explicitly preserving or reproducing a subset of the population that are considered to be 'best'.

elite set - in an evolutionary algorithm, the subset of the population that is considered to be 'best'. In a single-objective EA, the elite set might contain the *n*-best solutions; in a multi-objective EA, it might contain the Pareto-optimal set. In Darwin2K, it is defined to be the feasibly-optimal set.

evolutionary algorithm (EA) - an optimization algorithm (often probabilistic in nature) based on biological theories of evolution. Typical features include the use of a population of solutions, selection of solutions based on fitness (analogous to "survival of the fittest"), and creating new solutions by combining or modifying existing solutions.

evolutionary synthesis engine (ESE) - in Darwin2K, the program that maintains a population of configurations, creates new configurations, and sends them to one or more evaluation processes which measure their performance.

feasibly-optimal set - in Darwin2K, the subset of the population that is considered 'best'. If any configurations are feasible, then the feasibly-optimal set is the Pareto-optimal set taken over all feasible configurations in the population. If no configurations are feasible, then the feasibly-optimal set is the Pareto-optimal set taken over the entire population.

fitness - a figure of merit reflecting the performance of a solution in an evolutionary algorithm

fitness-proportionate selection - in an evolutionary algorithm, a method of selecting solutions for reproduction in which a solution is selected with probability directly proportional to its fitness

generational genetic algorithm - a genetic algorithm that creates an entire new population of solutions at once and then evaluates them all. This is the standard method for creating new solutions in a GA.

genetic algorithm (GA) - an evolutionary algorithm that represents solutions as a string of symbols (usually a fixed-length string of bits), uses crossover, duplication, and mutation to create new solutions, and selects solutions for reproduction on the basis of their performance.

genetic programming (GP) - an evolutionary algorithm that represents solutions as programs. The fitness of a program is determined by executing it, rather than by measuring some property of the program.

globally optimal planner - a planner that is guaranteed to find the globally-optimal motion with respect to a cost metric (usually either time or distance) down to the level of discretization used.

graph - a data structure or mathematical object consisting of nodes connected by edges. Each edge has two endpoints and may have a direction associated with it.

kernel - an initial configuration specified by the designer from which all other configurations are generated through the application of genetic operators.

kinematic simulation - computing the motion of a mechanical system (e.g. a robot) by considering only position variables and their derivatives, rather than forces and torques.

kinematics - the study of a mechanism's motions without regard to forces, torques, or inertia.

locally optimal planner - a planner that is not guaranteed to find the optimal motion. Typically much less computationally expensive than globally-optimal planners.

member - in OOP, part of the description of a class. A data member describes a property of objects that belong to the class, while a member function (or method) describes the behavior of objects belonging to the class.

method - see member

module - a self-contained software object representing part of a robot. In Darwin2K, a module contains data describing the its properties, and functions describing its behavior.

object - in Object-Oriented Programming, a self-contained set of properties and associated procedures.

parameter - in Darwin2K, a value that may be varied by the synthesis algorithm. Parameters are described by several features: the minimum and maximum values of the parameter, the resolution, an integer value, a real value, and a `const`-flag. Modules can have parameters, and configurations can have parameters associated with them that describe aspects of the task.

parameterized module - an object representing part of a robot, including both data and function members. Parameterized modules may have zero or more parameters describing arbitrary properties, and can specify connectors that indicate how the module can be connected to other modules.

parameterized module configuration graph (PMCG) - the representation for robot configurations used in Darwin2K. The PMCG consists of a list of modules and connections between them and allows both parametric and topological variation of robot properties.

Pareto-optimal set - given a set of multi-dimensional measurements (e.g. robot configurations with performance measurements), the Pareto-optimal set is those measurements that are better than or equivalent to every other measurement along at least one dimension. Also called the non-inferior set, as every member of the set is not inferior (i.e. worse than another element in all dimensions) to any other element of the set.

simple genetic algorithm (SGA) - the basic genetic algorithm, using a single population and representing solutions as fixed-length bit strings.

steady-state genetic algorithm (SSGA) - a genetic algorithm that continuously adds solutions to, and removes solutions from, a population. Contrast to a generational genetic algorithm.

task parameter - a variable or property that does not belong to a configuration but which can be optimized by the synthesizer. Examples include via point location and controller gains.

tool control point (TCP) - a position and orientation defined relative to an end effector (tool) that is used to specify the motion of the effector. For example, the TCP for a gripper might be the point midway between the gripper's fingers.

References

[Ambrose94] R. Ambrose and D. Tesar. The optimal selection of robot modules for space manipulators. In *Proceedings of the ASCE Space 94 Conference*, February 1994.

[Apostolopoulos96] D. Apostolopoulos. Systematic configuration of robotic locomotion. Technical Report CMU-RI-TR-96-30, The Robotics Institute, Carnegie Mellon University, 1996.

[Arocena98] J. Arocena and R. Daniel. Design and control of a novel 3-dof flexible robot, part 2: Modeling and control. *International Journal of Robotics Research*, 17(11):1182–1201, Nov 1998.

[Baraff96] D. Baraff. *Coriolis Documentation*, 1996.

[Bares93] J. Bares and W. Whittaker. Configuration of autonomous walkers for extreme terrain. *International Journal of Robotics Research*, 12(6), 1993.

[Bares99] J. Bares and D. Wettergreen. Dante II: Technical description, results and lessons learned. *International Journal of Robotics Research*, 18(7):621–649, July 1999.

[Barraquand92] J. Barraquand, B. Langlois, and J.-C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-22(2):224–241, March/April 1992.

[Barraquand89] J. Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. Technical report, Computer Science Department, Stanford University, 1989.

[Bentley99a] P. Bentley. *Evolutionary Design by Computers*. Morgan Kaufmann, 1999.

[Bentley99b] P. Bentley. From Coffee Tables to Hospitals: Generic Evolutionary Design. In P. Bentley (ed.) *Evolutionary Design by Computers*, pages 405–423. Morgan Kaufmann, 1999.

[Brayton90] R. Brayton, G. Hachtel and A. Sangiovanni-Vincentelli, *Multilevel Logic Synthesis*, Proceedings of the IEEE, Vol. 78, No. 2, pp. 264-300, February 1990.

[Cadence94] Cadence Design Systems, Inc. *Verilog static analysis and dynamic simulators*, San Jose, 1994.

[Chedmail96] 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.

[Chen95] 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.

[Chen96] I. Chen. On optimal configuration of modular reconfigurable robots. In *Proceedings of the 4th International Conference on Control, Automation, Robotics, and Vision*, 1996.

[Chen99] S. Chen, C. Guerra-Salcedo, and S. Smith. Non-standard crossover for a standard representation – commonality-based feature subset selection. In *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 1999.

[Chocron97] O. Chocron and P. Bidaud. Genetic design of 3d modular manipulators. In *Proceedings of the 1997 IEEE International Conference on Robotics and Automation*, pages 223–228, April 1997.

[Cole93] J. Cole. Rapid generation of motion plans for modular robotics systems. Master's thesis, Massachusetts Institute of Technology, 1993.

[Craig89] J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley, 2nd edition, 1989.

[Denavit55] J. Denavit and R. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *ASME Journal of Applied Mechanisms*, pages 215–221, 1955.

[Dowling92] K. Dowling, R. Bennett, M. Blackwell, T. Graham, S. Gatrall, R. O'Toole, and H. Schempf. A mobile robot for ground servicing operations on the space shuttle. In *OE / Technology 1992 Cooperative Intelligent Robots in Space*. SPIE, November 1992.

[Eby99] P. Eby, R. Averill, W. Punch III, and E. Goodman. *Evolutionary Design by Computers*, chapter The Optimisation of Flywheels using an Injection Island Genetic Algorithm, pages 167–190. Morgan Kaufmann, 1999.

[Farritor96a] 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.

[Farritor96b] S. Farritor, S. Dubowsky, N. Rutman, and J. Cole. A systems-level modular design approach to field robotics. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, pages 2980–2895, 1996.

[Farritor98] S. Farritor. *On Modular Design and Planning for Field Robotic Systems*. PhD thesis, Massachusetts Institute of Technology, May 1998.

[Funes99] P. Funes and J. Pollack. , chapter Computer Evolution of Buildable Objects. In P. Bentley (ed.) *Evolutionary Design by Computers*, pages 387–404. Morgan Kaufmann, 1999.

[Goldberg91] D. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, . Morgan Kaufmann, 1991.

[Holland75] J. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[Kelmar90] 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.

[Khatib86] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1):90–98, Spring 1986.

[Kim92] J.-O. Kim. *Task Based Kinematic Design of Robot Manipulators*. PhD thesis, Carnegie Mellon University, 1992.

[Kim93] 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.

[Koza92] J. Koza. *Genetic programming : on the programming of computers by means of natural selection*. MIT Press, 1992.

[Koza94] J. Koza. *Genetic programming II : automatic discovery of reusable programs*. MIT Press, 1994.

[Koza96] 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.

[Krasnicki99] M. Krasnicki, R. Phelps, R. Rutenbar, and R. Carley. Maelstrom: Efficient simulation-based synthesis for custom analog cells. In *Proceedings of the 1999 ACM/IEEE Design Automation Conference*, 1999.

[Krishnan99] A. Krishnan and P. Khosla. A methodology for determining the dynamic configuration of a reconfigurable manipulator system. In *Proceedings of the 5th Annual Aerospace Applications of AI Conference*, 1989.

[Kuh90] Kuh, E.S., and T. Ohtsuki, *Recent Advances in VLSI Layout*, IEEE Proceedings Special Issue on Computer-Aided Design, vol. 78, no. 2, pp. 237-263, February 1990.

[Latombe91] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Leger98] C. Leger and J. Bares. Automated synthesis and optimization of robot configurations. In *Proceedings of the 1998 ASME Design Engineering Technical Conferences*, 1998.

[Leger99] C. Leger and J. Bares. Automated task-based synthesis and optimization of field robots. In *Proceedings of the 1999 International Conference on Field and Service Robotics* (FSR99), 1999.

[Levine92] J. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, 1992.

[Liegeois77] A. Liegeois. Automatic supervisory control of the configuration and behavior of multibody mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-7(12):868–871, December 1997.

[McCrea97] 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.

[McGeer90] T. McGeer. Passive dynamic walking. *International Journal of Robotics Research*, 9(2):62–82, April 1990.

[Messuri85] 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.

[Muir88] P. Muir. *Modelling and Control of Wheeled Mobile Robots*. PhD thesis, Carnegie Mellon University, Dept. of Electrical and Computer Engineering, 1988.

[Nakamura86] Y. Nakamura. Inverse kinematic solutions with singularity robustness for robot manipulator control. *Journal of Dynamic Systems, Measurement, and Control*, 108:163–171, September 1986.

[Nakamura91] Y. Nakamura. *Advanced Robotics: Redundancy and Optimization*. Addison-Wesley, 1991.

[Ochotta96] E. Ochotta, R. Rutenbar, and R. Carley. Synthesis of high-performance analog circuits in astrx/oblx. Technical Report CMUCAD-96-55, Center for Electronic Design Automation, Carnegie Mellon University, 1996.

[Pads99] *PowerPCB - PCB Layout and autorouting software*, http://www.pads.com, 1997

[Paredis93] C. Paredis. *An Agent-Based Approach to the Design of Rapidly Deployable Fault Tolerant Manipulators*. PhD thesis, The Robotics Institute, Carnegie Mellon University, 1996.

[Paredis96] C. Paredis and P. Khosla. Kinematic design of serial link manipulators from task specifications. *The International Journal of Robotics Research*, 12(3):274–287, June 1993.

[Pedersen98] J. Pedersen. Robust communications for high bandwidth real-time systems. Master's thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 1998.

[Preparata88] F. Preparata and M. Shamos. *Computational geometry : an introduction*. Springer-Verlag, 2nd edition, 1988.

[Press92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes In C*. Cambridge University Press, 2nd edition, 1992.

[Rosenman97] M. Rosenman. The Generation of Form Using an Evolutionary Approach. In D. Dasgupta and Z. Michalewicz (eds.) *Evolutionary Algorithms in Engineering Applications*, pages 135–154. Springer-Verlag, 1997.

[Roston94] G. Roston. *Genetic Methodology for Configuration Design*. PhD thesis, The Robotics Institute, Carnegie Mellon University, 1994.

[Rutman95] N. Rutman. Automated design of modular field robots. Master's thesis, Massachusetts Institute of Technology, 1995.

[SSL99] University of maryland space science laboratory's ranger project homepage: http://www.ssl.umd.edu/homepage/projects/ranger.html. Online documentation, November 1999.

[Shigley89] J. Shigley and C. Mischke. *Mechanical engineering design*. McGraw-Hill, New York, fifth edition, 1989.

[Sims94] K. Sims. Evolving virtual creatures. In *1994 Computer Graphics Proceedings*, pages 15–22, 1994.

[Sunrise94] Sunrise, Inc., *ATPG (Automatic Test Pattern Generator)*, San Jose, 1994.

[Synopsis94] Synopsis Inc., *Synthesizer for VLSI net-list generation*, San Jose, 1994.

[Syswerda89] G. Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[Wettergreen99] D. Wettergreen, D. Bapna, M. Maimone, and H. Thomas. Developing nomad for robotic exploration of the atacama desert. *Robotics and Autonomous Systems*, 26(2-3):127–148, February 1999.

[Whitley90] L. Whitley and T. Starkweather. Genitor II: A distributed genetic algorithm. *Journal of Experimental and Theoretical Artificial Intelligence*, (2):189–214, 1990.

[Zeglin99] G. Zeglin. *The Bow-Leg Hopping Robot*. PhD thesis, The Robotics Institute, Carnegie Mellon University, 1999.

Index

T

V

W