Sensorimotor Primitives for Programming Robotic Assembly Skills

James Daniel Morrow

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

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

May 1997

This Research was supported in part by the Department of Energy Computational Science Fellowship Program, by the Department of Energy Integrated Manufacturing Predoctoral Fellowship, by Sandia National Laboratories, and by The Robotics Institute at Carnegie Mellon University. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies.

Abstract

This thesis addresses the problem of sensor-based skill composition for robotic assembly tasks. Skills are robust, reactive strategies for executing recurring tasks in our domain. In everyday life, people rely extensively on skills such as walking, climbing stairs, and driving cars; proficiency in these skills enables people to develop and robustly execute high-level plans. Unlike people, robots are unskilled -- unable to perform any task without extensive and detailed instructions from a higher-level agent. Building sensor-based, reactive skills is an important step toward realizing robots as flexible, rapidly-deployable machines. Efficiently building skills requires simultaneously reducing robot programming complexity and increasing sensor integration, which are competing and contradictory goals. This thesis attacks the problem through development of sensorimotor primitives to generalize sensor integration, graphical programming environments to facilitate skill composition, and design integration for exploiting the primitive capabilities during task design. Force and vision based sensorimotor primitives are built to provide sensor integration for task domains, not task instances. A task-level graphical programming environment provides an intuitive interface for the designer/programmer to compose the skill through a simple form of virtual demonstration. Finally, design agent representations of the primitives inside the CAD environment interact with the task model and the programmer to assist in composing the skill program. Experimental results for six different assembly tasks demonstrate sensor-based skill execution and illustrate primitive reusability.

Acknowledgments

Thanks to my advisor, Pradeep Khosla, for assembling a first-rate research group and facility in the Advanced Mechatronics Laboratory, and for giving me the freedom and support to pursue my own research direction. Thank you also to my committee members Mike Erdmann, Lee Weiss, and Brenan McCarragher for their interest and insights. Thanks to the Department of Energy Computational Science and Integrated Manufacturing Fellowship programs for their financial support of this research.

Many people in our lab group helped me considerably. Brad Nelson encouraged me often early in my PhD and provided valuable help on visual servoing techniques. Brad was also instrumental in guiding my research direction toward task-based primitives. Dave Stewart built Chimera and a number of software tools which I benefited from greatly throughout this research. Richard Voyles and I collaborated on research ranging from shape-from-motion calibration to M&M sampling. Rich provided subliminal messages and lots of laughs in addition to his hardware and real-time system expertise -- it wouldn't have been the same without him. Chris Lee and I had many discussions about building an event-driven level for Chimera -- these conversations had a strong impact on the 4th level implementation that I developed. Chris Paredis helped me with Telegrip in addition to providing stiff racquetball competition.

Thanks to the members of the AML group for feedback during our AML meetings on my research and to Henry Schneiderman, Carol Hoover, Chris Paredis, Chris Lee, and Richard Voyles for particularly helpful comments on my defense presentation.

The Robotics Institute is a great place to do graduate work. I really appreciate the terrific environment, resources, and especially the people -- students, faculty, and staff -- who are approachable and engaged in interesting research. I don’t think you could find a more open, supportive, and stimulating environment.

Most of all, I want to thank my wife, Joan, who made all of this possible. Her unwavering love and support as 4 years stretched to 6 was instrumental in my completing the PhD. Joan gave me the most wonderful gift 22 months ago: Maria Catherine. Aside from being loads of fun, Maria is a constant reminder of how far we still have to go in robotics.

Chapter 1

Introduction

1.1 The Problem

Adaptable, intelligent robot programs require sensor integration. Lack of sensing leads to very brittle robot control programs which cannot adapt to uncertain knowledge about the task during programming or imperfect control during execution. But sensor integration increases programming difficulty leading to a trade-off between programming difficulty and program adaptability. This undermines robotics' 'niche' as flexible automation for medium-volume manufacturing between people and hard automation. To realize rapidly deployable systems requires simultaneously integrating sensors while reducing the programming burden. This thesis addresses the problem of efficient sensor-based skill programming for robotic assembly tasks.

The difficulty of robot programming has been known for some time. Automatic planners were developed so that tasks could be described at very high level and then a robot program automatically generated. The problem is that we ask too much of planners -- that they give us a plan in terms of robot motions which will execute in spite of world-modelling errors used at plan-time and the uncertainty inherent in the real-world. The sequence of steps can be recovered at plan time, but the details of robot motion often cannot for tasks involving uncertainty. These planners need more powerful, reactive commands -- *skills*.

A skill represents an innate capability to perform a specific (and usually recurring) task. Examples from everyday life include walking, opening doors, climbing stairs, and driving. These skills enable people to execute high-level plans and limit the necessary level of communication detail. A skill is specific to a task but possesses the ability to deal with uncertainty and variations in that task. For example, a door opening skill might effectively deal with different handles, sizes and weights of the door, and spring/damper mechanisms attached to the door. Embedded in these skills is appropriate sensor use to resolve limited task uncertainty.

In contrast to people, robots are unskilled -- unable to do any task without extensive and detailed directions from a higher-level system. Developing robot skills for recurring, complex tasks would significantly impact the robot programming problem. Integrating such skills with high-level plans is an instantiation of the general principle of combining deliberative planning and reaction which is widely accepted in robotics today [35]. In the robotics literature skills mostly refer to sensorimotor mappings which are often learned. In this thesis, a skill implements a particular algorithm (usually multi-step) to solve a specific manipulation task. There is a many-to-one mapping between skills and a task -- that is, many different strategies or algorithms can be employed to solve a particular manipulation task. Simon et al [61] investigated this question for snap-fit tasks with a single-step strategy. One of the key conclusions they drew was that the optimal strategy (the ‘best’ parameter set of the move command) was a function of not just the task, but of the performance index chosen. The focus of this research is not on building optimal skills for specific tasks, but on developing and implementing an approach to efficiently building *satisficing* skills for several different, yet related, tasks by transferring portions of a skill for one task to a skill for a similar task. Assembly tasks are targeted since contact tasks tend to be the most difficult robot tasks to program.

Fitts [63] identified three phases of human skill acquisition: cognitive, associative, and autonomous. During the cognitive phase the basic strategy is recovered. During the associative phase the skill performance is improved through practice. The skill is replayed during the autonomous phase usually with little or no cognitive effort. In reality these phases are not distinct but overlap. Given the robot skill definition as a particular algorithm implementation for a specific task, robot skill synthesis is a two-part process: 1) strategy development in terms of the task representation, and 2) strategy translation for implementation on a particular robot/sensor system. Robot skills are difficult to synthesize for a number of reasons. Complex tasks are difficult to capture in a model. Interpreting the sensor signals to extract task information useful for controlling and monitoring the task is very difficult and time-consuming. Yet sensor use is necessary for dealing with task uncertainty and recovering from execution errors. The gulf between the task space and robot/sensor spaces makes strategy translation difficult. That is, strategies which are developed in terms of the task, must be translated and executed in terms of the robot/sensor system. Strategies must be implemented in terms of real-time software; the significant detail required makes strategy implementation tedious and error-prone. Finally, task similarities are rarely directly exploited in the construction of new skills.

Hardware and software composition for rapidly-deployable systems has been pursued over the last decade in the Advanced Mechatronics Lab at Carnegie Mellon University and we are leveraging portions of this past work to develop a system for efficiently building sensor-based skills. Paredis [55] developed the remote modular manipulator system (RMMS) and accompanying task-based design software to create custom, fault-tolerant manipulators for a specific path-tracking task. The reconfigurable software framework of Stewart [66] provides a structure within which to write reusable, real-time software modules based on a port-based object concept. The resulting framework allows the control system designer to focus on algorithm development, not real-time implementation details. Gertz [21] developed Onika, a graphical interface to the Chimera reconfigurable software, which allowed novice users to quickly compose pick-and-place robot task programs. Carriker [13] developed a system to map high-level assembly plans generated with Mattikalli's work in mechanics-based motion planning [38] into real-time software modules for execution. Recent efforts focus on extending the robot programming paradigm from textual, low-level programming to demonstration [28] and composition. The idea is to provide a more intuitive programming interface for the user and to map the demonstration onto sensor-driven primitives or skills for robust execution. Rather than recordings of the robot's kinematic trajectories, the resulting programs are represented in terms of the task and how it projects onto the robot's sensors. This provides robustness to inevitable world modelling errors during program composition and to imperfect control during execution. As part of this effort, Voyles [75] is developing a novel tactile sensor/actuator based on magneto-rheological fluids to provide tactile feedback both during programming for the human and during execution for the robot. This thesis addresses robotic skill acquisition for assembly tasks through building force and vision based sensorimotor primitives and integrating them into a CAD environment to support concurrent task design and task-level skill composition.

1.2 Prior Work in Skill Synthesis

Since fundamentally skill synthesis involves constructing robot programs to perform tasks, previous work in robot programming methods is relevant. More recently, skills have been specifically mentioned in the robotics literature ([1],[24],[28],[32],[53],[64],[71],[79]). Previous work in robot programming falls into four categories: 1) explicit programming, 2) automatic planning, 3) demonstration, and 4) learning. Each of these areas is discussed next in the context of skill synthesis.

Explicit programming of robots requires the translation of a task strategy onto the robot/ sensor system. For pure positioning tasks, robot programming is relatively straight-forward (though tedious), and requires well-calibrated robots and workcells. For contact tasks (e.g. assembly), the program is much harder to create. The chief difficulty is predicting how different task conditions appear in the sensor space. Whitney [78] and Strip [68] both analyzed insertion tasks in terms of their contact states for predicting the sensor space mappings of different task conditions. This analysis was then used to derive a sensor-based strategy for accomplishing the task. Schimmels and Peshkin [57] have synthesized admittance matrices based on task contact models as well. Assuming the task parameters are known and the initial conditions are satisfied, these algorithmic methods are guaranteed to succeed. The difficulty with contact state analysis is that it is difficult to perform on complex tasks. Even relatively simple tasks can have a large number of contact states. Other researchers have used heuristic methods for deriving task strategies for explicit programming. Paetsch and von Wichert [52] synthesized peg insertion strategies using a dextrous hand based on heuristic behaviors observed in human insertions. Michelman and Allen [41] constructed a strategy to remove a child-proof screw cap using a dextrous hand. These methods do not provide a guarantee of success, but are experimentally verified. Both of these efforts ([52][41]) involved the use of hand control primitives to simplify programming tasks with a multi-fingered hand.

What are the weaknesses of explicit robot programming? The level of detail which must be specified for the robot to perform a task is significant. One rarely realizes just how much until one programs a robot to perform a simple task. Add in the error recovery and sensor use, and the programming burden is onerous. The real-time nature of the software implementation requires additional knowledge in the creation of real-time systems. The current languages available for robot programming (e.g. VAL, AML) encapsulate useful robot motion primitives, but they are robot-centered which makes strategy translation a tedious, mistake-prone process. Robot programmers need to be both robotics and task experts; highly-skilled technicians and engineers are required for programming robots, whereas the factory floor worker is the preferred programmer. The need to further ease robot programming spawned the field of automatic planning.

The basic idea behind automatic planning [33][29] is to represent the task in a state space along with operators which transform states. The task is defined as a transition from a given initial state to a final state (or sets of states). The planning problem is to find the sequence of operator instantiations which will achieve that task through (intelligent) search of the state space. Lozano-Perez et al [34] applied this approach as a general solution to planning fine-motions. When the prior knowledge is accurate, this method provides good solutions. However, large state spaces and deep searches create combinatorial search explosion making such planning methods infeasible for many real tasks. In addition, since the *a priori* knowledge is approximate, the resulting plans may not properly execute due to inevitable errors between the real world and the robot's state-space representation of it. Planning is effective at the symbolic level to capture high-level strategies. However, in the author's opinion, pushing automatic planners to handle very low-level detail (e.g. contact states) is ill-advised since the complexity increases sharply and the reliability of resulting plans plummets. The strategies required for fine-motion tasks involving contact are strongly influenced by task design and should be addressed through specific task features which facilitate robust mating behavior (e.g. chamfers).

The third method of robot programming, human demonstration, has been around the longest but is currently undergoing some fundamental advances. Teaching the robot a path by operating a teach pendant (or by physically guiding the robot) is the most direct method of programming robots. This method is often performed by factory floor personnel after some initial training and does not require advanced knowledge of robotics (e.g. kinematics, control, etc.) This method essentially "compiles" the strategy into the robot space very early. However, the resulting programs are very brittle because small workcell perturbations or robot calibration errors will make them fail. Robots executing such programs usually do not use sensors to resolve uncertainty and adapt to changes in the environment. In addition, direct teaching is not appropriate for many tasks involving significant contact since it is usually difficult for the operator to teleoperate the robot to perform a contact task due to poor tactile and force feedback devices to the human.

More recently, "learning from observation" methods have been developed whereby an operator performs the task and the robot observes this demonstration and generates its own program. Kang [28] developed a *learning from observation* system which uses computer vision to observe a task demonstration, segment the strategy, and then map it onto a multi-fingered hand/arm system. *Learning from observation* is a much more intuitive method of programming robots since it allows the human to demonstrate the task through natural performance (i.e. not through the robot). However, significant research issues remain for this method. One of the most significant is the mismatch between the human's effectors and sensors the robot's effectors and sensors [2]. The robot system must be able to sense appropriate task features to guide the strategy. For example, if the task is a close-tolerance insertion and the robot has no force sensor, then it is highly unlikely that a successful strategy will be recovered based on vision sensing alone since the human is relying heavily on tactile and force feedback during the final stage of insertion. So far, the efforts in *learning from observation* have focused on positioning tasks. Further research is needed to extend these methods to contact tasks which require tactile and/or force feedback.

Learning from observation is an example of a larger class of methods called supervised learning. In supervised learning, a teacher (programmer) provides correct demonstrations of the task for the "student" (robot system). Yang et al [79] have applied hidden markov models to skill modelling and learning from telerobotics. The skills learned are manipulator positioning tasks without force or vision feedback. Some researchers are applying supervised learning approaches to recover strategies for contact tasks like deburring. Liu and Asada [32] use a neural network to recover an associative mapping representing the human skill in performing a deburring task. The task is performed using a direct-drive robot with low friction and a force sensor integrated with the workpiece. This allows the human to perform the task will little interference from the robot while much relevant information is measured by the robot sensors. Shimokura and Liu [60] extend this approach with burr measurement information. ALVINN [56] is a car-steering skill acquired by observing a human drive a car and training a neural network on the input road images and the human's steering commands. The advantage of supervised learning methods is their efficiency -- the teacher provides "directed" learning. The primary difficulty with applying supervised learning methods to learn contact tasks is the difficulty in collecting data and/or performing the task through the robot. Because of these problems, most work in applying learning to contact tasks has focused on *reinforcement learning*.

Rather than a teacher, reinforcement learning has only a critic. The difference between a teacher and a critic is that the teacher provides feedback on how to modify the action to improve performance whereas a critic provides only an evaluation of performance. The student is left to discover, on his own, the proper actions to improve performance. In general, supervised learning will result in faster learning because the system is being "taught" how to improve. So why use reinforcement learning? Because a teacher is not always available. This is frequently the case in contact tasks given the difficulty of humans in evaluating sensor signals in the context of the task.

Many researchers have applied reinforcement learning methods to the recovery of a peg insertion skill. Simons et al [62] learn how to interpret a force feedback vector to generate corrective actions for a peg insertion which has an aligned insertion axis. The output motion commands are restricted to the plane normal to the insertion axis. Changes in force are used to reinforce (penalize) the situation/action pair. Gullapalli et al [23] learned close tolerance peg insertion using a neural network and a critic function consisting of the task error plus a penalty term for excessive force. The input to the network was the position vector and force vector and the output was a new position vector. About 400 training trials are required to recover a good strategy. However, the learned "skill" is specific to the peg geometry which it is trained on and is specific to the location of the peg in the workspace because the absolute peg position is produced as output. If the peg location were moved in the workspace, this skill would fail because it would be very difficult to accurately (relative to the insertion clearance) specify the relative transformation between the new location and training location. A few more training trials would probably suffice for learning the new skill, but one does not want to learn and store a different skill for every location in the robot's workspace.

Vaaler and Seering [72] have applied reinforcement learning to recover production rules (condition-action pairs) for performing a peg insertion task. The critic function is a measure of the forces produced from the last move increment; higher forces are penalized. The termination conditions are an absolute Z position (Z is the insertion axis) and a Z force large enough not to be caused by 1 or 2 point contact (common during insertion). Ahn et al [2] learn to associate pre-defined corrective actions with particular sensor readings during iterative training and store these mappings in a binary database. Again, the critic function penalizes moves which increase the measured force. Kinematic analysis of the task can be used to "seed" the database with a priori knowledge, but this is not necessary for the method to succeed. Lee and Kim [31] propose a learning expert system for the recovery of fine motion skills. Skills are represented as sets of production rules and expert a priori knowledge is used. The critic function is the distance between the current state and the goal state, but does not explicitly include an excessive force penalty. The method is tested on a simulated 2D peg insertion task.

This section reviewed the major categories of robot programming relevant to skill synthesis and cited some examples in the literature. Most of the work which specifically pursues skill synthesis relies on reinforcement learning to recover strategy mappings which are difficult to directly recover through task modelling. The next section outlines specific skill synthesis problems.

1.3 Skill Synthesis Problems

Skill Representation. "Black-box" skill representations are seemingly attractive because they hide the internal detail of skills and support the common view of skills as encapsulated programs. In addition, black-box representations are consistent with the view of skills as task-specific and difficult to articulate. Consider, for example, neural network implementations of skills -- each network is a black-box, and portions of those networks are not re-used for similar tasks. O'Sullivan et al's [50] work with explanation-based neural network (EBNN) learning trains various networks with "domain-theory" and then combines them to aid learning a new task. "Domain-theory" refers to the aspects of the tasks which are transferrable. However, the learned task network is independent of the domain-theory networks (their functionality is duplicated in the task network), and the domain-theory is used only to speed learning. *Black-box skill representations do not support the skill transfer which is necessary to rapidly create new skills.* To facilitate skill transfer between related tasks, a skill representation must be developed in which well-defined primitives are combined to form skills. The lack of such a skill representation has prevented the direct exploitation of task similarities when developing new skills. Gullapalli et al [23] exploit a round peg insertion skill to learn a square peg insertion skill, but the final result is two unrelated black-boxes.

The work involved in developing skills can be categorized into *domain-specific* (applicable to other, related tasks in the domain) and *task-specific* (applicable only to the specific task). A task is a very specific problem instance while a task domain is a set of similar or related tasks. One of the key assumptions in this research approach is that skills for similar tasks can share the same sensor-based primitives. The goal with a primitive-based skill representation is to focus on developing domain-specific primitives to effectively amortize the primitive development effort over a task domain. In addition, the existence of a library of task-relevant, sensor-integrated primitives will facilitate the construction of new skills in the domain. Since skills are strategies for complex tasks, the primitives can be associated with subgoals of these tasks. This *divide-and-conquer* approach is an effective way to develop solutions for complex problems. Primitive-based skills may be an effective way to pursue the longer-term goal of *skill-morphing* whereby a new skill is automatically constructed from existing skills. Exploiting task similarities for skill transfer has not been thoroughly researched in previous skill synthesis work.

Sensor Integration. Sensor application is critical to improving the robustness of skills by providing the ability to express and execute the strategy in terms of the task. Tedious and difficult to accomplish, it should be encapsulated for re-use whenever possible. Most robot programming continues the separation of sensing and action. Ahn et al [2] connect task-specific error corrective actions to sensor signals through on-line learning. Erdmann [18] analyzes the information requirements of a task through the design of abstract sensors for a specific task. Other methods [23][32] integrate sensing and action into neural networks which are task-specific. The availability of structured, sensor-integrated commands is very limited: guarded moves and compliant motion are the only common sensor-integrated commands in use today. The lack of sensor-integrated commands forces the burden of sensor integration onto the application programmer for each task instance. It is up to the application programmer to determine how to use the raw sensor data to monitor and control the task. As extracting information from the sensor signals is very difficult, integrating sensors on a task-by-task basis is very inefficient, especially for complex and difficult tasks.

One approach of directly applying sensors to a complex task hides the sensor use inside a "black-box" (e.g. neural network). While sensor-based strategies for difficult tasks can be recovered this way, it continues the per-task integration of sensors. Another method is to break the complex task into reusable subgoals which are tractable for sensor application. An intermediate sensorimotor layer based on these subgoals can integrate sensing into reusable primitives. This would extend the sensor-integrated command library available for robot programming and provide sensor-integrated primitives which could be quickly composed into task-specific solutions.

Programming Complexity. A persistent problem with robot programming is its complexity. Every detail must be specified for the program, which is tedious and error-prone for a human. In addition, the task strategy is developed and represented in terms of the task, yet it must be translated and executed in terms of the robot/sensor system. Sensor integration further exacerbates this problem by increasing the complexity of the programming problem. The existence of robot programming languages (e.g. AML, VAL) has impacted the problem through the integration of robot-centered language primitives. These languages, however, have only robot domain knowledge, not task domain knowledge. *Task-relevant* (not just robot-relevant) commands would significantly impact the programming problem by providing more direct primitives in which to develop and *execute* task strategies. In addition, the judicious use of graphics in a programming environment can significantly ease the programming burden. This is currently evident in the graphical block-diagram building tools of software packages like MATLAB.

1.4 Technical Approach

The thesis goal is to conceive, develop, and demonstrate a framework in which sensor-based robotic assembly skills can be efficiently composed by a human programmer. Rather than try to fully automate skill synthesis, the approach builds skills out of parameterized primitives which are reusable and provides graphical programming environments to facilitate the composition of these primitives into event-driven skills. The framework allows the insertion of more sophisticated algorithms when they are available, but the emphasis is on supporting the human, not replacing him. One difference between this work and that of previous researchers in skill synthesis is that the focus is on building multiple difficult skills, not learning one difficult skill. The three skill synthesis problems outlined in the previous section are addressed as follows: sensorimotor primitives are developed to integrate sensors for a broad class of tasks; skills are represented as finite-state machines of primitives to encourage reuse; and programming complexity is controlled through graphical programming and design agents which assist the skill designer/programmer in the CAD design/programming environment.

Sensorimotor primitives integrate force and vision sensors for a class of tasks (a task domain) rather than individual task instances. The primitives may be thought of as a specialized commands for a particular task domain. This is similar to software packages like MATLAB which facilitates the construction of control system software through incorporation of domain-specialized building blocks. To create such a primitive set, one must first identify what is common or similar about the set of tasks to guide the development of primitives useful for the task domain. For assembly tasks, the similarity is relative-motion constraints. A taxonomy identifies 20 different relative motion classes which are used to guide sensor-based primitive development. Each class has multiple interpretations of the constraint in terms of geometry and mechanics. In addition, the constraints may be defined through contact or non-contact interpretations (or combinations of contact and non-contact). Figure 1-1 shows a graphical depiction of the idea. The primitives occupy an intermediate level between the task domain and the generic sensor and action spaces of the robot. Building primitives requires using task information to interpret sensor signals. Dealing with the raw sensor and action spaces is “high-dimensional” because many details must be specified. More importantly, many of these details are not specific to a particular task. For example, a particular controller, trajectory, and event detector set may be applicable to many tasks. The goal is to capture the reusable development work into parameterized primitives which can be quickly re-applied to other tasks. The interface between the task space and the sensorimotor space is “low-dimensional” in that relatively few details must be supplied to select and

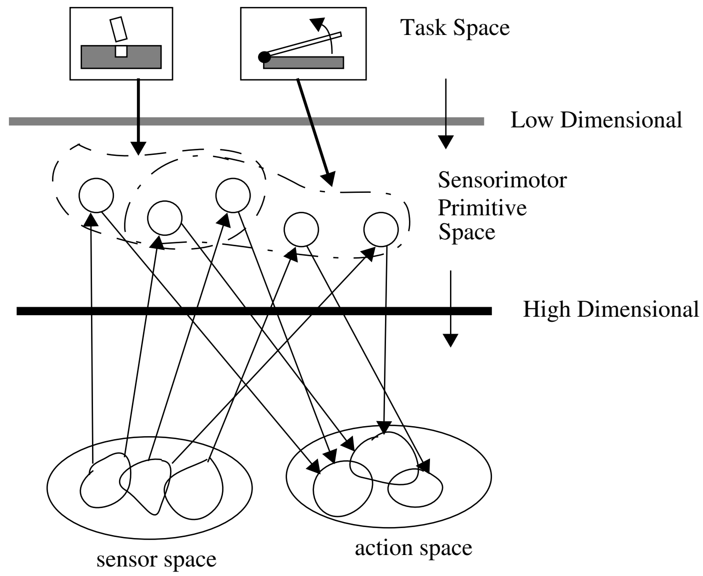


Figure 1-1: Sensorimotor Space

parameterize the primitives into a specific program for a particular task *if the appropriate primitives exist.*

Since skills are typically built for complex tasks, they are usually multi-step algorithms. Finite-state machines are natural representations of primitive-based skills which require discrete changes in the subgoal and corresponding execution primitive. The graphical nature of representing and interpreting fsm's supports GUI-based skill composition. A fsm graphical editor provides the ability to view and edit the state machines which encode skill programs.

Explicit task design for robotic execution is critical. Since a skill represents in some sense the 'hardest' part of the overall application, it is appropriate to consider the execution strategy and the task design concurrently and to involve the human in the development. This *co-design* approach respects that poor task design generally cannot be overcome by more complex execution algorithms. It also respects that automatic planners at their current stage of development are unable to deal with these tasks which are very complex. The primitives represent the most cost-effective sensor-based commands (because they have been previously developed) and should be used whenever possible to maximize their return-on-investment. The tight integration of the primitives into the design environment allows the user to modify both the task design and the strategy design when developing a new task and skill pair.

For exploiting the primitive capabilities at the design stage, the concept of *interactive software components (ISC)* extends software primitives to have both execution and design components. The software composition process then becomes a collaboration between the human programmer and the software components which make up the program. Thus ISC's are active participants in their incorporation into the software. This is an important concept to control programming complexity as software components gain capability and complexity. To demonstrate this concept for the assembly domain, a CAD environment, the real-time system, and a rigid body simulation capable of modeling collisions and impacts have been tightly integrated. The definition of a primitive is extended to include a set of design agents in the CAD environment to support the primitive use. These design agents do not provide design advice, rather they represent the primitive capabilities inside the design environment. The primitive is the execution component while the design agents are the design component of the software. The integrated system allows the designer to instantiate an assembly strategy in terms of available primitives and evaluate performance via simulated execution including force and vision sensor feedback.

Figure 1-2 shows a graphical representation of the framework components and their connections. Underlying the approach are sensorimotor primitives which were introduced above and are discussed in more detail in Chapter 4. Guiding the development of sensorimotor primitives are manipulation task primitives which are indicated by the MTP block in the figure and discussed in Chapter 2. The sensor space refers to a force sensor and vision sensor and the action space refers to cartesian motion capability; these resources are described for completeness in Chapter 3. Primitive-based skills are described in Chapter 5 for six different real-world tasks including canonical peg insertions and difficult connector insertions. The extension to the Chimera real-time operating system for executing these event-driven skills is also described in Chapter 5. The design integration of the sensorimotor primitives is discussed in Chapter 6 which includes the design agents, Telegrip, and SIM blocks in the

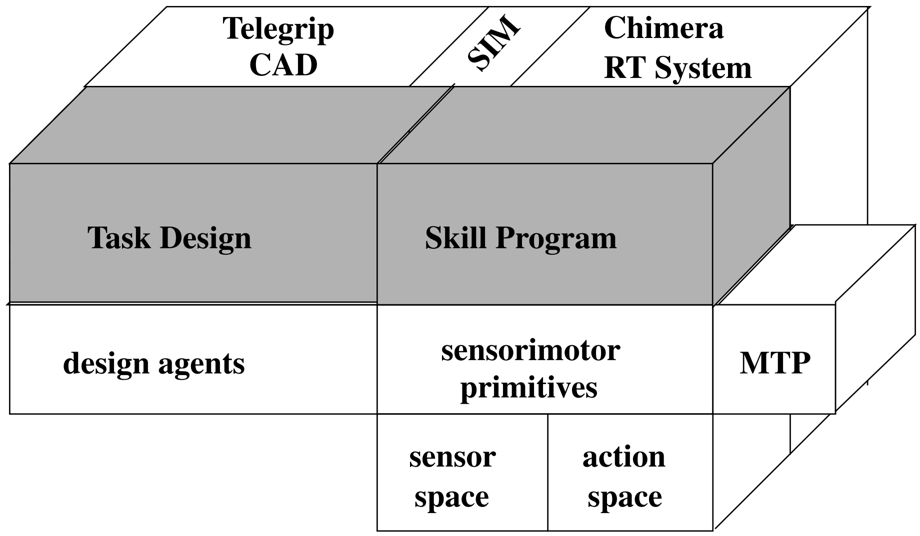


Figure 1-2: Skill Synthesis Framework

figure. Finally, conclusions are drawn, limitations discussed, and future work is outlined in Chapter 7.

1.5 Contributions

- A primitive taxonomy based on relative motion between two rigid parts is used to drive sensorimotor primitive development. My contribution involves both a different representation of the taxonomy and, more importantly, the connection of the taxonomy to executable sensor-based primitives. This extends the use of the taxonomy from an abstract concept to one that motivates and drives executable primitive development. The key concept is that the specific task features which actually define the motion constraint are directly sensed for task control.
- A collection of sensorimotor primitives was built based on the task primitive taxonomy classification. A novel dithering/correlation motion constraint detector was developed along with a combined force/vision primitive for implementing 3 translation DOF constraints. Existing algorithms for guarded moves, accommodation, and visual servoing were also incorporated to realize other task primitives. Sensorimotor primitive reusability is demonstrated through the con-

struction of skills for different difficult tasks using these common sensorimotor primitives.

- Skills are represented as finite-state machines to leverage primitive development. To execute these skills, this thesis extends the Chimera reconfigurable software framework to a “4th level”. This level provides two new objects one which processes events and one which processes both events and data. It complements the periodic, data-driven nature of the 3rd level objects with asynchronous, event-driven 4th-level objects. This 4th level has been integrated with a GUI for editing state-machines and a CAD system for building the skills through a task-level programming interface.

- An important contribution of this thesis is the extension of the sensorimotor primitives from the execution system (Chimera) to the task design system (Telegrip) as design agents. Because any primitive set has limitations, it must be deliberatively exploited during the task design stage. The design agents represent the primitives inside the design environment to facilitate their incorporation into programs; design agents do not modify the task design, rather they assist the user/programmer in selecting and parameterizing the primitives to execute particular steps of the strategy. The broader concept of Interactive Software Components (ISC) considers software objects as having both execution and design components. The execution components appear as part of the final program. The design components interact with the user and task model during the software composition process and range from simple data entry assistance to automated selection and parameterization algorithms. The ISC concept transforms the composition of skill software into a collaborative process between the primitives and the human programmer.

- The seamless integration of the CAD design environment with the Coriolis mechanics-simulation package and the real-time system provides some unique capabilities. Executing the same primitive object code which runs the actual robot removes the possibility of code mismatch between the design/programming (simulation) stage and the robot (real) execution stage. It also provides a useful environment for developing sensor-based algorithms, especially for contact tasks without fear of damaging the hardware. The Coriolis integration provides the ability to simulate contact tasks involving collisions and impacts which is crucial for modelling assembly tasks and providing simulated force feedback. Finally, by combining the ability of Telegrip to generate camera views inside the CAD world with the Silicon Graphics workstation's ability to output video from the screen, 'synthetic' video is supplied to the image processing system. Visual tracking algorithms implement vision-driven primitives instead of pin-hole camera feature projections which assume that visual tracking is possible. With advances in CAD system visual rendering, this capability will be important in modelling the photometric effects which are so important to visual tracking algorithms.

- Six difficult, real tasks were robotically executed using primitive-based skills. The experimental results show the successful execution of these tasks with common sensorimotor primitives. These tasks include canonical peg insertions as well as more difficult connector insertions including one involving a press fit requiring significant force.

Chapter 2

Manipulation Task Primitives

2.1 Introduction

Programming complexity is controlled through the introduction of higher-level programming abstractions relevant to the problem domain. Maple and Mathematica are examples of this approach from the mathematics domain. These packages are targeted at a particular domain and are not generally helpful for a different domain. The benefit of these packages is the encapsulation of domain knowledge in a form which is easily composed into task-specific solutions. The availability of domain-relevant primitives greatly shortens the time required to synthesize a specific task solution compared to using a general-purpose programming language (e.g. C or FORTRAN). The benefit is largely due to the directness by which a problem solution can be expressed in the language -- there is little ‘translation’ which must be done compared to a general-purpose language. This thesis investigates the creation of such a package for a certain class of robotic tasks -- rigid body assembly.

Robotics-relevant primitives and structures (e.g. move commands and homogeneous transforms) are embedded in languages like VAL and AML. However, these primitives are robot-centered and have no information about the task domain -- they are *task-domain neutral*. These capabilities are focused on the robot, and not on a particular task domain. Although they do help the robot programming problem through integration of robot-centered knowledge, they ignore the most difficult aspect -- translation of the task solution into robot primitives. Attempts to automate the translation process through task planners [29] have met with some success, but have not generally solved the problem. One reason why is the lack of powerful primitives in which to terminate the plans. The result of this is that task planners are forced to generate very fine-granularity plans even though their information may not be reliable enough to do so. Besides being computationally expensive (and combinatorially explosive), the resulting plans are rarely robust because some information upon which they are based is suspect. In addition, significant task uncertainty can make developing such a priori plans very difficult, if not impossible, *especially in a robot-centered form*. What is needed is to instantiate a task plan which remains in a 'task-centered' form until *run-time*, at which time it is translated into specific robot motions. To accomplish this requires the development of task-relevant primitives for the particular task domain which are implemented in terms of specific robot resources.

To robustly execute task-relevant functions at run-time, primitives must be task-driven. The overall goal is *task control* and the robot is merely a tool to effect it. Sensor integration must integrate task measurements into the robot control loop for effective task control. Very few sensor-integrated commands exist in robot programming languages, which are designed to be very general (and hence widely applicable). Currently sensors are sparingly used and integrated for specific tasks at the application level. Sensor integration necessarily means introduction of specific task models for interpreting the sensor signals as task information. The most common sensor-integrated command in current languages is a guarded move which terminates motion on a force threshold. The guarded move is used often because it encompasses a common task function (acquiring a contact) and it employs weak task assumptions which are easily satisfied ('stiff' parts).

The key challenge is to create 'general' sensor-integrated primitives which can be re-applied to different but related complex manipulation tasks. Primitives can be evaluated with respect to their generality and their power (Figure 2-1). Higher primitive generality means it can be applied to more tasks. Typically, a very general primitive is domain-neutral (e.g. joint moves or cartesian moves). A very specific primitive is applicable to a single task instance (e.g. a specific instance of a peg-in-hole task). A primitive's power is related to how much uncertainty and ambiguity it can successfully resolve (more uncertainty resolution = more power) and how much information it contains. These two attributes are consistent since typically the resolution of significant uncertainty requires more information to be embedded in the primitive. Typical robot-centered primitives (e.g. cartesian moves) are open-loop with respect to the task and cannot resolve any task uncertainty whatsoever. There is a trade-off between generality and power -- the most general primitives are robot-centered which have little "power" for the task. Powerful primitives for very difficult tasks often have a 'black-box' flavor which severely limits their generality. The middle ground balances generality and power. In the author's opinion, maximizing both generality and power is not feasible -- one is sacrificed for the other. Improving primitive power necessarily requires increasing primitive specificity.

The purpose of this chapter is to propose and develop an approach to identifying manipulation task primitives based on classifying the types of relative motion between two parts. This chapter begins with this classification followed by an extension to frame-based manipulation task models. Geometrical/mechanical interpretations of the classifications are provided. Task primitive definitions are strongly influenced by the resource capabilities. The types of task uncertainty to be considered are outlined and the connection between skills and primitives is discussed.

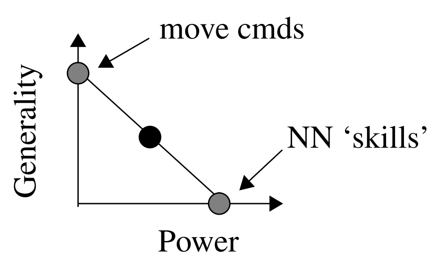


Figure 2-1: Primitive Capabilities

2.2 Manipulation Task Primitive Classifications

A common method of describing manipulation tasks involves the specification of position and orientation of frames attached to the parts. This is depicted graphically in Figure 2-2 where W corresponds to the world frame, B to the robot base frame, E to the robot end-effector frame, and P1/P2 to the frames associated with each part. This thesis focuses on the class of manipulation tasks which can be described as the relative positioning of two parts, one of which is held by the robot and one of which is fixtured in the environment. Assembly tasks can be naturally mapped into this class. Other tasks may be (fundamentally) defined differently; for example, grinding is defined by the removal of burrs along an edge. One strategy is to move a grinding wheel or tool along the edge, but *this relative motion is not the goal*. Instead, a process model is needed to relate the goal (burr removal) to the relative motion which the robot can implement.

A *manipulation task primitive* (MTP) can be classified by a particular relative motion between two parts. This is a more task-centered definition than Michelman and Allen [41] or Speeter [65], who use the term to refer to primitive multi-fingered hand motions useful for manipulation tasks. Speeter generates a collection of coordinated hand joint motions which implement useful finger motions (e.g. grasping or pinching). The effect of these primitives is to collapse a very high-dimensional joint space into a lower-dimensional primitive space

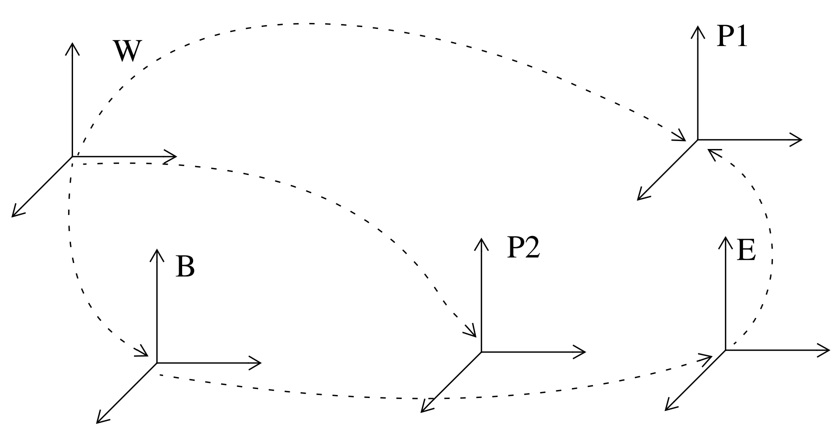


Figure 2-2: Frame-based Manipulation Task Model

which has been carefully designed to capture useful hand joint-patterns for tasks. Our goal is similar except we are incorporating sensors and using a simpler motor resource (only 6 DOF). The goal is the same: to discover patterns in the higher-dimensional space which are useful for tasks and capture them in parameterized primitives. Other robotics researchers have realized the need for such reactive, sensor-based primitives. Smithers and Malcolm [64] combine task-achieving behaviors with a planner for a SOMA-cube world. Hopkins et al [26] suggest the development a force primitive library for assembly but do not provide a methodology for creating it.

There are 64 (2^6) different possible definitions of relative motion between two parts (each of 6 DOF has two possible values: 1=artificial or 0=natural) and 20 of these are unique. Morris and Haynes [42] identify 17 as “reasonable” for describing assembly constraints. We use a different method of representing the relative motions than Morris and Haynes [42]. Rather than list each DOF with a 1 or 0, the translation and rotation DOF for an axis are combined into one symbol which encodes translation/rotation classification (Table 1). The possible DOF (T/R) for each axis can be expressed as a letter from a 4-letter alphabet. A 3-symbol ‘word’ represents the DOF of the task frame (permutations of the same three symbols have same meaning).

Table 2-1: Axis DOF Classifications

Symbol	trans/rot	meaning
a	1/1	both free
b	1/0	t free (only)
c	0/1	r free (only)
d	0/0	both fixed

Table 2-2 shows a classification of MTP’s by the allowable relative DOF between two parts. Each classification is expressed as a 3-letter word drawn from a four letter alphabet describing the 2 DOF (T/R) associated with an axis of a frame. This classification is expressed solely on the frame motion -- but the constraints are determined by specific aspects of geometry and mechanics of the task. Each of the classifications can have multiple geometrical/mechanical interpretations and represent a range of task primitives related by

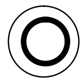
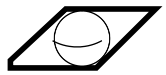
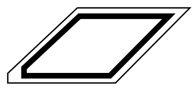
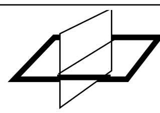
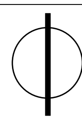
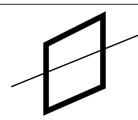
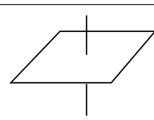
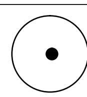
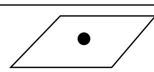
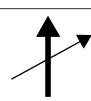
Free DOF		T	R	class
6		3	3	*aaa*
5		2	3	*aab*
		3	2	*aac*
4		2	2	*aad*
		2	2	*abc*
		1	3	*abb*
		3	1	*acc*
3		2	1	*abd*
		1	2	*acd*
		1	2	*bbc*
		2	1	*bcc*
		3	0	*bbb*
		0	3	*ccc*
2		1	1	*add*
		2	0	*bbd*
		0	2	*ccd*
		1	1	*bcd*
1		0	1	*cdd*
		1	0	*bdd*

Table 2-2: Relative DOF Classification

that particular motion constraint. *The motion classification is insufficient to fully define an executable primitive.* Some preliminary interpretations of the geometry/mechanics for these classifications are provided in this chapter. These interpretations may be interpreted through both contact or non-contact task features. Mixtures of contact and non-contact constraint interpretations are also possible. Specific constraint completion depends on the capabilities of the available resources and will be deferred until after the resources are introduced in the next chapter.

The relative DOF classes are shown in Table 2-2. The constraints are illustrated geometrically by splitting the rotational and translational DOF -- two 3D geometric shapes illustrate the constraint type. A sphere represents 3 DOF in rotation or translation. A plane indicates 2 DOF, a line 1 DOF, and a point 0 DOF. These shapes help illustrate the coverage of the primitive classes. For example, 4 DOF can only be made up by (1,3) or (2,2) pairs. The (2,2) pair is seen through the geometric representation to have two cases: parallel planes and perpendicular planes. By using the shape representations, one can see that the classes completely cover the different possible relative motions.

Table 2-3 shows the MTP classifications organized into meta-classes according to specific geometric and mechanical interpretations of the task constraint types. Many of these are familiar constraints and others are difficult to conceive any mechanism or contact situation for. The assembly meta-class includes those primitive classes which have well-defined, stable contact interpretations between geometric primitives (point/plane, edge/edge, etc.). Contact interpretations of constraints naturally maps into the hybrid control of position/velocity and force. In the direction of the constraint, force must be controlled as position freedom is constrained. Two of the assembly primitive classes (*aab* and *abb*) involve fundamentally non-contact constraint interpretations. The second meta-class is “common” mechanisms which indicate constraint sets more easily envisioned as mechanisms than simple contacts. “Common” mechanisms include the ball-and-socket joint and simple hinge/crank. The last meta-class captures the rest of the primitives’ classes. In general, these are related to relatively rare mechanisms and sometimes very contrived ones. The meta-class decomposition is ad-hoc; it is possible that a new interpretation of a primitive class would allow it to

Table 2-3: Manipulation Primitive Meta-Classes

Meta Class	Class	DOF	Geometric Interpretation(s)
Assembly	*aaa*	6	free
	aab	5	edge parallel to surface (no contact)
	aac		1. point against plane 2. edge against edge
	abb	4	surface parallel to surface (no contact)
	abc		edge against surface
	bbc	3	surface against surface
	add	2	1. peg in hole (round) 2. slider in slot 3. crank w/ free handle
	bdd	1	1. sq. peg in hole 2. large-pitch screw
	ddd	0	fixed
"Common" Mechanisms	*bcc*	3	T-slider in slot
	ccc		ball-in-socket
	bcd	2	X-slider in slot
	cdd	1	1. hinge/crank 2. small-pitch screw
Unusual Mechanisms or Contacts	*aad*	4	???
	acc		1. ball-in-slot 2. ring in tube (M&H)
	abd	3	???
	acd		???
	bbb		translation mechanism
	bbd	2	translation mechanism
	ccd		rotation mechanism

change meta-classes. It may also be possible to have the primitive classes straddle meta-classes due to different geometric interpretations.

One advantage of the new representation of the task frame DOF is seeing common sub-patterns in different classes. Three of the most difficult classes to assign geometric interpretations to (*aad*, *abd*, and *acd*) are seen to have the *ad* pair in common; *ad* represents a fully free axis with a fully fixed axis. Quite contrived contact situations or mechanisms are required to visualize these constraints. This similarity was not evident when representing the task DOF with the conventional method (e.g. 110 110).

The reason for classifying manipulation primitives is to help understand the types of primitives which the robot needs to "know how to do" to perform manipulation tasks. The classification helps to guide the primitive identification, definition, and development. From the primitive classifications and our everyday experience, it is clear that robots should know how to operate certain types of simple mechanisms like hinges and sliders, because these mechanisms are commonly encountered in our world. In addition, for assembly tasks, the robot should be adept at operating in different contact constraint regimes which commonly occur between two parts.

2.3 Manipulation Task Modelling

The MTP classification is insufficient for developing sensor-based primitives because it only captures the expression of the constraint(s) but not their definition. The constraints are defined by specific task features including geometry and mechanics. To develop primitives, the manipulation task model must be augmented to include these features. So far, the manipulation task models have been frames and they must be augmented with information about the task geometry and mechanics. The geometry specification includes coordinate systems (frames) and shape descriptions in those frames. The mechanics specification includes physical laws (e.g. Newtonian mechanics and Coulomb friction) and parameters (masses, stiffnesses, friction coefficients, etc.). The manipulation task action is specified as start and goal regions of the moving part relative to the fixed part.

For part mating tasks, the geometric shape uncertainty is small. If it were not, then the task definition would be ill-conditioned (if a peg is too big for the hole, then it makes little sense to define their mating operation because it cannot succeed). The task frame is defined

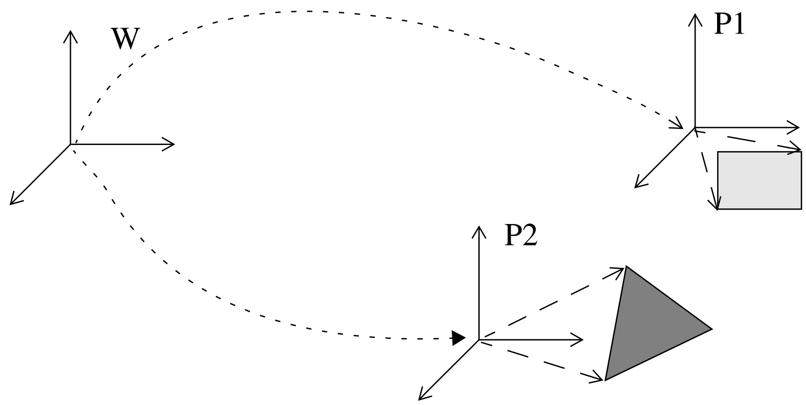


Figure 2-3: Manipulation Task Modelling

by specific (nominal) geometric task features. There are three cases of task frame definition: 1) defined completely by the moving part, 2) defined completely by the fixed part, or 3) defined by features on both parts. The task frame origin is always fixed relative to the moving part and thus moves with it. The task frame orientation, however, may be partially or completely dependent on the fixed part. If the task frame can be defined completely by the moving part geometry, then a hand-fixed control frame orientation can naturally be used. If the task frame is completely defined by fixed part geometry, then a world-fixed control frame should be used. However, sometimes the task frame is determined by features on both parts (e.g. edge/edge contact the constraint direction is determined by the edge cross-products). In this case the task control frame should be updated dynamically according to part motion.

The geometric pose uncertainty can be significant, especially relative to the mating requirements. This uncertainty can be reduced for the fixtured part by on-line sensing to locate it in the workspace (e.g. using an overhead camera) or by careful initial positioning. For the grasped (or controlled part), the grasp operation must be carefully executed to minimize the introduction of errors in the grip transform. The grasp operation also requires knowledge of the world-location which can benefit from either sensor-based location or careful initial positioning. We can also employ real-time sensor feedback during the grasp to reduce the grasp error. Since we will often use a hand-fixed task control frame, grip transform errors will introduce task control frame errors. Often the gripper will rely on friction to complete force closure (e.g. two-fingered grippers with flat gripping surfaces). Task forces can cause slipping of the part in the gripper and lead to additional uncertainty in the grip transform. Understanding the sensitivity of task control to grip transform errors is important. Ideally, primitives should be robust to grip transform errors since these errors cannot be easily removed.

The mechanics component of the task model is even more uncertain than the geometric component. First, the physical laws are only approximate representations of the real natural laws. While Newton's law is fairly accurate, the Coulomb friction law that we use is a very simple representation of a very complex phenomenon. The parameters (e.g. mass and friction coefficients) used in these laws are also very uncertain. Any strategy which requires precise knowledge of these parameters is likely to fail in practice. It is important that primitives use mechanics information qualitatively and not rely on accurate prediction or measurement of task forces.

The task goal is inherently defined by relative positioning between the two parts. Pose uncertainty makes executing this positioning difficult because even if the absolute control fidelity is very good, the knowledge about the (absolute) controller setpoints is often poor relative to the task requirements. One approach is to recover the absolute part positions in 3D space through sensor measurement (e.g. laser rangefinder or stereo vision). One problem is that the sensor and robot must be very well-calibrated -- their world views must coincide very closely. The second problem is that controlling the relative part position through long kinematic chains also requires excellent calibration so that absolute positions are very accurately known. In this thesis, the task execution is based on relative rather than global task measurements by using end-point sensing which reports task relative position information. This approach is insensitive to calibration errors of both the robot manipulator as well as the sensor/robot combination.

2.4 Current Robot Programming Primitives

Current robot programming primitives are nearly exclusively motor commands which are devoid of task-relevant sensing. Although the robot uses feedback loops around the joints, these feedback loops are robot-centered and not task-relevant. As referenced before the lowest level commands to the robot are joint trajectories. Tasks specified as such tend to be very brittle since all reference to the task is lost prior to execution. Common examples include spot welding, spray painting, and pick-and-place assembly. To be successful, this approach requires very tight control over the task environment.

The next step is to at least share the motion space -- cartesian space provides a convenient shared ontology for describing both robot motions as well as task motions. Cartesian motions can be specified relative to a hand frame or a world-fixed frame. A hand frame-based motion is generally ‘differential’ from the initial hand location. Orientation errors in the motion vector will generate increasing positional errors with distance travelled. Such an approach offers the capability of using local reference and calibration to resolve some uncertainty. However, the translation process is still quite early, and the task execution specification in terms of frames is quite abstract which undermines handling greater degrees of uncertainty.

What is missing from the current robot primitives is task-relevance. To achieve task relevance generally requires sensing appropriate task features and constructing feedback algorithms which operate directly on this information. With the exception of the common guarded move, robot programming primitives are devoid of sensing and therefore of any uncertainty resolution capability. Rather than construct planners which attempt to resolve uncertainty on a per task basis, we are focusing on providing more task-relevant primitives which can resolve uncertainty on their own. Planners could avoid combinatorial explosion by terminating in such primitives. The first requirement of such primitives is to be reactive and closed-loop around task measurements. More sophisticated primitives might include a planning capability which would enlarge their sphere of applicability.

2.5 Resources, Strategies, and Task Uncertainty

Teach and Replay. Considering the resources and the amount of task uncertainty leads to different classes of robotic task strategies. For very small task uncertainty, the approach is to translate the task strategy into robot joint trajectories for replay. The most successful applications of robotics -- spray-painting, welding, and pick-and-place -- take this approach. It requires very precise calibration and accurate robot control and is very sensitive to errors in either. In addition, it can tolerate very little task uncertainty -- only very small perturbations about the 'taught' trajectory are likely to still accomplish the task. Relatively unskilled operators can teach the positions or paths. With this approach, the robot is used as a piece of hard automation replaying the same path over and over again. It exhibits no intelligence or adaptability to task circumstances and so the resulting programs are very brittle. Humans and robots must be carefully separated for safety reasons.

Measure and Move. To deal with more task uncertainty the above 'teach and replay' approach must be abandoned and sensors must be introduced. Since the robot is designed to be an accurate positioning device, the most direct method is to use sensors (e.g. cameras) to recover absolute setpoints for the robot. In addition, Cartesian control is used to allow straight-line motions to be executed which is an important class of motions for grasping and approaching for part mating. The problem with using absolute setpoints derived from sensors is that precise calibration is required between the sensors and the robot. Errors in this calibration severely degrade the ability to exploit the robot's accurate control. It can also be difficult to measure the part absolute locations to the necessary accuracy to exploit the robot's accurate control. Limitations in control fidelity and/or setpoint accuracy while near contact makes compliant motion necessary to regulate contact forces.

Measure while Moving. To reduce reliance on accurate calibration, task information measured while moving is incorporated into the robot control loop. Doing so makes the robot-motion task-driven during motion instead of discretely determining a (task) setpoint and then executing it in open loop (with respect to the task). Faster computation has opened this alternative approach. The advantage is that the control loop is closed around the task measurement at high bandwidth so that sensor/robot calibration errors are rejected. One may even be able to use less precise robots (which would be cheaper) because extremely accurate absolute positioning is also not required. With endpoint sensing more endpoint compliance can be introduced to improve contact stability margins. One reason compliance is not introduced is because of the reliance on rigid chain kinematics to infer the position of the part -- if the part position important to the task can be measured, then some compliance can be introduced while preserving fine controllability of the endpoint part.

What emerges in this discussion is the tight coupling between the resource capabilities, strategies, and task requirements (Figure 2-4). The ultimate goal is bridging the task and robot spaces with sensor-integrated commands. This chapter has focused on understanding manipulation task primitives based on relative motion constraints between two parts. But the motion classifications must be augmented with interpretations of the constraint enforcement by specific task features. The resource capabilities (e.g. force and vision sensors) strongly influence the types of strategies that can be employed which in turn influence the types of

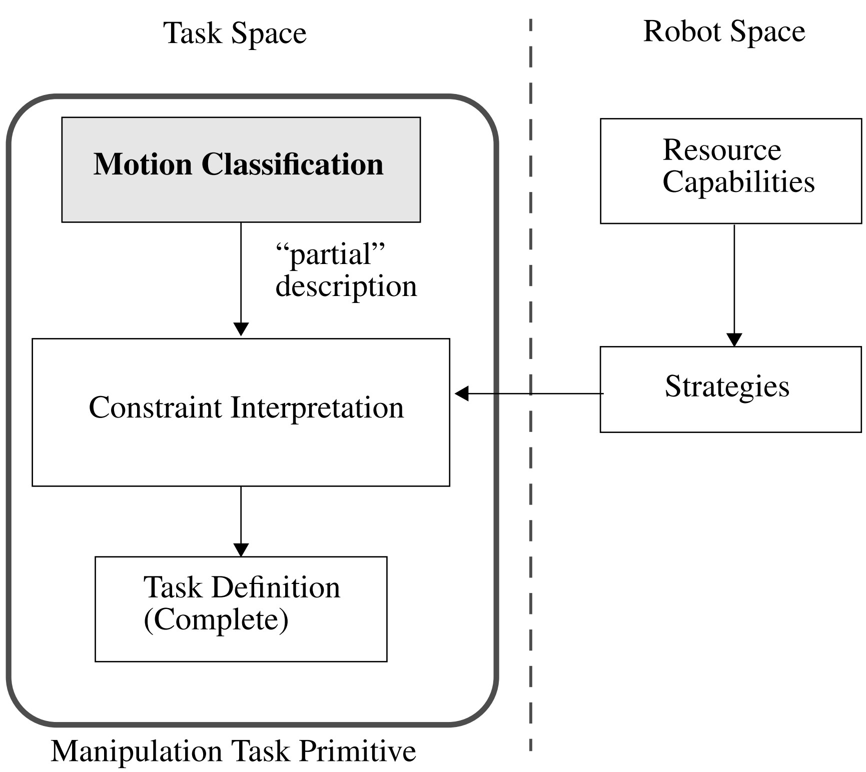


Figure 2-4: Manipulation Task Primitive

tasks which can be executed. The possible strategies supported by the available resources should influence the task primitive definitions which are driven by the relative motion classification. The strategies encompass not only the sensor capabilities but the programmer's understanding of how to apply them to the task domain.

2.6 Complex Tasks, Primitives, and Skills

From an application programmer's point of view, skills and primitives are equivalent: robust, parameterized solutions to recurring tasks. From a 'domain-programming' point of view, skills are composed of primitives. Primitives are developed by hand and are expensive; the idea is to amortize their development cost by reusing them for many different tasks. A *skill* is typically a solution to a more complex task which does not have a primitive solution. Such tasks may involve more uncertainty than a single primitive can resolve and the strategy will generally require multiple primitives to be used in a state-machine control architecture. Non-linear behavior is accomplished through 'piece-wise' combination of the primitives. Complex tasks require the composition of existing primitives and/or the creation of new primitive(s). Primitive composition is cheap because it leverages previous work; primitive development is expensive because it involves task modelling, sensor mapping analysis, and detailed code development. From the 'domain-builders' point of view, skills are divisible, while primitives are not. The task-programmer does not care about the divisibility, only what the task function is. For example, a guarded move is a primitive while a connector insertion is a multi-step skill with a guarded move primitive used in one or more steps. A new task instance will prompt a search to see if a current primitive (or skill) will solve it. If not, it is decomposed into existing primitives to maximize the reuse of work. This approach can be extended one step further by designing complex tasks explicitly as compositions of available primitives. Without a broad set of primitives, however, the range of possible tasks which can be designed will suffer. MTP classification is one tool to try to ensure a primitive set with broad coverage of basic manipulation motions.

The same task with different parameters might be considered both primitive and complex. For example, primitives exist for solving close tolerance insertions [68] but typically these primitives have fairly small applicability regions (i.e. deal with relatively small uncertainty regions). So the 'same' insertion task with a larger initial uncertainty region would be considered complex if no primitive existed to solve it. One decomposition of it could include a more limited insertion primitive. The earlier steps might focus on ensuring that the preconditions of the final (limited) insertion primitive were met.

2.7 Summary

In this chapter, a classification for manipulation tasks in terms of relative motion between two parts was proposed. Complete task primitive definition must be driven not only by this classification (to encourage generality), but also by the resource capabilities which define the types of strategies which may be employed. The ultimate goal is to create more task-relevant robot programming primitives which integrate sensors to resolve uncertainty. The cost of doing so is to restrict the primitive set to a particular class of tasks (e.g. assembly). This represents a customization of the robot programming language to a particular task domain. Such an approach has been very fruitful in other domains (Mathematica & Maple for mathematics, and Matlab/Simulink for control systems). Understanding the task domain through primitive development can illustrate the need for specific types of resources (e.g. sensors or algorithms) to solve recurring, primitive tasks. Although primitive development is still expensive (and human-intensive), the reuse of such primitives allows amortization of their development costs over many task instances. Finally, by providing more capable primitives, planning complexity can be controlled. In AI it has been realized for some time that the path to more capable systems lies not necessarily with more powerful search engines, but with more knowledge. This research follows this approach by incorporating more task knowledge into the programming primitives.To build skills which are expressed in terms of the task requires sensor integration. The next chapter discusses specific resource capabilities using force and vision including the basic control architectures and their characteristics and limitations. Based on these capabilities, a number of specific primitives and algorithms for solving them with those resources are introduced in the following chapter.

Chapter 3

Sensor and Motor Resources

3.1 Introduction

In the previous chapter, a classification for manipulation tasks was introduced to guide sensorimotor primitive development. Fully defining the manipulation task primitives required requires considering the specific resource capabilities. Those resources are introduced in this chapter. The 'motor' resource is a 6 DOF robot with a pneumatic two-fingered gripper. Command and control of Cartesian motions is supported along with joint-level control. The sensor resources include a 6 DOF wrist force sensor and a CCD camera with 480x512 pixels at 30Hz framerate. Force and vision are complementary sensors for manipulation since force provides information in contact and vision provides non-contact sensing with a wider field-of-view. The purpose of this chapter is to describe how these resources are integrated. For each sensor, the disturbance sources which corrupt the task information present in the signal are outlined. Based on the resource descriptions here and the MTP classifications in the previous chapter, specific MTP/SMP's are developed in the next chapter. Figure 3-1 shows where the resource capabilities fit into the overall picture of primitive development.

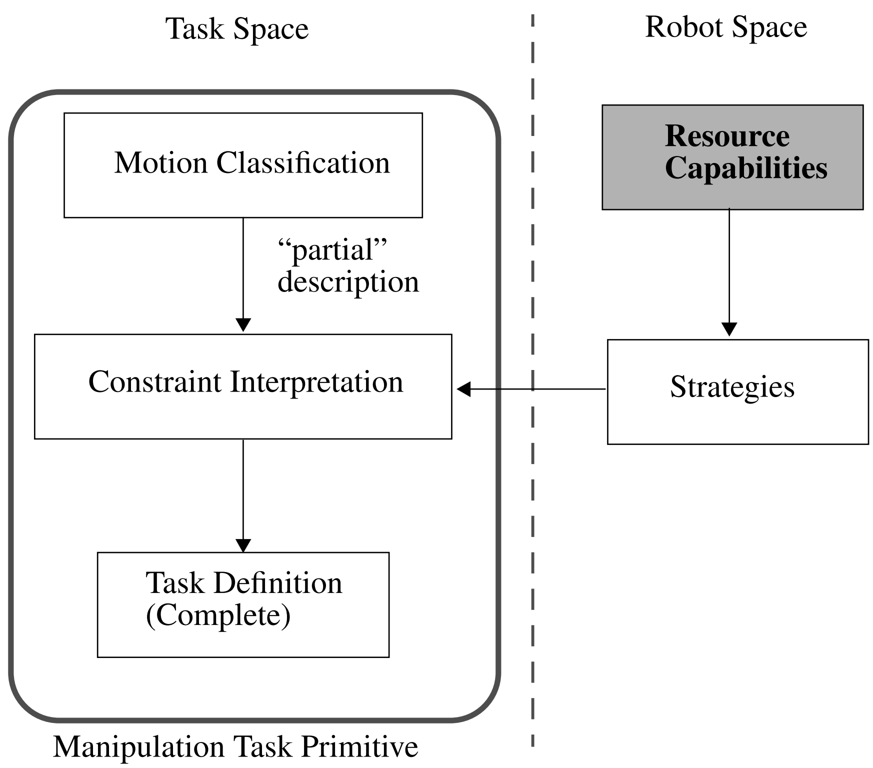


Figure 3-1: Resource Capabilities

3.2 The robot: a 'motor' resource

The robot is controlled by individual PID control loops about each joint and Cartesian controllers supply joint setpoints to these controllers. The stiffness of the joint controllers ensures that the joint setpoints are closely followed. The lowest level method of commanding the robot is to supply the joint trajectories. Task-relevant commands requires a Cartesian controller which provides a common space in which to describe robot motions and task manipulation motions. There are two basic choices for cartesian robot control: resolved-rate and inverse-kinematics. Resolved rate control involves specifying a velocity vector to drive the endpoint of the robot. The robot jacobian, J(q), maps the changes in joint angles to changes in the task frame position and orientation. For non-singular manipulator configurations, the Jacobian is inverted and used inside the control loop to map desired incremental changes in task frame position into incremental changes in joint setpoint values. These joint values are then fed to individual, stiff PID joint controllers. Resolved-rate control is especially useful for trajectories which are determined at run-time -- for example from teleoperation input or sensor-feedback. Resolved-rate control is ‘open-loop’ with respect to the cartesian space because absolute cartesian setpoints are not used (only cartesian velocities).

In inverse-kinematics, a task frame defined in the absolute world frame is mapped into a robot joint vector (again, assuming non-singular configuration of the robot). The absolute cartesian goal position is identified and then a cartesian trajectory generator generates intermediate cartesian goals which are mapped via inverse-kinematics to robot joint values. This method is closed-loop with respect to the cartesian space since absolute cartesian setpoints are used as input. The inverse-kinematic approach makes sense when the manipulation task strategy is defined in terms of absolute world positions.

The resolved-rate approach to cartesian control is better suited to on-line determination of robot motion (e.g. teleoperation or sensor-based control), while the inverse-kinematics approach is better suited to replaying known absolute cartesian trajectories. Since significant task uncertainty requires the on-line determination of robot motion, the resolved-rate controller is the natural choice. Task-based velocity setpoints for the resolved-rate controller are generated by sensor measurements which allows the control to be task-based, rather than robot-based or world-based.

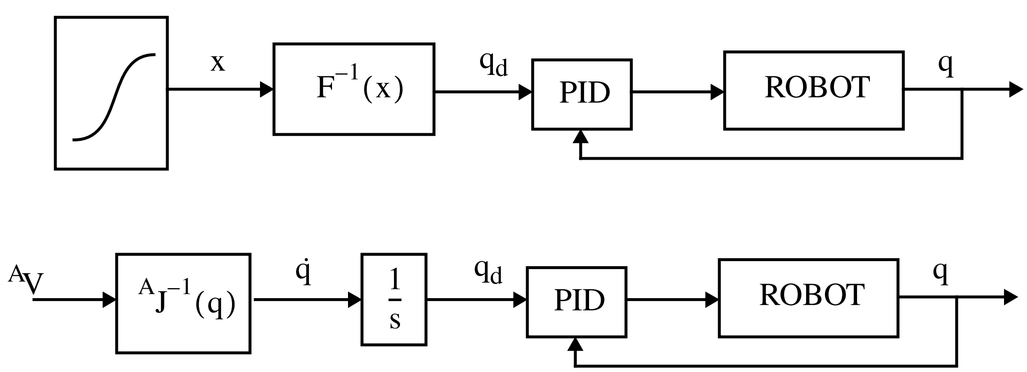


Figure 3-2Cartesian Controllers around Joint Controllers

Cartesian resolved-rate control is implemented around individual joint control loops. A task frame is specified on the hand -- the *location* is expressed fixed to the hand frame, while the *orientation* may be hand-fixed or world-fixed. A 6 DOF velocity vector is specified to move the task frame origin (and hence the robot hand). Errors in task frame orientation will cause accumulating errors during a move. The lack of kinematic redundancy in the robot requires the Cartesian moves to be relatively small to avoid robot singularities. Since we are focusing on fine-motion or gross/fine motion transition, these small Cartesian motions (~10 cm) are adequate.

The resolved-rate cartesian controller is implemented with Chimera [67] reconfigurable modules (Figure 3-3). These modules are port-based objects with well-defined input/output and data processing functions. The advantage of the Chimera architecture is that modular real-time software components can be quickly developed and tested. The velocity setpoint may be generated by many different modules. The *movedx* module is an open-loop module which generates a constant velocity vector for a specified time to execute a cartesian move. Using sensor-based modules to generate the velocity vector will be discussed later.

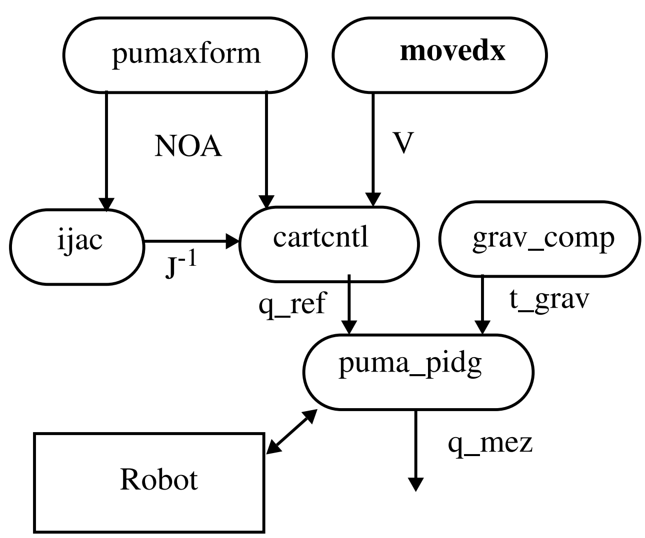


Figure 3-3: Chimera Resolved-Rate Cartesian Controller

Module	Function
pumaxform	computes task frame relative to world given the task frame definition in terms of the hand frame
puma_pidg	• computes PID control laws for each joint and writes torque values to robot • writes out measured joint values
grav_comp	computes gravity compensation torques for joints 2 and 3 of PUMA 560
ijac	computes a 6x6 inverse jacobian matrix for Puma 560
cartcntl	maps the task velocity vector into joint increments and adds them to the current joint vector
movedx	given a desired cartesian differential move (e.g. delta x), the module generates an open-loop constant velocity vector for the appropriate time to effect the move

Table 3-1. Chimera Modules for Resolved-Rate Control

Like other aspects of the primitive, the controller is defined about a specific task frame. The velocity-based motion command has two parts: translation and rotation. A translational error in the frame placement only generates an error due to the non-zero rotation part of the command. The translation portion of the command remains unaffected although the absolute trajectory is offset. An orientation error in the task frame, however, affects both orientation and translation commands. A pure rotation occurs about the wrong axis and a pure translation incurs linearly-increasing errors with distance travelled. The Cartesian robot move primitives are *motor* primitives with respect to the task. The sensorimotor primitives developed in this thesis produce a task-based velocity vector as the ‘motor’ command. Force sensing is incorporated into damping force control to provide compliant motion capability.

3.3 Force Sensing and Control

The robot is an *impedance* -- it imposes a position on the world and accepts a force in return, while the world/task is an *admittance* which accepts a position from the robot and imposes a force on it. The robot controller is built very stiff to reject all force disturbances and achieve its goal position. High controller stiffness is important so that joint values can be supplied and the robot will closely track them in spite of disturbances (e.g. unknown information about the mass, friction, and damping properties of the robot and well a coupling torques from other joints). The fundamental problem is that two disturbances are present: friction and the contact force, and the robot should accept one but reject the other. The friction disturbance is very difficult to predict and must be rejected to get accurate control. The contact force, however, should not be rejected but complied to -- unfortunately, the joint control loop does not discriminate between these two types of disturbances and rejects both.

Force control is necessary when performing tasks involving contact between parts to regulate the contact forces which the joint controller views as disturbances. Force sensors can deliver contact information only when contact is attained. Since the robot is very stiff, the necessary task condition is that the parts have sufficient stiffness to generate forces when in contact. The task information in the force signal pertains to contact, but the signal is corrupted by inertial and gravity forces as well. The mass of the gripper can be several pounds which introduces both gravity loads and significant noise from mass vibration during motions. Processing the force sensor signal to extract task-relevant information requires understanding these sources of signal noise and adequately compensating or avoiding them.

3.3.1 Sensor Configuration and Signal Processing

Initially the measured force at the sensor frame must be transformed to the task frame. *The task frame definition is the sensor placement for force sensing*. Placement of this task frame is task-dependent. The task frame origin is always fixed relative to the hand and thus moves with the hand. As discussed earlier, the task frame is defined by specified geometric features on one or both parts. Recall that the parts are considered either controlled (by the robot) or fixtured. Thus the task frame may be defined fixed in the moving part (and to the hand), fixed to the world (and fixtured part), or a hybrid frame which depends on the contact between the two parts. In this thesis, the task frame is always fixed relative to the held part which allows a constant transformation between the sensor and part to be used to define the task frame (and which assumes that the part does not move relative to the sensor -- i.e. no slip).

Before transforming the force signal to the task frame, it is preprocessed. Note that besides the contact forces, gravity and inertial loads also influence the signal. Three steps preprocess the force signal: low-pass filtering to remove noise, biasing, and transforming from the sensor frame to the task frame. Due to gripper mass beyond the sensor and control-induced vibrations, the signal noise is significant. The force signal is filtered through a first-order, low-pass filter to attenuate high-frequency noise at the expense of introducing some phase lag. A bias term is recorded at the beginning of force control for subtraction from the force signal. If orientations do not change very much during the force control, this effectively removes the gravity load from the signal. Orientation changes will introduce a gravity disturbance since the original bias term will not completely cancel it. Since there is accurate orientation information available, a gravity compensation feedforward term can be computed based on the knowledge of the center of mass (COM). For handling very light parts, the gripper inertia dominates the COM. For heavier parts a grip transform is needed. But even if an accurate COM is known, measurements on the sensor indicate a variation of ~10% of gripper weight *magnitude* over significant orientation changes. Obviously, since

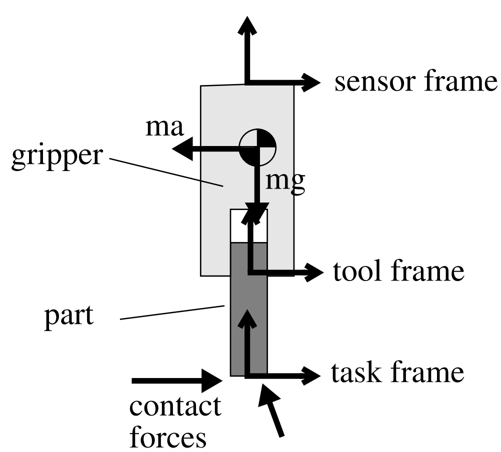


Figure 3-4: Acting Forces

the mass is not changing the measured weight should be constant. Using a more advanced calibration technique [74] can reduce this variation to ~5%, but completely cancelling the gripper weight with a feedforward term is not easy. When parts are in contact (or very near) low velocities are required for (stable) low-gain force feedback, and inertial loads due to acceleration are ignored.

Subtracting the constant bias term gives an estimate of the contact force in the sensor frame. The force signal is transformed from the sensor frame to the task frame using (3-1):

$$\begin{aligned} F_T &= {}^T R_S F_S \\ \tau_T &= {}^T R_S \tau_S + {}^T R_S ({}^S r_{TS} \times F_S) \end{aligned} \qquad (3\text{-}1)$$

where F_S and τ_s are the measured force and torque, ${}^T R_S$ is the rotation matrix of the sensor frame relative to the task frame, and ${}^S r_{TS}$ is the vector from the task frame to the sensor frame expressed in sensor frame coordinates.

To use the measurement in control it is subtracted from a reference force (in task frame) to get a force error. Since the force error is mapped to velocities in the task frame, small errors may cause drift by commanding a non-zero velocity. Because of this, and because the error will rarely be exactly zero, a deadzone is used to ignore small errors. The computed error is passed through this deadzone before it generates a compensating velocity. Different spherical deadzones for force and torque suppress the drift effects which small errors will introduce. Unless the force vector penetrates the sphere surface, the result is zero. When the force does penetrate the sphere, the direction is preserved but only the magnitude beyond the sphere is used to generate a force error. This provides smooth operation when leaving the deadzone. Using a deadzone helps to compensate for unknown disturbances but can introduce significant errors for control purposes, especially if very small contact forces are desired (on the order of the deadzone). Ultimately, larger force setpoints improve the signal-to-noise ratio but some tasks are not suited to large force setpoints (e.g. those involving fragile parts and close-tolerance insertions where wedging can occur). When small force setpoints are necessary, force feedback which is “closer to the task” (e.g. fingertips) is probably

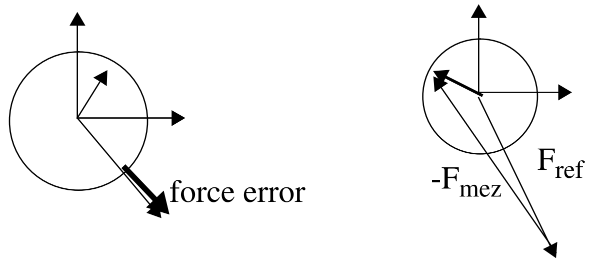


Figure 3-5: Force Error Deadzone

a better solution than wrist force sensing.

3.3.2 Control

Our basic force controller is damping force control [37] which translates force errors into velocity perturbations (Figure 3-6). The controller accepts both force and velocity setpoints in the task frame for input (V_d and/or F_d). Selection of force and velocity controlled axes is done via 6x6 diagonal selection matrices (S_V and S_F). Hybrid control specifies either force or a velocity command along each degree of freedom which amounts to $S_V + S_F = I$. This explicitly sets one of the components to zero. Some tasks however, require a zero force but non-zero velocity along a particular DOF. Using a hybrid control approach requires the force error to generate the required velocity. Such tasks can benefit from supplying *both* types of setpoints to the same axis -- zero force plus a model-computed feedforward velocity. Errors in this feedforward velocity will be compensated by the force loop, but the feedforward term will reduce the force disturbance.

Force control stability is a well-known problem in robotics [73][17]. The crux of the

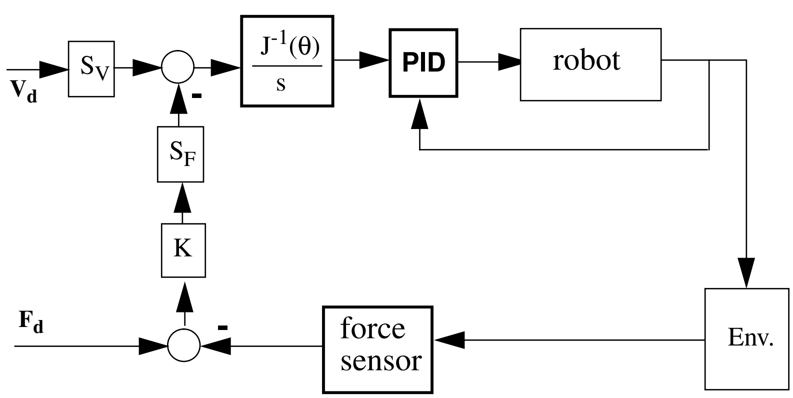


Figure 3-6: Damping Force Control

problem is causality violation -- stiff robots contacting stiff environments lead to marginally stable systems. When contacting stiff environments, low force-feedback gains improve the stability of the system at the expense of response bandwidth. This trade-off means that slower command velocities are necessary to limit the contact force transients. Lowering the stiffness at contact (passively) improves the stability margins of force control but lowers the theoretical bandwidth of force tracking. One of the main obstacles to introducing endpoint compliance is violation of the rigidity assumption used to determine the robot (or tool) end-point position from joint angles using a kinematic chain. If the end-effector position can be independently sensed, then closed-loop control can be used to reject compliance-induced errors.

Many different force control laws are possible and the appropriate selection depends on the specific task. Rather than select a non-parametric control representation (e.g. a neural network [4][23]) capable of non-linear mappings, we will consider linear mappings between force errors and task velocities. Non-linear task control can be realized through event-driven controller transitions. One reason to avoid the nonparametric approaches is to preserve the connection between controller parameters and task design parameters. Determining controller gains is driven by the task mechanics model and task frame placement. Other researchers [37][10] have studied in detail the synthesis of control laws for many of the manipulation task primitives with a contact interpretation: peg-in-hole, crank, slider, etc. This previous work can be leveraged to develop sensorimotor primitives for those tasks. Because any strategy has necessary preconditions (i.e. initial requirements), we will use both force and vision sensors to enlarge the ‘width’ of the entrance funnel to the primitive (i.e. make the primitive as ‘powerful’ as possible).

A task frame error will result in mixed control along the task directions. That is, both force and velocity will be controlled along the same task DOF to an extent. For the case a constant task frame orientation error, task frame errors produce coupling disturbances between force controlled-directions and velocity-controlled directions. Figure 3-7 shows graphically the true task frame along with a task control frame with error in it. Controlling force along the Y and velocity along the X task control frame directions will yield coupling disturbances because of the task frame error.

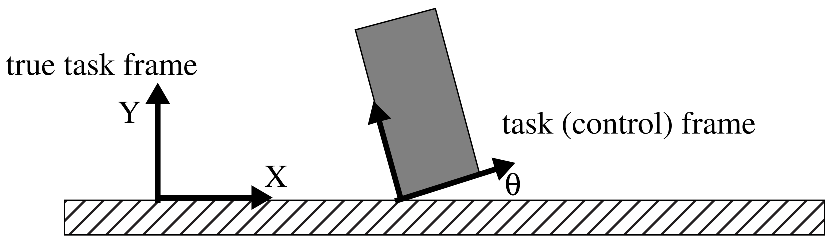


Figure 3-7: Task Frame Error

$$F_Y = F_c \cos(\theta) - \frac{V_c \sin(\theta)}{K_F}$$
$$V_X = V_c \cos(\theta) + F_c \sin(\theta) K_F \tag{3-2}$$

where K_F is the force feedback gain and F_C and V_C are the setpoints provided in the task control frame. F_Y and V_X correspond to the force and velocity realized in the actual task frame. These equations do not include the disturbance effect of a frictional contact.

3.3.3 Trajectory/Setpoint Specification

The force signal is not used to generate a setpoint, but to perturb it. For example, a force setpoint may be specified and the force measurement is used to achieve it. Or a specific velocity trajectory may be commanded and the force feedback will modify it in the face of contact which generates forces. Generally, prior knowledge or non-contact sensing like vision is used to determine the inputs to the force controller. As mentioned in the beginning of this section, force feedback only delivers task information during contact and orientation changes are limited under force control as well. To enlarge the preconditions of primitives to accommodate more initial task uncertainty, a wider field-of-view, non-contact sensor is needed. The next section discusses the incorporation of visual feedback in the robot control loop to effect task-driven control.

3.4 Visual Servoing

Weiss [76] first studied the incorporation of vision directly into the robot servo-control loop and today the technique is known as visual servoing. A recent survey can be found in [27]. Image-based visual servoing is used to incorporate vision into the feedback loop because it supports closed-loop control around task errors and because it is robust to calibration errors between the camera and manipulator. The traditional look-and-move approach to incorporating vision into robot control is open-loop: an image is sampled and analyzed and then an absolute setpoint is sent to the robot controller. *During* the robot move, no vision feedback is used. This approach was largely driven by the processing time of the image (on the order of minutes or longer) and requires very accurate calibration between the robot and the camera. The evolution of faster computing opened the alternative method of visual servoing. Features on the image are used to generate control inputs to the robot at the camera frame rate (30 Hz). The main advantage of this approach is the rejection of calibration errors between the camera and robot due to the closed-loop nature of vision integration.

The price paid for this robustness is simpler vision-processing algorithms and limited use of the vision information. A full-resolution image (512x480) at a 30Hz framerate delivers about 7Mb/s of information. Features are selected and tracked which reference only a fraction of this information so that the tracking can be done in real-time. Noise and other factors disturb the tracking effort and redundancy is typically used to add robustness at the price of tracking speed -- there is a delicate trade-off between tracking speed and robustness. Frame-rate tracking speed is achievable now with off-the-shelf, affordable hardware, but improved robustness is needed for real environments.

In order to use visual servoing, features must be tracked on the image plane and used to generate errors which are converted to a task frame velocity by inverting the image jacobian. Fixed-camera (rather than eye-in-hand) visual servoing where the camera is placed to image both parts supports measuring directly the task error defined by relative part positioning. This section reviews several fundamental aspects of visual servoing: feature selection and tracking, image-plane error computation, image jacobians which map task motions into feature motions on the image plane, and fusing the commands generated by two different camera views.

3.4.1 Feature Tracking

Features are usually subportions of the image which can be tracked in real-time. This is a critical requirement of visual servoing to achieve closed-loop control and reject calibration disturbances. In addition to being trackable in real-time, features must be task-relevant and, for primitives, recurring. Task-relevant means that they can report task information which is useful in guiding and monitoring the task action. Recurring means that the features are fairly common and are expected to be useful for many different tasks. This is important so that primitives which are based on the features can be reused.

Two types of features have been used in this thesis: SSD windows and corners. SSD tracking involves selecting a rectangular window as the feature and then performing a search in the next image to find its location, which minimizes the *sum-of-square differences* (SSD) between the template and the image [3]. This rectangular-window feature is considered a "point" feature for control purposes. The feature selected should have strong gradients in both directions to provide strong discrimination information in both directions. The SSD approach is fairly computationally expensive as it requires performing a correlation of the template with a fairly wide area on the image to locate the best match. To speed this up, the image and feature are subsampled and the computation is performed in a 3-level, coarse-to-fine pyramid search. Using this technique, four 8x8 features can be tracked at 40Hz on the Datacube Max860 system.

SSD feature tracking has several problems. First, it is very sensitive to illumination changes. Second, it is really only appropriate for internal features which do not include both foreground and background pixels. And third, the templates are sensitive to rotations which generally warp their appearance in the image and depth/scaling which changes their size. If a template includes background pixels and they change significantly, a feature template which includes them will differ from the reference template. This is especially apparent for some types of tasks (e.g. assembly) which have defining features naturally occurring on the occluding boundaries of parts and therefore mixing foreground/background pixels. Care must be taken to select/define features with strong gradients in two nearly orthogonal directions to provide the information to locate the template in the image. If the image contains similar-looking patterns to the feature template, the tracking algorithm can be easily fooled during the search process. Resampling the feature template window at each tracking update can alleviate some of these problems (e.g. illumination changes and rotations), but is not a general solution to the problems inherent in SSD tracking.

In response to these problems, we have developed a different feature tracker based on a technique introduced in [70]. The basic assumption is that foreground color or intensity of the tracked object has enough spatial and temporal stability to be recognized in successive images (i.e. it is relatively constant). This assumption precludes tracking objects which are textured or otherwise have a mix of intensities. This new tracker is called a *corner-tracker* because it keys on corner feature projections. Corners are good features because they recur, have strong gradients in different directions which supports x,y location in the image, are scale invariant, and can be quickly tracked. In addition, corners respond naturally to rotations about the optical axis. The corner feature is parameterized by six image-plane parameters: x,y is the location of the corner point, and [dx1,dy1] and [dx2,dy2] are vectors defining the two lines of the corner from the center point.

The corner-tracking algorithm first finds edge points, combines points into lines, and then combines two lines into the corner. Each line needs three or more edge finders -- currently five are used. The edge finders are 1-dimensional windows used to find the edge by matching the mode of the template foreground to the window at edge regions. This edge finding thus is dependent on both the presence of an edge and loosely on the intensity level of the region defined by the edge. A derivative-of-gaussian filter of width 5 is used to compute edges along the edgefinder window and these are filtered by a threshold to reject weak edges. The edge finding method relies on both the presence of an edge as well as agreement of the intensity value.

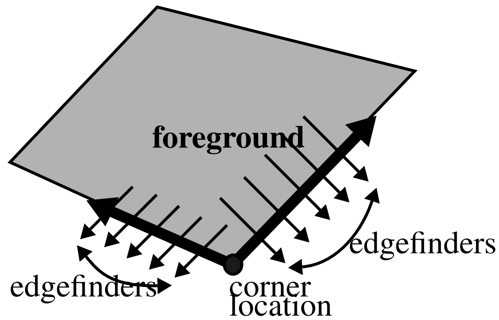


Figure 3-8: Corner Tracking

$$p'(x) = \frac{-x}{P}\frac{1}{\sqrt{2\pi P}}e^{-\frac{x^2}{2P}} \tag{3-3}$$

$$P = \left(\frac{N-1}{6}\right)^2 \tag{3-4}$$

To digitize the filter, an odd integer, N, is chosen as the filter size. The filter template is computed for steps i=0 to N-1 by computing p'(x), with x = i + (1-N)/2, and normalizing the filter coefficients so that the sum of the filter magnitudes is 1.

After the edges in a window are found using the above filter and thresholding, one must be selected which divides the foreground/background. Figure 3-9 shows a figure of four edge pixels (two are adjacent) along an edgefinder. Each is a candidate as the foreground/ background divider. The mode (most frequently-occurring pixel value) is computed of the region associated with each edge. An edge region is defined as the a connected set of pixels beginning at either the first pixel in the edgefinder window or the first pixel after the last edge and ending with the edge pixel itself. Starting at the beginning of the window, the mode for each edge point (region) is computed. All the pixels up to and including the edge point are included in its region. The most frequently occurring pixel value, or mode, is used

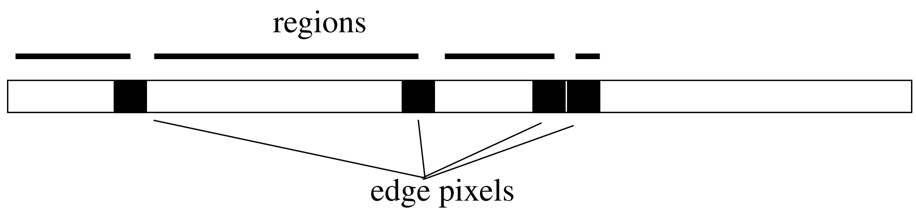


Figure 3-9: Edge Region Pixels

to represent the pixel intensity value in each region. Each pixel has a range of 0-255, or 8-bits, and we reduce this resolution to 5 bits to compute the mode by right bit-shifting by 3 bits. Thus, the region mode values can range from 0 to 31. Once the mode values for each edge region are computed, the edge is selected which gives the smallest error between the region mode and the mode in the last image corresponding to the foreground/background edge region. For two edges with the same error value, the edge is selected which is closest to the end of the edgefinder. Edges with region mode errors greater than 2 are not selected.

Once an edge is found, its confidence (0-1) is computed based on the consistency between the edge direction and the line direction. For example, an edge for a horizontal line, should be vertical. A missing edge point has a confidence value of zero.

$$\text{point confidence} = \frac{1}{2}(1 - \cos(\alpha - \beta)) \qquad (3\text{-}5)$$

where α is the edge angle and β is the line angle. The edge angle is computed as the atan(dy,dx) where dy and dx are the gradients in the y and x directions, respectively. The resulting confidence value is maximal (1) when the difference between the corner angle and line angle is +/- 90 degrees, and minimal (0) when the angle difference is zero or 180 degrees.

To track a line, multiple edgefinder windows are placed along the line -- five are currently used which provides redundant information for line fitting. Edgefinder windows have an orientation either vertical, horizontal, or diagonal (1x1 step) -- the particular orientation is selected to be nearest the line normal. Edgefinders are placed equally along the line and find an edge point in each. These edge points are then fit using weighted least squares (confidence=weight) to compute the line m,b parameters. Two different parameterizations of the line: y=mx+b or x=my+b are used to preserve robustness in the least squares computation. Parameterizations are switched at the m=1 slope. When computing new line parameters, the parameterization is based on the bounding box (dx>dy => y=mx+b) -- the goal is to keep the slope -1<= m <=1.

To provide robustness against spurious noise, a weighted update is performed of the line parameters according to a confidence value of the line. The line confidence is computed based on the component point confidences (c_i) and has a value in the range of 0 to 1.

$$\text{line confidence} = \left(\frac{1}{N}\sum_{i=1}^{N} c_i\right)^2 \tag{3-6}$$

When the equations for each line have been found using the above techniques, they are intersected to find the corner location. Constant lengths along each line are used to place the edgefinders. The assumption is that the image lines defining the corner will always be at least as long as this fixed length. Of course, this assumption may not always hold and motions (e.g. rotations) may render a corner untrackable by shortening the edge projections on the image plane. This restriction has not been too difficult in practice. Rotations about an axis parallel to the image plane will modify the projection length of edges on the image and so must be done very carefully. The projection length is insensitive of pure translations and rotations about the optical axis. More sophisticated methods of (adaptively) setting this length could be model-driven. Finally, note that corners have distinct foreground and background components. The type of corner must be identified so that the edgefinder windows can be properly directed with their beginning in the foreground pixels. The corner designation as acute or obtuse cannot be changed during tracking.

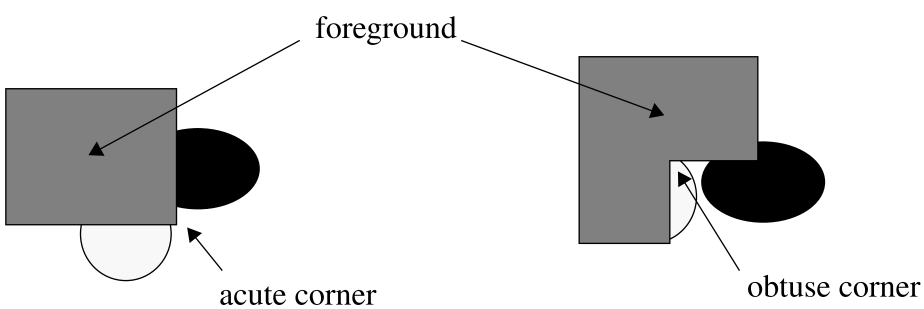


Figure 3-10: Acute and Obtuse Corners

This corner tracker was implemented using a Datacube Maxvideo20 image-processing system. Using a full-resolution image which takes ~17ms to transfer, two corners can be tracked at ~38Hz, or faster than frame-rate using a max860 RISC processor in the Datacube system. Since image transfer time is such a large part of the image processing time and so few pixels (~250 per corner) are actually used by the algorithm, more efficient image transfer would make possible tracking more corners to support stereo tracking and/or the use of more sophisticated tracking algorithms.

3.4.2 Image Plane Errors

The point of tracking features is to compute image-plane errors and generate a robot velocity vector based on these errors. Generally two corners are tracked in the image which belong to different parts in the workspace. Several errors may be computed from this pair. First, there are the x,y translation errors between the corner origin (3-8a). An angular orientation error can be defined between selected lines on each corner. Finally, the difference in magnitudes of two corner angles might also be used as a error signal.

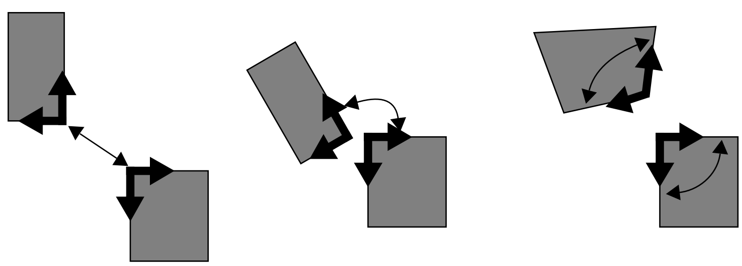


Figure 3-11: Corner Errors

One of the part motions is influenced by robot commands (i.e. the robot grasps one part). The other part is often fixtured, but may be in unpredictable motion (i.e. on an uncalibrated conveyer). Now that feature error measures have been defined based on the trackable features, the relationship between the robot motion and the feature motion must be derived.

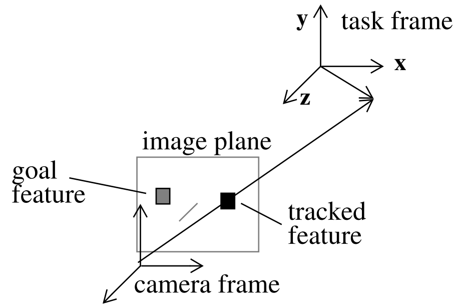


Figure 3-12: Fixed Camera Visual Servoing

3.4.3 Image Jacobian

Tracking the features of interest is an important first step for visual servoing, but the ultimate goal is to control the robot (and hence the task) with the information. Understanding the image jacobian, which describes how the robot motion affects the image plane feature motions, is a critical step in deriving these control laws. The first step is to understand how motion of a (task) point affects its projection on the image plane. A pin-hole camera model, is used to compute the projection of a task point onto the image plane. Differentiating this equation leads to the familiar optical-flow equation which relates task point velocities to image plane velocities. From Nelson et al [49], the result is:

$$\begin{bmatrix} \dot{x}_s \\ \dot{y}_s \end{bmatrix} = \begin{bmatrix} \frac{f}{s_x Z_C} & 0 & -\frac{x_s}{Z_C} & -\frac{Y_T x_s}{Z_C} & \left[\frac{f Z_T}{s_x Z_C} + \frac{X_T x_s}{Z_C}\right] & -\frac{f Y_T}{s_x Z_C} \\ 0 & \frac{f}{s_y Z_C} & -\frac{y_s}{Z_C} & -\left[\frac{f Z_T}{s_y Z_C} + \frac{Y_T y_s}{Z_C}\right] & -\frac{X_T y_s}{Z_C} & \frac{f X_T}{s_y Z_C} \end{bmatrix} \begin{bmatrix} \dot{x}_T \\ \dot{y}_T \\ \dot{z}_T \\ \dot{\omega}_{X_T} \\ \dot{\omega}_{Y_T} \\ \dot{\omega}_{Z_T} \end{bmatrix} \tag{3-7}$$

where the image jacobian relates a point feature velocity on the image plane to the task frame velocities for the task frame aligned with the camera frame.

3.4.4 Control

In this thesis simple proportional control laws are used based on the errors between a moving (and controllable) feature and a frozen (uncontrollable) feature imaged with a single camera configuration. These simple control laws are used because they are easy to implement and because at frame-rate feature tracking speeds, they exhibit sufficient robustness to calibration errors of the imprecisely modelled camera/lens/manipulator system. The difference between the features forms the projected task error on the visual sensor space. Image-based control laws rather than pose-based control laws are used to reduce sensitivity to camera/manipulation calibration and sensor modelling errors.

Resolvability analysis developed by Nelson [46] drives simple control law decomposition. For a single camera tracking a single corner, the x,y motion on the image plane and rotation about the optical axis are the most resolvable errors (i.e. generate the largest sensor signals). To control additional DOF one should add additional cameras with good resolvability placement which normally means orthogonal to the first camera. Alternatively, one could track additional features on the same object to control more DOF from a single camera view.

$$ {}^{A}v = {}^{A}R_{C}K\begin{bmatrix} e_x \\ e_y \\ 0 \end{bmatrix} \qquad {}^{A}\omega = {}^{A}R_{C}\begin{bmatrix} 0 \\ 0 \\ K_\theta e_\theta \end{bmatrix} \tag{3-8} $$

The image plane velocity is transformed into the task frame by a task/camera transformation which only needs to be approximately correct for stable control. For servoing at roughly constant depth (camera to task), the gains can be made constant to achieve acceptable control. For a depth of ~20cm, the linear gain is K=0.002 m/s/pixel and the rotary gain is $K_\theta = 0.5$.

The effect of camera/task transformation errors is warping of the image-plane trajectories. For example if the true task frame is slightly rotated about an image axis from the assumed one (Figure 3-13), then the task velocity will have a depth component and the projected component on the image plane will be shorter. Small depth velocity components are not a problem but point out the need to have a second camera view if the calibration is especially poor and the task requires tight control in the depth direction. The actual X velocity in the image plane will be Vcos(θ) instead of V. This causes a curved trajectory like that shown in Figure 3-13 rather than the expected straight direction.

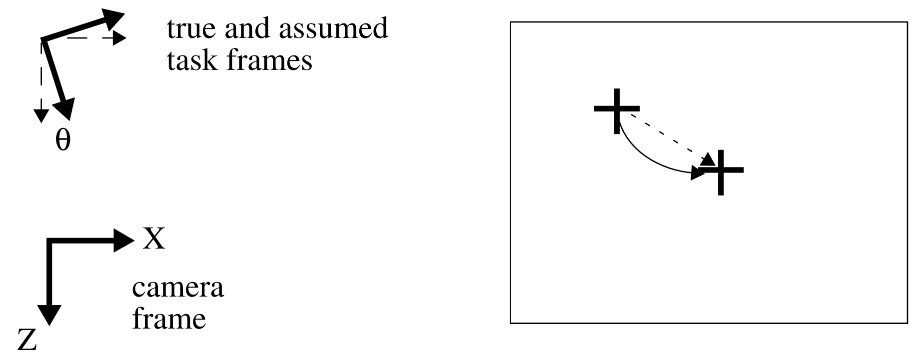


Figure 3-13: Task Frame Error Effects on Visual Servoing

Using two cameras is necessary when the task uncertainty cannot be resolved from one camera view or when the camera/task calibration is especially poor. If the cameras can be placed orthogonally, then the projection onto task frame is independent. However, typically the cameras are not placed exactly orthogonally, so coupling occurs. Using the approximate knowledge about the camera placements relative to the task allows one to minimize these disturbances. For example, one camera can be designated 'primary' and used to control x,y,q on its image plane (projected onto the task). The second camera can be used to control depth relative to the second camera view. This simple decomposition was used to implement control of 3D translation for both approach and grasping control.

3.5 Summary

The basic resources were introduced in this chapter for building sensorimotor primitives: resolved-rate cartesian control as the motor resource, and force and vision sensing as the sensor resources. The control strategies using damping force control and visual servoing were described. The force sensor signal processing and feature tracking to extract information from the signals for control purposes were outlined. The next chapter explores the application of these resources to solve specific manipulation task primitives.

Chapter 4

Sensorimotor Primitives

4.1 Introduction

The goal of the previous chapters has been to set up the development of sensorimotor primitives for robotic assembly tasks. As discussed in the introduction, the goal is to build an intermediate layer which integrates sensors for a set of related tasks. In Chapter 2, the taxonomy of relative motion was introduced and in Chapter 3, the resources were introduced. This chapter joins the two into sensorimotor primitives as algorithms for implementing specific interpretations of motion constraints defined by the relative motion taxonomy. As shown in Figure 4-1, a sensorimotor primitive bridges the gap between the robot space and the task space. Unlike an abstract manipulation task primitive class, a sensorimotor primitive is directly executable. Unlike a robot move command, it has task context and meaning because of the interpretations attached to the sensor data.

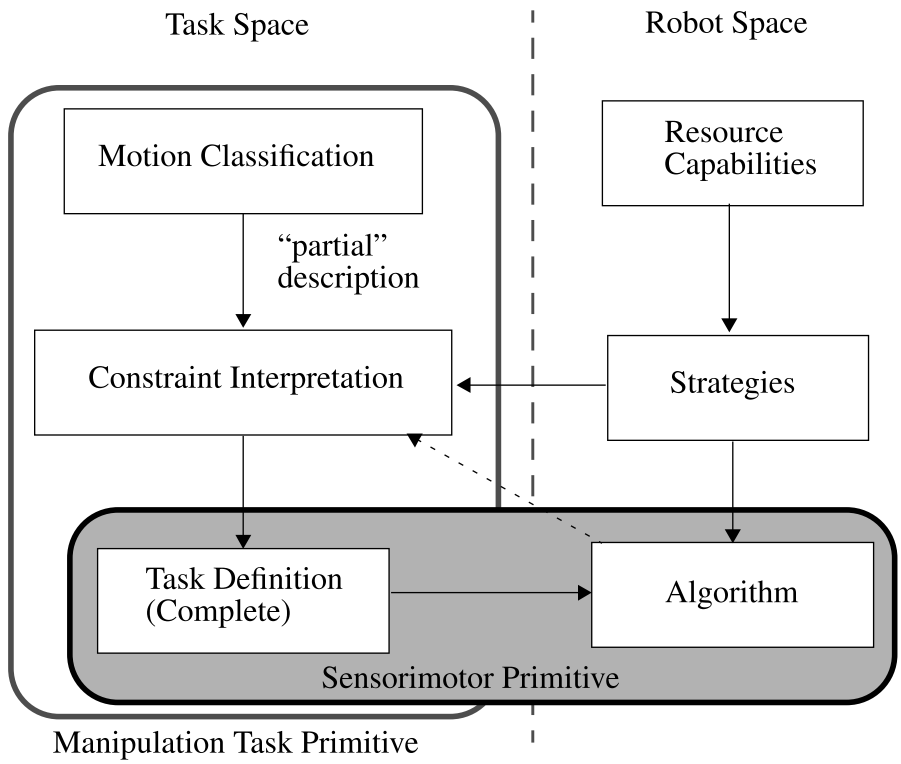


Figure 4-1: Sensorimotor Primitive

A *sensorimotor primitive* (SMP) is defined as the solution implementation of a particular MTP. The term *sensorimotor* indicates the fusing of sensing and action into one command with a task-relevant definition. This thesis focuses on primitives which involve relatively small motions at the gross-fine motion boundary and fine-motion [77]. Because of this, the primitives do not consider 'gross' constraints like avoiding joint singularities or avoiding obstacles. Task-driven primitives are defined via sensor information. The goal is to control the task, and the robot is merely a tool to effect it. An SMP integrates domain knowledge (how to achieve the manipulation primitive) with facility capability into a form which preserves task-context and is directly executable.

This chapter does three things: 1) introduces the structure of sensorimotor primitive implementations of MTP algorithms, 2) defines example MTP's based on the motion classifications and resource capabilities, and 3) implements algorithms to solve the MTP's.

4.2 Sensorimotor Primitive Structure

To execute a specific MTP an algorithm must be conceived and implemented on specific resources (robots, etc.). There are three basic elements in the execution phase: 1) the controller, 2) the trajectory, and 3) event detectors. In addition, there are initialization and finish phases to execution for configuring the controller transition, sensor configuration, and "clean-up" when the primitive terminates. The discrete phases in the primitive and in the larger program (sequencing primitives) illustrate the *event-driven* nature of the control architecture. Central to this approach is event-detection to drive the controller changes in different parts of the task strategy.

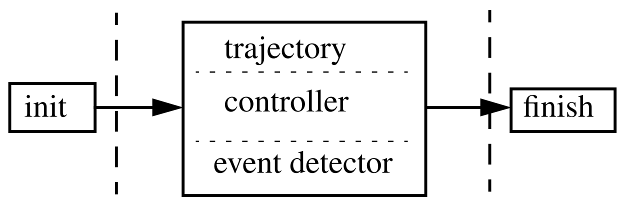


Figure 4-2: SMP Phases

The desired task motion trajectory (or task action) is defined relative to specific task part features -- so-called 'defining' features. Often this is transformed into a specific robot trajectory on the basis of prior knowledge (such as the beginning locations and shapes of each part). Precompilation of task goals into robot trajectories undermines the ability to compensate for task uncertainty. The goal here is to actually define the task trajectory at run-time through direct measurement of these 'defining' features. This can be achieved through visual servoing by defining an error function from feature locations on the image plane. Of course, such a primitive still requires preconditions to be met: for example, these defining features must be visible to the sensor. Prior assumptions on shape and pose are used to generate open-loop trajectories to satisfy these preconditions. Force setpoints are usually set based on initial information -- not on the force measurements. For example, insertions require zero force/torque setpoints along the constrained directions.

Manipulation tasks executed by robots are fundamentally multi-step and *event-detectors* are needed to drive the strategy changes. The purpose of the event detectors is to capture the algorithms for detecting specific task events through sensor data processing based on the task model and control strategy chosen. Events can be 'internal' to the strategy signifying the change in strategy (e.g. transition from position to force control) or they may be exiting events indicating the strategy is finished (succeeded or failed). At a minimum, the goal state must be reliably detected to indicate when the primitive has successfully completed. Often the goal is considered attained when the trajectory finishes. However, when deriving the trajectory from task measurements, the termination condition must also be determined by task measurements. Event detectors typically involve computing a scalar value and comparing it to a threshold value to generate an event.

The dual variables of the controlled axes are typically monitored for event detection. For example, in guarded moves the velocity is controlled and the force is monitored. Implicit in this is the expectation that the force will increase at the appropriate time (e.g. contact). In general event detection requires predicting the event projection onto the sensor signal. Predicting this projection is currently more of an art than a science. Complex contact state effects on force signals are notoriously difficult to predict algorithmically. McCarragher and Asada [39] have used qualitative interpretation of force signals along with a dynamic task process model to recognize discrete state changes in a robotic assembly task. This thesis focuses on simpler event detectors which can be re-applied to different tasks. Using all the information about the task model and the control strategy can help in designing event detectors. A normalized correlation measure between the reference and force-perturbed velocity signals detects the presence of a planar motion constraint [43].

In this section the basic structure and components defining a sensorimotor primitive were introduced. The rest of this chapter is organized around the different MTP classifications of relative motion. Geometric interpretation(s) which actually define the constraints are used to derive or implement algorithms to execute the primitive motion. Both contact and non-contact interpretations of the constraints will usually be provided. Contact constraints are imposed by the task mechanics 'naturally' -- the control challenge is to obey these constraints through appropriate force feedback. Non-contact constraints are typically both defined and realized through visual-servoing control. In one case, a mixture of contacts (force) and vision is used to define the constraint.

4.3 Visual Constraints

In the case of contact, the mechanics of the task provide the constraints -- the primitive's job is to comply and observe those constraints while keeping forces inside acceptable bounds. However, algorithms for many tasks with contact constraints have fairly small entrance funnels. Consider a peg in hole task -- once the peg is in the hole, four DOF are constrained -- but this state can only be entered from a fairly precise non-contact state (or at least a state with fewer DOF constrained). Attaining this 'precise' non-contact state can be specified and controlled through visual feedback. Implementing MTP primitives with visual feedback requires defining the motion constraints on the image plane.

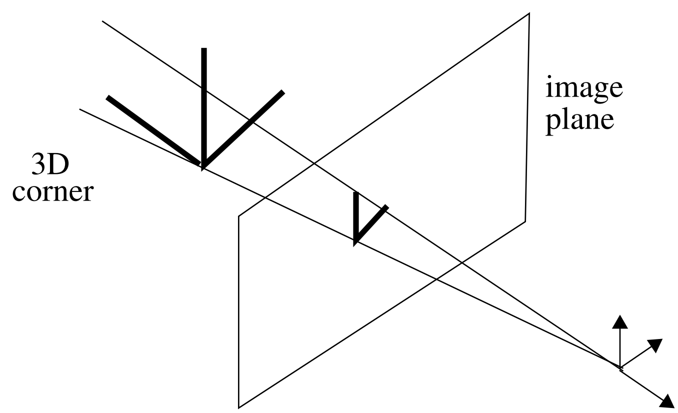


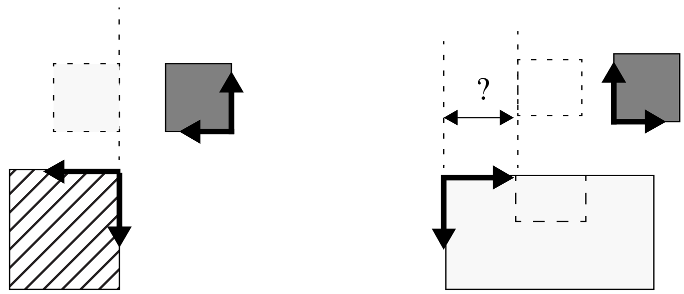
Figure 4-3: Pin-Hole Camera Model of Corner-Tracking

With a single camera configuration, tracking one corner feature in a single image can provide three non-contact constraints: x,y translation in the image plane (the corner x,y location) and the orientation of the corner in the image (rotation about the optical axis). If one specifies both the orientation and the corner angle magnitude, then all three rotations can be fixed. With a single corner, one cannot fix the third translation, and, interestingly, one cannot fix only two rotation DOF. A two camera configuration with each tracking one corner can specify the cases of three translation DOF and two rotational DOF which were not possible with a single camera configuration. The motion constraints which can be specified with one and two camera visual servoing configurations in which one corner pair in an image generates an error is summarized in Table 4-1. The square indicates two free DOF, the line one free DOF, and the point zero free DOF. Thus, a single camera configuration can constrain 1 or 2 translation DOF, but not all three.

Table 4-1: Visual Constraint Specification

# cameras	**DOF**					
	□ 1		↑ 2		• 3	
	T	**R**	**T**	**R**	**T**	**R**
1	x	x	x			x
2	x	x	x	x	x	x

Direct features are visible to the sensor. In the case of bringing the corners of two blocks together, both corners are directly visible (and trackable) on the image (Figure 4-4a). However, this is not always the case. Consider an initial positioning goal for a peg-in-hole task (Figure 4-4b) -- the feature of interest (the hole) may not be visible to the sensor. In this case, a secondary feature must be tracked from which the location of the feature of interest can be inferred in the image. In the example given, the distance between the outside corner and the hole must be known. Using indirect features increases the sensitivity to uncertainty in task knowledge (i.e. the distance between the defining (direct) feature and the observable (indirect) feature). Also, when the goal position is actually offset from a feature this increases our sensitivity to camera placement uncertainty since the projected distance will change with viewing direction. So the offset may be computed based on one viewing direction but the execution performed with a different viewing direction.



a) Direct Feature b) Indirect Feature

Figure 4-4: Direct and Indirect Features

Figure 4-5 shows the camera pose errors between the "design" camera pose and the actual camera pose. The camera pose error is parameterized by (X_e,Z_e, and θ_e). The question is what is the effect of this camera pose error on the image plane error between two points in the task? By transforming the points into the new camera frame, the new projection equations can be written in terms of the original frame parameters.

$$x' = \left(-\frac{f}{s}\right)\frac{X + \theta_e Z - X_e}{Z - \theta_e X - Z_e} \tag{4-1}$$

For zero camera pose error, the result collapses to the original projection point. Also note that for zero rotational and depth (Z) errors, the difference between the image plane projections of two points is unaffected by errors in X. The rotational error cannot be too

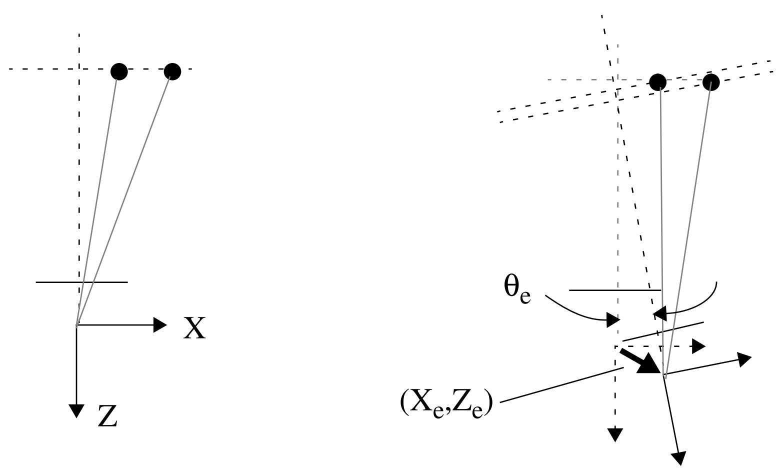


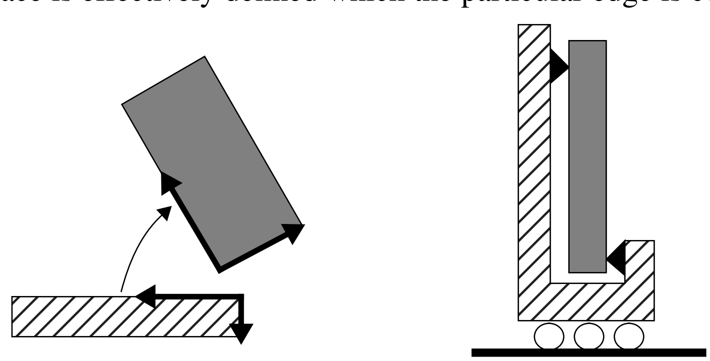
Figure 4-5: Camera Pose Errors

large or the projection quality will suffer. Also, the rotation and X error cannot be “too independent” or the features will not appear on the image plane. The image plane error between two projected features is most affected when the change in depth is significant compared with the depth value. A focal length significant relative to the depth also increases the sensitivity to depth errors. In practice, visual servoing was quite robust to camera/task pose errors. The main sensitivity to camera pose errors is with the feature acquisition algorithms.The very simple one implemented in this thesis requires good camera alignment.

4.4 Constraining One DOF

4.4.1 aab

This primitive involves specifying one rotation degree of freedom. Since constraining a rotation by contact usually means at least a half-constraint on translation, it is difficult to conceive of a contact situation which constrains only one rotational DOF. Contact situations would more naturally constrain at least a translation and a rotation DOF. A non-contact interpretation of this class is edge parallel to surface. This constraint can be implemented using corner-based visual servoing (CBVS) by defining it on the image plane. Our implementation is to specify the orientation of a projected corner -- the control output is rotation about the optical axis. Either one of the lines forming the corner can be used as an orientation indicator or the corner bisector can be used. By specifying a particular angle on the image plane, a surface is effectively defined which the particular edge is constrained to par-



a) non-contact (image plane) b) contact

Figure 4-6: aab interpretations

allel. All are affected by rotations about the x,y axes as well as the optical z axis. The frame in which the constraint is defined and controlled is the camera frame.

4.4.2 aac

More common than aab, the aac class involves constraining a single translation DOF. There are both contact and non-contact interpretations of this constraint. For contact, there are two interpretations involving 'primitive' contacts: 1) vertex/plane and 2) edge/edge. Either type of contact removes one translation DOF. To realize either of these contact interpretations requires the use of force feedback -- for example, damping control which accepts velocity and force setpoints in the task frame. The task frames are defined according to the assumed task geometry -- only the axis defining the direction of constraint is actually important to define. The point/plane interpretation requires a task frame axis to be normal to the plane surface. The velocity in this direction is zero and the force setpoint is the desired contact force. The edge/edge task constraint axis is defined by the cross product of the two edges -- this varies dynamically if the parts rotate relative to each other. Using a hybrid control approach, force is controlled in the constraint direction and velocity along the surface directions -- notably, the force or velocity setpoints are effectively zeroed for velocity and force controlled directions, respectively. In addition, a non-contact interpretation can be defined on the image plane. Specifying the image plane x (or y) coordinate amounts to constraining the corner projection along 1 direction in the image plane. In this case, the task frame is the image plane and the constraint is both defined and controlled on the image plane.

4.5 Constraining Two DOF

4.5.1 abb, acc

These two classes contain the (3,1) pair of DOF. The abb class refers to removal of two rotational DOF while preserving all three translation DOF. A non-contact interpretation of this constraint is the alignment of two surfaces without contact (or the specification of an edge orientation). This primitive can be very useful for aligning insertion axes before mating

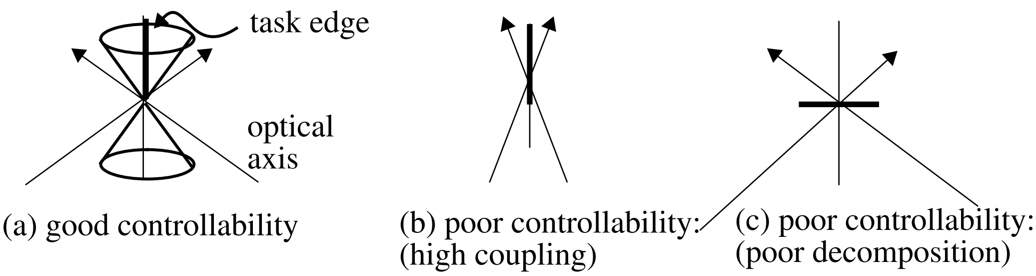


Figure 4-7: Edge Controllability

(assuming they corresponded to the surface normals). It can be implemented by using the orientation rotation information in our corner feature, but two cameras are needed to provide independent information. If a corner is tracked in each image, two different rotation constraints can be specified -- one in each image. Each camera will produce an angular velocity about the optical axis of that camera. Unless the camera optical axes are orthogonal, there will be coupling between the two rotation commands. If their approximate orientation is known (which is required for control) then one angular velocity vector can be projected to be perpendicular to the other. As before, each angular specification defines a surface in 3-space which the corresponding edge must parallel. Defining two such surfaces constrains the free rotation to their intersection direction. There are two cases to consider.

Case 1 is controlling the same edge (or parallel edges) viewed in two images. Ideally, the optical axes should be orthogonal to each other and to the edge. If the optical axes nearly aligned, then the resulting commands are highly coupled (Figure 4-7b). Minimum tracking length constrains the edge orientation to lie inside a cone centered at the cross product of the cameras' optical axes (Figure 4-7a). A very short task edge (in 3D) will make the cone very steep and narrow and limit the applicability and robustness of the primitive. Alternatively, if the edge lies in the plane formed by the two optical axes (Figure 4-7c), independent control from one camera is effectively lost. Both cameras cannot independently control their edge projections, so this configuration is ill-conditioned. The challenge with this primitive is ensuring that the edge feature remains visible in each image. If the 3D corner angle is very sharp (small), then this is easier than if the corner angle is shallow (large).

Case 2 involves controlling two perpendicular edges on the same part via different images, but the capability is much more restricted. Consider two optical axes which are perpendicular and a task feature which is a 90 degree corner. Suppose two adjacent, orthogonal edges are imaged in the different cameras -- only if their cross-product direction is aligned with the cross-product of the optical axes is a rotation freedom preserved (only 2 rotations specified). A rotation about Z, the optical axes perpendicular, will not affect the edge projections onto the images. Non-perpendicular optical axes will not affect this constraint. Any corner orientation which is "off the Z axis" fixes all rotation DOF because there is no free rotation which will not modify the edge projections onto the images. The goal is to place the cameras to that each image can produce a control signal which will not disturb the other.

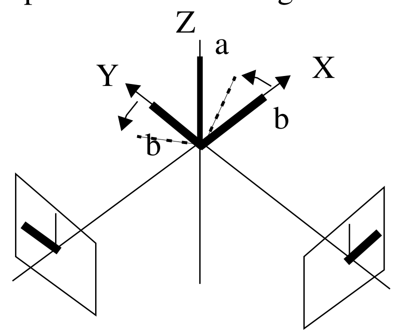


Figure 4-8: Corner Alignment

The *acc* class specifies two translations while leaving all rotations free. It is quite simple to define this via the TVS primitive with setpoints for both the x and y coordinates of the corner. The part is free to translate along the optical axis of the camera. Rotations are free but are limited by preserving the visibility of the feature in the images -- i.e. one cannot rotate the corner so that either edge lines up near the optical axis.

4.5.2 abc

The abc class is a (2,2) pair and enforces one translation constraint and one rotation constraint along different axes. The contact interpretation is edge against (flat) surface. Hybrid

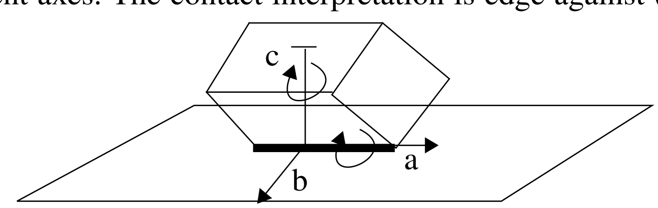


Figure 4-9: Edge/Surface Contact

control is partitioned according to the contact directions. The true task frame is defined by features on both parts: the surface normal of the fixed part and the edge direction of the controlled part. If a measurement of the surface normal was available, the frame could be dynamically updated based on the edge orientation and the surface normal. Since a surface normal measurement is not available, a hand-fixed frame is used for task control, where one axis is specified to align with the surface normal during contact. Recall that significant orientation changes while in contact are disallowed to minimize gravity-induced force control errors. Alternatively, a non-contact interpretation of the abc class can be implemented with visual servoing primitives. Using a single-camera configuration, the rotation about the optical axis can be constrained along with either x or y translation in the image plane.

4.5.3 aad

This is the other (2,2) pair with the loss of rotation and translation DOF along the same axis, but fully free motion along both other axes. This is an example of a relative motion constraint which is very difficult to interpret geometrically. The difficulty centers around constraining a rotation in one direction *without introducing a translation constraint in another direction.* It is very difficult to conceive of a device which will persistently impose this constraint set. The strange mechanism shown in Figure 4-10 will *momentarily* impose

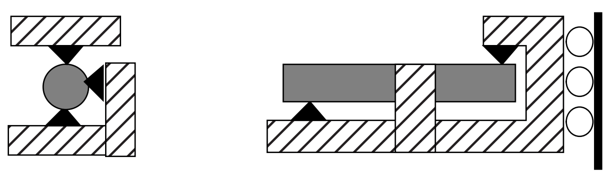


Figure 4-10: aad mechanism

such a constraint.

Again, this constraint set can be interpreted through visual servoing by considering a two-camera arrangement. Two cameras are required since a single camera provides translation measurements in the image plane and rotation information about the optical axis which, by definition, is perpendicular to the image plane. Since translation and rotation constraints are required along the same direction, the optical axes of two cameras must be perpendicular. Assume that camera A will control the translation and camera B the rotation about the optical axis. The projection direction of camera B's optical axis onto camera A is needed to reduce the coupling. Because the constraints apply along the same axis, the camera calibration must be known fairly accurately. Assume that the cameras are aligned so that the projection of B's optical axis is along the x-direction of A's image plane. Then in image B the orientation of the corner is controlled while in image A its x-position is controlled.

4.6 Constraining Three DOF

4.6.1 bbc, bcc

These are the (1,2) pairs for specifying three DOF. A surface against surface is one contact interpretation of bbc. The hybrid control specification is straight-forward to comply to this constraint set (see, for example, Mason [37]). Assuming Z is the surface normal direction, force is commanded in Z, and torques about X, Y are zero. Class bbc can also be interpreted as a non-contact constraint set implemented with visual feedback. In this case, the abb primitive discussed earlier can be combined with a single translation constraint in one of the images.

One bcc class contact interpretation is peg-in-slot. The true task frame is defined by a combination of task features on both parts: the axis of symmetry of the controlled part and the surface normal of the slot wall. The control task frame is again chosen to be hand-fixed with one axis coinciding with the symmetry axis and the other corresponding to the expected surface normal direction. Again, the hybrid control specification is straight-forward. Assuming the peg axis is Z and the slot wall normal is X, command force in X and Z, and zero torque about Y. Errors between the assumed slot wall surface normal and actual

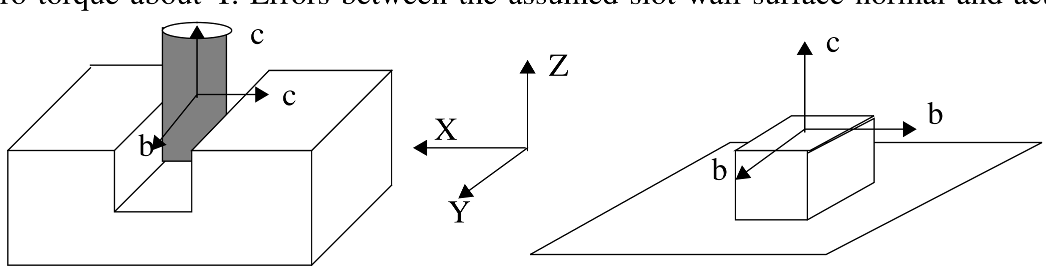


Figure 4-11: bcc and bbc contact interpretations

normal result in disturbances between the force controlled direction into the slot wall and the velocity commands along the slot. Using a hand-fixed frame limits our rotation about the controlled part's axis of symmetry (since this would introduce errors between the task 'control' frame and the true task frame).

4.6.2 bbb, ccc

These two constraint classes correspond to (0,3) pairs. The bbb class corresponds to elimination of all rotation DOF while preserving all translation DOF. One contact interpretation of this constraint class is a strange 'translation' mechanism made up of non-round sleeves and sliders which is rarely encountered in everyday life. The task frame is defined by

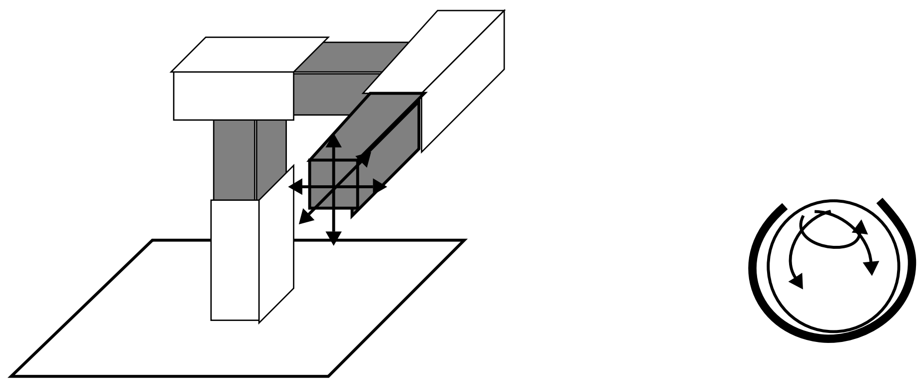


Figure 4-12: bbb and ccc contact mechanisms

the mechanism topology, although because of the symmetry it is arbitrary. The hybrid control is straight-forward -- position-control along all translation DOF, and torque control over all rotations. The bbb class can also be interpreted through visual servoing. Consider Case 2 of the abb non-contact interpretation. If the special configuration defining abb is avoided and the angles of two task edge projections are specified on different image planes, then the orientation of the controlled part is fully constrained. This is because there remains no possible rotation which will not modify at least one edge projection onto an image plane.

The ccc class corresponds to the elimination of all translation DOF while preserving all rotations. A very common contact interpretation of this constraint is a ball-in-socket joint. The task frame is naturally defined in the center of the ball -- its orientation is arbitrary. Hybrid control allows position/velocity control of all rotations with force control along all

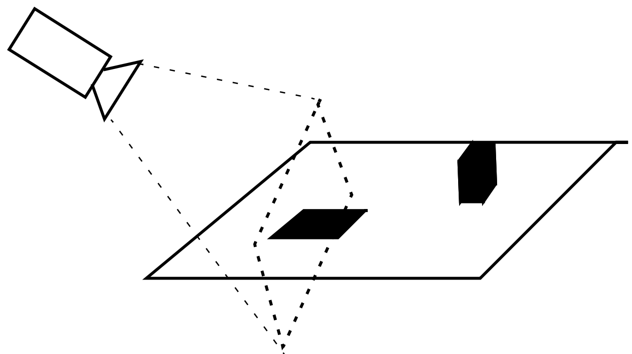


Figure 4-13: Surface Visual Servoing

translations. Again, a non-contact definition of the ccc constraint exists using visual servoing translation setpoints with a two camera configuration. A single camera can constrain two translation DOF of a corner feature -- using a second camera allows one more translation constraint to be specified. If the cameras are not orthogonal in their view, some coupling will occur between the constraints specified and control disturbances will result.

A much more common interpretation exists which combines force and vision for defining the three translation constraints. Consider a part moving along a flat surface. In this case, to maintain contact with the surface sets 1T DOF and the location on the surface constrains two more T DOF. So in this case force along the surface normal to fixes one DOF and visual feedback to fixes the other two DOF along the surface. One problem is that the visual servoing control formulation must be extended to prevent large disturbances to the force controller from the vision controller because of the required camera placement. Figure 4-13 shows the projection of the image plane intersects with the real surface. Thus, errors in the Y image direction (the vertical image plane coordinate) will map into velocities into/out of the surface which will tend to drive the part into or out of the surface. The velocity projection should be roughly along the real surface instead of the projected image plane.

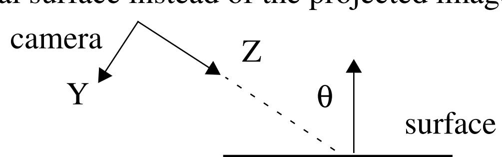


Figure 4-14: Camera/Surface Orientation

If we assume that the camera X axis is parallel to the surface (i.e. that we rotate about X to ‘see’ the surface), then only the Y and Z directions must be coordinated to realize a velocity parallel to the surface (Figure 4-14).

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = {}^{A}R_C \begin{bmatrix} K_x e_x \\ K_y e_y \\ K_y e_y \tan\theta \end{bmatrix} \qquad (4\text{-}2)$$

Besides coupling Y and Z, a new gain function is needed because the assumption of constant depth which allowed the use of constant gains is now violated. The gain factor which depends on the Y position is used to modify the previously constant gains.

$$K(y, \theta) = \frac{K_0}{\left(1 + \frac{sy}{f}\tan\theta\right)^2} \qquad (4\text{-}3)$$

where y is the pixel coordinate, s is the scale factor (m/pixel), f is the focal length, K_0 is the gain for the depth along the optical axis, and θ is angle between the optical axis and surface normal as shown in Figure 4-14. The zero Y coordinate corresponds to the center of the image, while the bottom of the image corresponds to +240, and the top to -240. Intuitively, the gains are higher for larger depths and lower for smaller depths. The goal here is to keep the image plane response approximately constant by scaling the velocities up/down depending on the depth estimate.

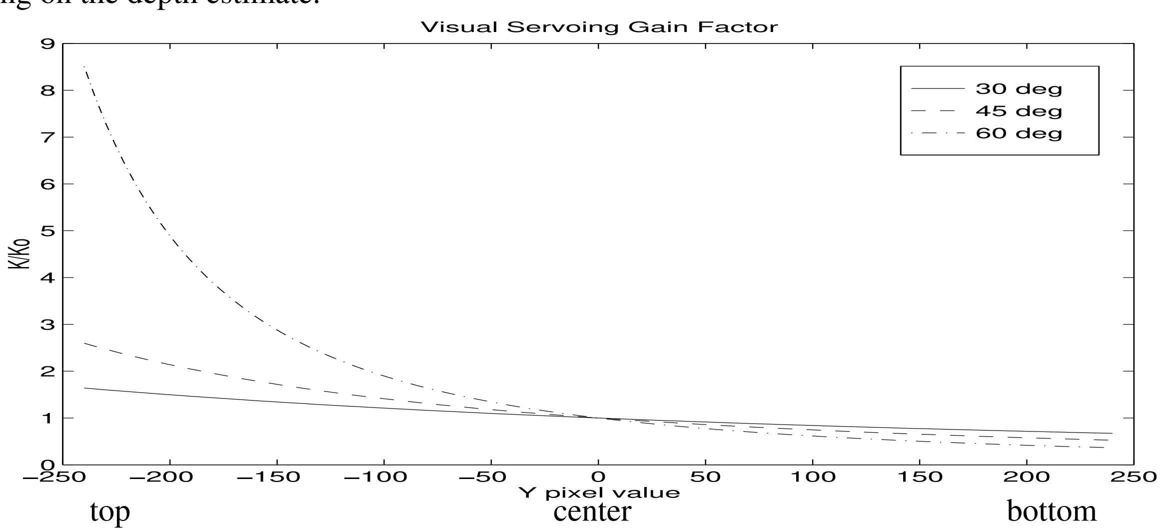


Figure 4-15: Visual Servoing Gain Factor

4.6.3 abd, acd

These classes are (1,2) pairs -- like the aad class, contact interpretations of these two classes are very difficult to generate. The acd class has two translation constraints and one rotation constraint along the free translation axis. This maps perfectly into a single camera visual servoing configuration where the x,y location of the point is specified and the corner orientation in the image is specified. The abd class has a single translation constraint along with two rotation constraints. Although the single translation DOF does not require two cameras, the general specification of only 2 rotation DOF does.

4.7 Constraining Four DOF

4.7.1 add

The add class has a very common contact interpretation: round peg-in-hole. Whitney [78] showed the appropriate task frame location is at the tip of the peg leading motion (or beyond it) -- this location positions the center of compliance to provide proper rotation in response to insertion forces and torques. The hybrid control strategy is position/velocity control along the ‘a’ axis and force/torque control along the ‘d’ axes. The task frame is naturally defined by the peg symmetry axis and is attached to the peg.

A non-contact interpretation of this constraint also exists since a two orthogonal camera arrangement can implement rotation and translation constraints along two orthogonal directions. In theory, a visual implementation exists but in practice the reliance on indirect features makes the result unrobust to task variations.First, the cameras must be orthogonal and preferably aligned with the surface of the hole. The orientation aspect requires only that the occluding edge of the peg be trackable in each image and an edge on the fixed part *which is parallel to the insertion axis*. The orientation part will work even if the cameras are not parallel to the surface, but they must be near to preserve robust tracking and rotation controllability. Next, to align the x,y position of the peg, the specification will include an image plane distance describing the location of the defining feature relative to the visible feature -- here the alignment of the cameras with the surface is important otherwise one does not know which direction of the image plane corresponds the defining feature lies relative to the visible feature. Also this distance changes with depth and with orientation of the camera relative to the task (unless the task has occluding symmetry consistent with the hole -- e.g. a hole drilled in the center of a cylinder). If the visible feature is near the defining feature, the errors can be kept small -- but the further the visible feature is from the defining feature, the more sensitive to two types of errors. First, it is more likely that the two features will not both be contained in the image and second, larger errors will be introduced by camera/task orientation errors. The root of the problems incurred when trying to servo the peg x,y position is that the defining feature (the hole) does not project onto the images as corners, but as ellipses. Only if the cameras are carefully aligned will the projection be an ‘invisible corner’ -- this is why a visible ‘indirect’ feature was needed.

This points out the need for additional feature trackers -- e.g. for ellipses. Extending the basic image processing to create an ellipse tracker by using the 1D edgefinder windows should not be difficult. Indeed, more general snake-trackers have been created to track non-rigid bodies [7]. An ellipse feature would have the x,y location, the lengths of the principal axes, and the orientation of the ellipse in the image. An ellipse feature could be used to define in each image both the rotation and translation constraints to bring the peg and hole together in the proper alignment for mating. The peg axis alignment is driven by the ellipse minor axis direction in the image which would normally be roughly aligned with the Y image direction. The edge point of the ellipse is the target position for the peg edge point

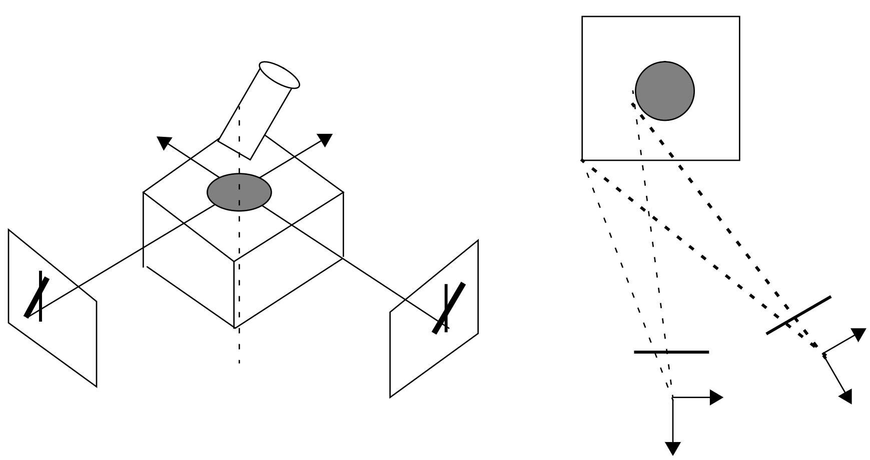


Figure 4-16: Visual Implementation of add

along the major axis direction (Figure 4-17). One problem is the major/minor distinction of the ellipse axes disappears when the ellipse becomes a circle under a particular viewing configuration. The orientation of the ellipse will then be undefined because no major axis is uniquely defined.

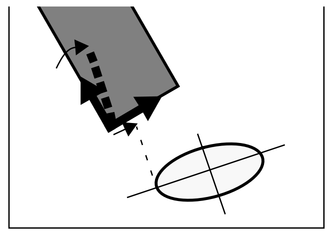


Figure 4-17: Ellipse/Corner Interaction

4.7.2 bbd, ccd

These constraint classes are the (0,2) pairs. In the case of bbd, a similar 'translation' mechanism proposed for bbb conceived an be with one fewer translation DOF as a contact interpretation. The hybrid control specification and task frame definition are straight-forward. A non-contact interpretation can be guided by corner visual feedback. The abb or bbb approach can be used to enforce orientation constraints while the addition of a translation constraint is simply a matter of introducing one x (or y) setpoint in one image.

The ccd class has no translation freedom but two rotation DOF. A 'rotation' mechanism can be designed to implement such a constraint set (e.g. a universal joint). A similar mechanism has two yokes: one pivots relative to ground, the second rotates orthogonal relative to the first. The task frame is defined by the two axes which are free to rotate -- the free rotation axes always remain perpendicular. The task frame moves with the final (cylindrical) link. The hybrid control is straight-forward.

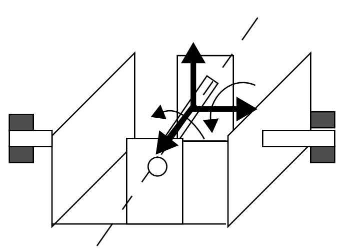


Figure 4-18: ccd rotation mechanism

The ccd class also has a non-contact implementation which can be realized through visual feedback. A two (orthogonal) camera configuration is necessary so that 3 translation constraints can be specified through x,y image plane setpoints. Then, in either camera, a rotation setpoint can be provided to constrain one rotation DOF.

4.7.3 bcd

The bcd class corresponds to the (1,1) pair where a single DOF of each type remains along different axes. One contact interpretation of this constraint is a peg-in-slot variation.

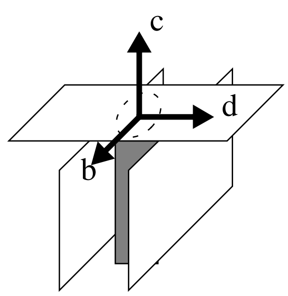


Figure 4-19: bcd contact interpretation

Given the task frame, the specification of hybrid control gains directly follows.

The non-contact version of bcd through visual servoing encounters the same problem as trying to specify only two rotation DOF using corner-features. Specifying two translation constraints is easy with either one camera or two. Specifying only two rotation DOF and leaving one free, however, requires having to view and control the same task edge (or two parallel edges) in both cameras. The free rotation corresponds to the rotation about the edge in 3-space. The extent of allowable rotation about this edge is constrained by the edge-shortening effects in the images.

4.8 Specifying Five DOF

4.8.1 bdd

The bdd primitive is one of two (0,1) pair primitives -- it allows one free translation and has constrained all 3 rotations. A very common contact interpretation is non-round peg in hole. The task frame can be defined by either part and we will use the controlled part -- following Whitney [78], the task frame should be located at or beyond the tip of the peg which leads motion into the hole. Hybrid control is straight-forward to specify -- force control about all axes DOF except the insertion translation direction.

Again, a simple non-contact implementation can be implemented under visual servoing. Using the bbb camera configuration to constrain rotations, two translation setpoints can be specified on the image planes (can be either on a single image or one per image).

4.8.2 cdd

Finally, the cdd primitive has all translation constrained and only one rotation is free. A common contact interpretation of this constraint class is a crank or hinge. Ideally, the task frame would be located on the rotation axis -- hybrid control specifies zero velocities and force/torques in all force-controlled directions and a non-zero angular velocity about the hinge axis. However, often the task frame will be located in the hand in which case mixed control is preferred over hybrid control. In mixed control, both force and velocity setpoints influence a particular DOF control. This is effectively moving the task frame. If a task frame displacement is known, then the translational velocity can be fed-forward which is defined by the rotational velocity along the hinge axis. Since the estimate of the radius will be slightly incorrect, the zero force setpoint will modulate the actual velocity to comply with the constraint. This is effectively moving the task frame to lie on the hinge axis. Again, the cdd constraint has a non-contact implementation under visual servoing. Combining the abb rotation constraint configuration with 3 translation constraints specified in the two images will yield the five constraints in the set.

4.9 Transition Primitives

The previous sections illustrated how to use the basic force and vision controllers to implement specific interpretations of various MTP classes derived in Chapter 2. Assembly also requires transitioning between different constraint classes. To do this requires using strategies developed above and coupling in specific event-detectors. There are many transition primitives which can be defined within the manipulation primitive taxonomy, and just a few based on assembly-type contact constraints are defined in this thesis.

4.9.1 Guarded Move

The common guarded move is perhaps the only common sensor-integrated command available in robot programming languages today. It provides a robust method to acquiring a contact from free motion. Expressed in terms of MTP class transitions, a guarded move is *aaa* -> {*aac*, *abc*, or *bbc*}, where the specific output class is not specifically controlled. The only consistent output result is that a->c for one DOF, which reduces the free DOF by one translation. The geometric model of the mechanically-stable *aac* contact can be either point/plane or edge/edge.

The SMP algorithm is very simple. A constant velocity is commanded in the direction opposite to the surface normal under cartesian control. The force in this direction is monitored against a threshold to generate the termination event. The initial force reading is noted and used as a bias to be subtracted from subsequent force readings. When the force difference exceeds a pre-set threshold, the 'contact' event is generated and the velocity is set to zero. A maximal move distance is specified -- if the force threshold event is not generated before this distance is completed, the move terminates (velocity is set to zero) and the 'fail' event is generated. The primitive dos not necessarily result in contact, though it often does. (The parts may bounce off each other ending the move with no contact.) Nevertheless, the parts are *effectively* in contact since a free motion cannot be legitimately commanded in the direction of contact after a guarded move since contact is (at least) imminent.

Task frame errors manifest themselves as errors in the velocity direction. Under this algorithm, they have little effect as long as the two parts contact. Errors in the approach direction shrink the allowable relative pose region of the two parts. The commanded velocity can even be specified relative to the hand frame with little impact on the strategy. Success is guaranteed only if the initial poses of the parts along with the task frame assignment can guarantee the intersection of the two parts during the move. This is fairly weak and generally easy to achieve with a gross motion.

4.9.2 fstick

This is really an inverse guarded move -- maintaining a translation constraint and generating an event on its loss. Besides acquiring an a->c constraint, we may also wish to maintain it and detect its loss. The task geometry and frame assumptions are the same as the guarded move. Since we do not know the exact contact state (*aac*,*abc*,or *bbc*), we assume the most restrictive in terms of free motions (*bbc*). Thus, we limit ourselves to commanding free motions in the plane and rotations about the plane normal. The task action is maintenance of the constraint on free motion in the direction of the plane normal. A constant force setpoint is applied under damping control to maintain contact. At the beginning of the move, the cartesian position is saved as a bias position. Changes from this position are noted to indicate a loss of contact; this detection algorithm is only valid if the surface does not move. This primitive assumes a motionless surface and the expectation is that when contact is lost, it is due to the moving part 'falling off' the fixed part. Tiny dP thresholds (less than a few mm) are problematic since they can be easily falsely tripped through motions while in contact. A more sophisticated algorithm for loss detection might look at changes in the force signal as well as motor torques (currents), but noisy signals make extracting information difficult.

Task frame errors are essentially errors in the surface normal. The first problem is that the contact may not stick if the applied force falls outside of the friction cone. The expectation is that the force setpoint will not cause motion along the surface (only normal to the surface) -- this will only occur if the force direction falls inside of the friction cone of contact.

4.9.3 Dithering and Correlation

Sinusoidal dithering with correlation-based event detection is one approach to acquiring and detecting a bilateral motion constraint. Lee and Asada [30] use a similar approach to adjust to the minimum stiffness location during an insertion operation. The defining event is the loss of freedom along the surface motion direction. The advantage is that two dithers can be combined to implement a randomization or search function in the plane useful for finding holes. The basic idea is to compute a normalized correlation of the commanded and force-perturbed (actual) velocity signals over a dither cycle period (4-4). In the absence of a constraint, the contact force is zero and these two velocities track very closely yielding a constant normalized correlation of $\frac{\pi^2}{8}$. Once the constraint is acquired, the correlation drops and this can be used to detect the constraint-achievement event through comparing the correlation value to a constant threshold. The correlation value is related to the phase difference between the two signals because of the normalization.

$$C = \frac{\left(\sum_{i=0}^{N} f\left(\frac{i2\pi}{N}\right)g\left(\frac{i2\pi}{N}\right)\right)N}{\sum_{i=0}^{N}\left|f\left(\frac{i2\pi}{N}\right)\right|\sum_{i=0}^{N}\left|g\left(\frac{i2\pi}{N}\right)\right|} \tag{4-4}$$

$$C = \frac{\pi^2}{8}\cos\phi \tag{4-5}$$

A simple mechanics model illustrates what is happening. Consider the fact that the constraint really has finite stiffness as does the force sensor and assume that there is no gap. The motion of a point connected to a grounded spring is commanded and the command is modified through the force in the spring.

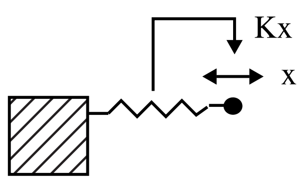


Figure 4-20: Mechanics of Constraint

The equation of motion for the endpoint of the spring under force control is:

$$\dot{x}(t) = V\cos(\omega t) - Kx \tag{4-6}$$

where K is the combined spring constant and force feedback gain.

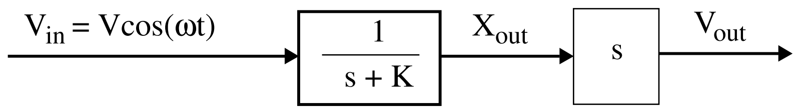


Figure 4-21: Laplace Transform of Command/Reference Velocities

The Laplace transform shows the relationship between the input (command) velocity and the force-perturbed ('reference') velocity. Since we use a normalized correlation function, we care about the phase difference between the two signals.

$$\phi(\omega) \;=\; 90 - \tan\left(\frac{\omega}{K}\right) \tag{4-7}$$

This phase equation is intuitively consistent. For very large stiffness K, the phase difference approaches 90 degrees and yields zero correlation. For small stiffness K, the phase difference approaches 0 which yields maximum correlation.

The effect of a gap is to reduce the effective stiffness. A gap can be modelled as a dead-

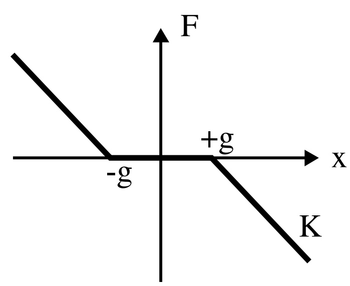


Figure 4-22: Stiffness with Gap

zone in the stiffness function (Figure 4-22). The effective (linearized) stiffness depends on the amplitude of x. For small x that remains within the gap, the stiffness is zero. For very large x, the stiffness approaches the upper bound of K. If we linearize the stiffness based on equivalent spring energy of the linearized model and the non-linear spring model at maximum deflection x=A, we can gain some intuition to the effects of finite gaps on the correlation value. Equation (4-8) is derived from setting the spring energies equal at the maximum amplitude A>g (where g is the gap size).

$$K_{AV} = \frac{K(A-g)^2}{A^2} \tag{4-8}$$

where K_{AV} is the linearized stiffness. If we let g = nA, where n<1, we can write:

$$K_{AV} = K(1-n)^2 \tag{4-9}$$

The amplitude of motion can be written as a function of this linearized stiffness based on the Laplace transform:

$$A = \frac{V_0}{\sqrt{\omega^2 + K(1-n)^2}} \tag{4-10}$$

If we pick some typical values for V_0, ω, and K, we can see the effects of different gaps (n) on the value A. Consider V_0=0.02 m/s, ω=3 rad/s, and K=10 (reasonable since stiffness of sensor might be ~10,000 N/m, but feedback gain is 0.001).

The following experimental plots show the correlation drop when a constraint is acquired and the correlation value when there is no constraint. The significant reaction forces perturb the input velocity and introduce a phase shift which is detected. In this case, the dithering continued and lost the constraint. Normally dithering should stop when the constraint is achieved.

Table 1: Gap Effects

n	A	g	$(1-n)^2$
0.1	0.0048	0.00048	0.81
0.5	0.0059	0.0029	0.25
0.9	0.0066	0.006	0.01
1	0.0067	0	0

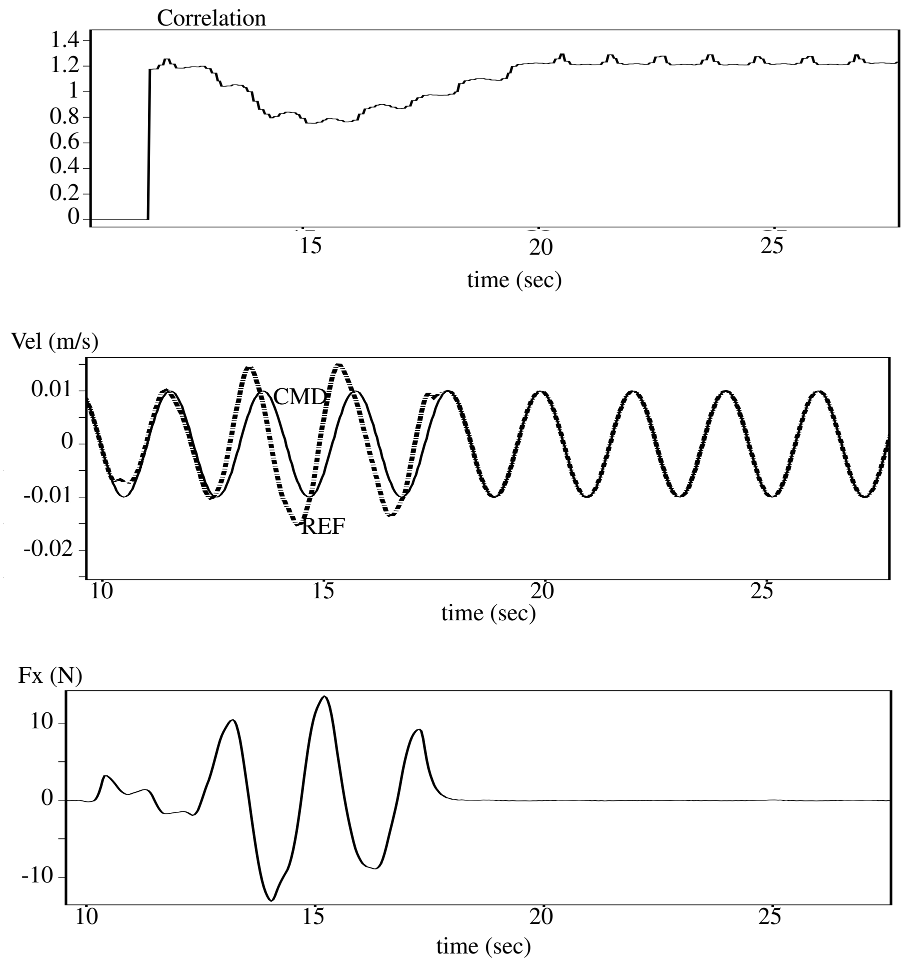
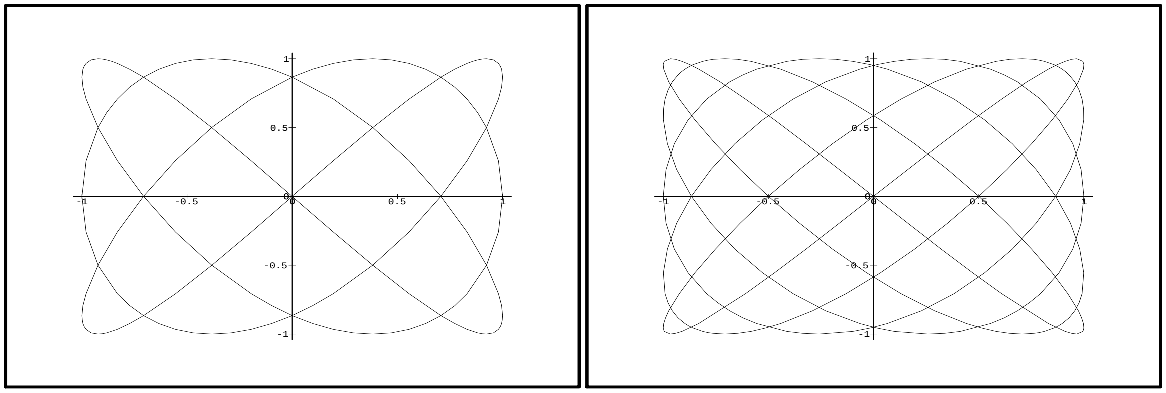


Figure 4-23: Correlation for Constraint Detection

4.9.4 Dither Combinations

Dithering and correlation have been joined into a linear/rotary combination or a linear/linear combination. If one dither frequency is f and is executed for N cycles, the other dither frequency should be chosen as (N+1)f/N and executed for N+1 cycles. This will ensure a Lissajous pattern is executed which explores the parameter space. Why dither like this instead of just randomizing? Because it combines the exploration search with the event-detection through the correlation. Also, the detection uses many data points and exploits knowledge about the trajectory input and controller effects in detecting the constraint acquisition event. In addition, a randomization requires an explicit step to determine the termination event.



n=3 n=5

Figure 4-24: Lissajous Patterns

4.10 Summary

In this chapter, the basic structure of a sensorimotor primitive: trajectory, control, and event detection was introduced. Damping force and corner-based visual servoing control resources were applied to instantiate MTP solutions to the different motion classifications under both contact and non-contact interpretations of the constraints. The hybrid control strategy necessary for contact interpretations was outlined and the implementations of non-contact interpretations of the constraints using visual feedback was discussed. Finally, a few transition primitives involving the design of event-detectors to robustly detect the transition from one constraint type to another were discussed. In the following chapter some of these primitives are used to construct skills for various assembly tasks.

Chapter 5

Robot Skills

5.1 Skills as Primitive Compositions

To address the problem of skill representation, the skill is expressed as a finite-state machine (FSM) which naturally supports the implementation of complex decision trees. The non-linear nature of a typical skill control law is realized through discrete control transformations. Rather than try to capture the entire strategy in some non-linear function or mapping, a segmented strategy achieves the non-linearity. Task events cause state transitions to occur during the strategy at which time a fundamentally different goal can be pursued with the subsequent controller, trajectory, and even-detector changes.

A skill makes demands of both the task and of the resources. The idea is to have the skill make modest motor demands easily met by a large class of robots -- i.e. ability to execute straight-line translation motions as well as rotations about a hand-fixed axis. The sensor requirements are also fairly general: a wrist force/torque sensor and an external CCD camera. However, the primitives have specific requirements regarding the task projection onto the sensors. This manifests itself as sensor placement relative to the task as well as specific sensor features which the task must produce. This task feature dependence is what specializes the primitive so it is important that reusable features are selected. The features may be artificial (i.e. fiducials) or natural (e.g. corners). The task feature projection assumption addresses the problem of extracting task relevant information from the sensor signal. It places constraints on the task/sensor interaction according to the processing algorithms used -- i.e the features must be visible to the sensor. This visibility requirement may spawn several different preconditions. For example a vision sensor may require that the lighting is sufficient to see the feature, that the feature be unique enough in the scene to identify and track, and that the feature lie within the field-of-view of the sensor. These primitive requirements are passed onto the skill as its requirements (at different states).

To implement these finite-state machines required extending the Chimera reconfigurable software framework developed by Stewart et al [67]. This extension is described in the next section and then example skills are described with experimental results presented.

5.2 Chimera Agent Level

Implementing these event-driven skills required extensions to the Chimera real-time reconfigurable software environment (Figure 5-1). The Chimera SBS level provides an excellent framework for creating modular real-time software for implementing periodic task modules [66] and is an enabling technology for this research. Without the ability to construct modular, reusable real-time software, skills could not be efficiently composed. The periodic modules can be naturally combined into controllers which have a ‘continuous’ nature. The higher-level, “meta-control” of these modules was missing.

Robot programming is inherently event-driven -- with different parts of the task program requiring different controllers, trajectory generators, and event detectors. The task strategy is segmented into different phases or states. This characterization points to the need for an asynchronous event-driven software level to complement the periodic SBS level. The SBS

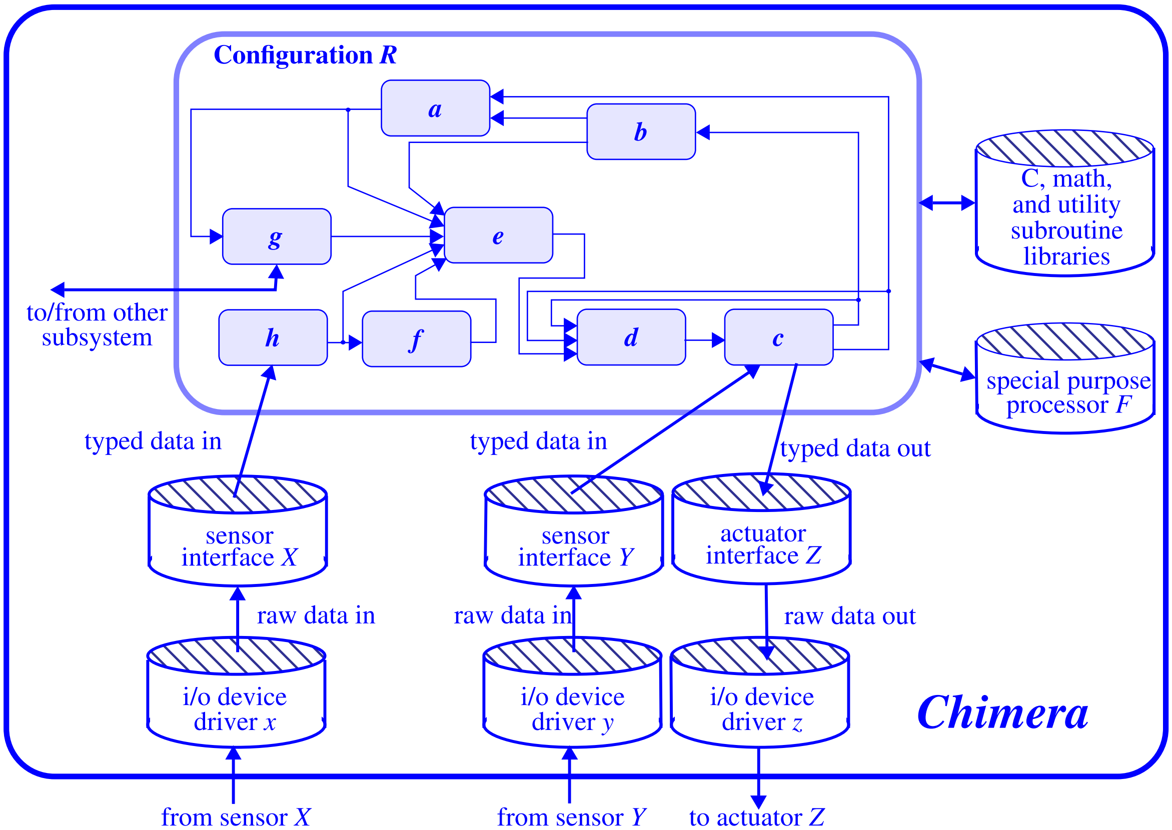


Figure 5-1: Chimera Reconfigurable Software Framework

level supports writing smaller, modular pieces of real-time code which can be reused -- this has the advantages of keeping the modules small and easing their development. However, composing more complex functions requires effectively and quickly managing sets of these modules. The Chimera Agent level addresses the need for this "meta-control" of module configurations. Whereas the SBS level is fundamentally modelled after periodic, port-based agents which process data, the Agent level is based on asynchronous, event-driven agents which process events and data. And whereas SBS modules have effectively two operating states (ON/OFF), agents in general have multiple, user-specified operating states. Generally an agent which manages n modules can have up to 2^n states, though in practice the number is usually much smaller (e.g. the controller agent manages 15 modules but has only four states: off, joint, cartesian, and damping).

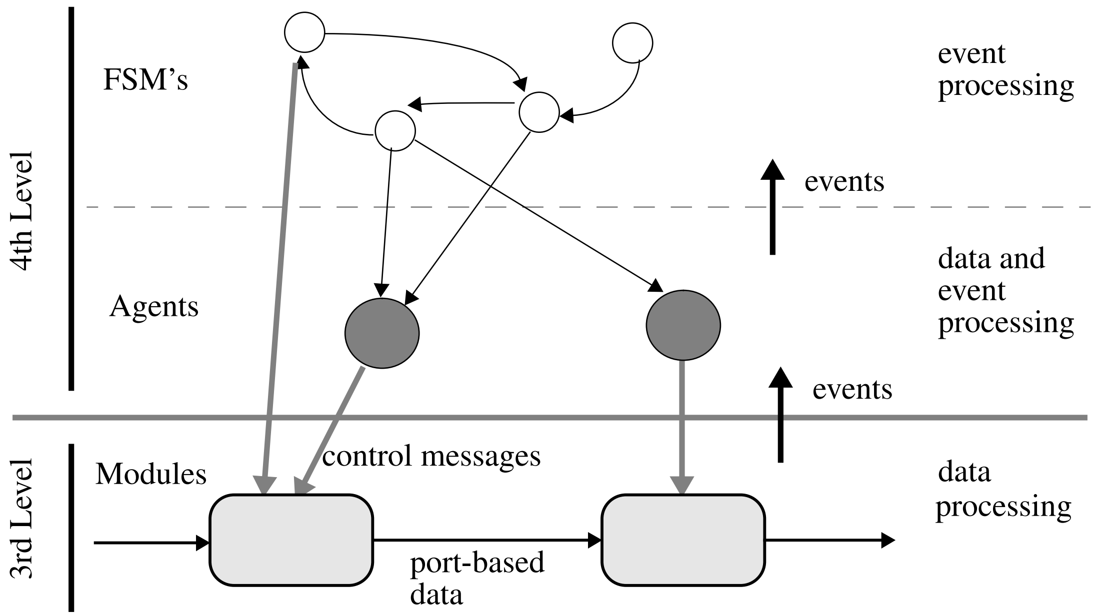


Figure 5-2: 3rd Level and 4th Level Interaction

SBS module management has several requirements. Encapsulating sets of these real-time modules into higher-level agents which can persist throughout an application provides the ability to hide agent complexity and details from its clients. Managing these module sets, or configurations, requires general-purpose computation to support complex decision making. Configuration decisions may also require access to the SBS state variable table to read relevant data from the 3rd level. Collections of modules usually require the module parameters to be shared and/or coordinated and the Agent level supports this as well. The highest-level programs must be quickly composable from existing primitives and modules without requiring a compile/debug cycle. The finite-state machine interpreter supports fast connection of events to configuration commands and the graphical user interface facilitates construction of these state-machines.

Two new objects implement the Agent level: agents and finite state-machines. Both are designed to be event-driven entities with multiple internal states. Their periodic components are executed as SBS port-based object modules and their higher-level, asynchronous coordination functions are executed in response to discrete events or commands. Agents are written in “C” code and can directly control SBS modules. State machines do not have general

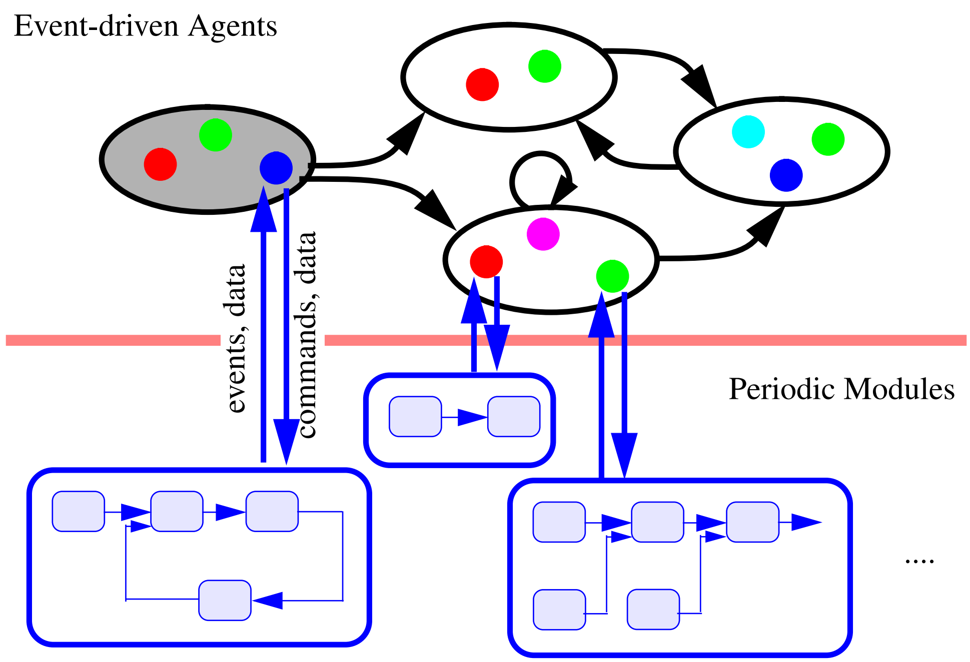


Figure 5-3: Chimera Agent Level

purpose computation capability -- they implement command and event transitions but they can reference SBS modules, agents, and other state machines and represent the ‘top-level’ of programs.

One of the key ideas implemented in the agent level is that of object persistence. Certain agents, for example controllers, persist throughout a task strategy but require different states during different strategy states. Also, the modules which connect directly to hardware (i.e. sensor device driver interfaces) must be able to service multiple clients. The agent level handles this by implementing a connection scheme whereby an object can be connected to by multiple higher-level objects. For example, object “A” to instantiate object “B” and then if object “C” can connect to “B”. If object A is destroyed B will persist since C is connected to it. In the case of SBS modules, when an agent/fsm turns an SBS module “ON” it becomes its parent and assumes authority over the object control. Once an object relinquishes control, the parent is reset and another object is free to control the child object.

The general-purpose computation ability available to agents means that complex algorithms can be implemented in them. Thus, these objects represent an extensible capability for constructing more complex, sophisticated agents. In particular, planning algorithms or more sophisticated error-recovery algorithms can be implemented as agents. Our controller agent provides joint, cartesian, and damping force control capabilities through shared SBS modules. Prior to the development of the Agent level, transitioning between different control modes was fraught with peril as very specific on/off sequences must be observed to prevent catastrophic behavior. In fact, early on, an accident caused by incorrect module sequencing during controller transition broke the gripper fingers when the gripper plunged into a hard surface. Dither/correlation agents which encapsulate orthogonal dithers for implementing a localized search/randomization coupled with event-detection have also been implemented as 4th-level agents.

FSM's are collections of states and transitions which move between states. Each state has two lists associated with it: a command list and an event list. The command list is an ordered list of control commands which can reference any type of object (module, agent, or fsm). FSM's can be nested up to 32 levels deep. A command has the form:

type	**cmd**	**object**	**parameters**

A command example would be "sbs on movedx L 0 1 0 0.01 0.02" The movedx is a simple differential move command which takes a code (L=linear move), an axis (0 1 0), a displacement (0.01m), and a speed (0.02 m/s).

An fsm event has the form:

type	**object**	**signal**	**nextstate**	**[exitcode]**

The last field is optional and only applies if the nextstate is 'halt' -- it indicates the "success" of the fsm termination. The signal is a 32-bit value which can be generated by any of the three types of objects. Generally, agents will trap events which their child modules generate and handle them. However, they may forward the event to their parent state machine for handling or generate a different event depending on the situation.

Five default FSM states exist: create, start, halt, destroy, and reset (Figure 5-4). The user is free to define additional states to compose an event-driven program. The command list is executed upon entry to the state. The event list defines the transitions away from the state as connected to specific events. Events may be generated by either SBS modules, agents, or other fsms. When entering the halt state, the fsm is effectively ‘finished’ and informs its parent of the outcome through an exit code (1=success, 2=failure). Whatever state transitions into the halt state supplies the exit code (the default is 2). A graphical user interface, the Skill Programming Interface (SPI), written in tcl/tk is available to support the rapid composition of event-based programs. The user interface is described in more detail in the Appendix of this thesis. The output of the SPI is a text configuration file which is loaded for real-time execution.

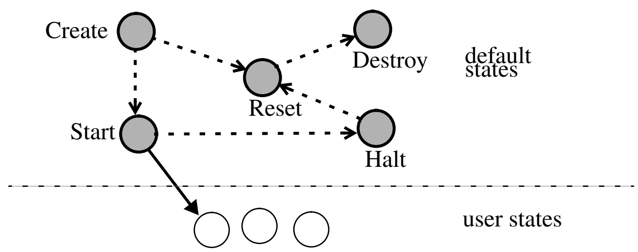


Figure 5-4: FSM Default and User States

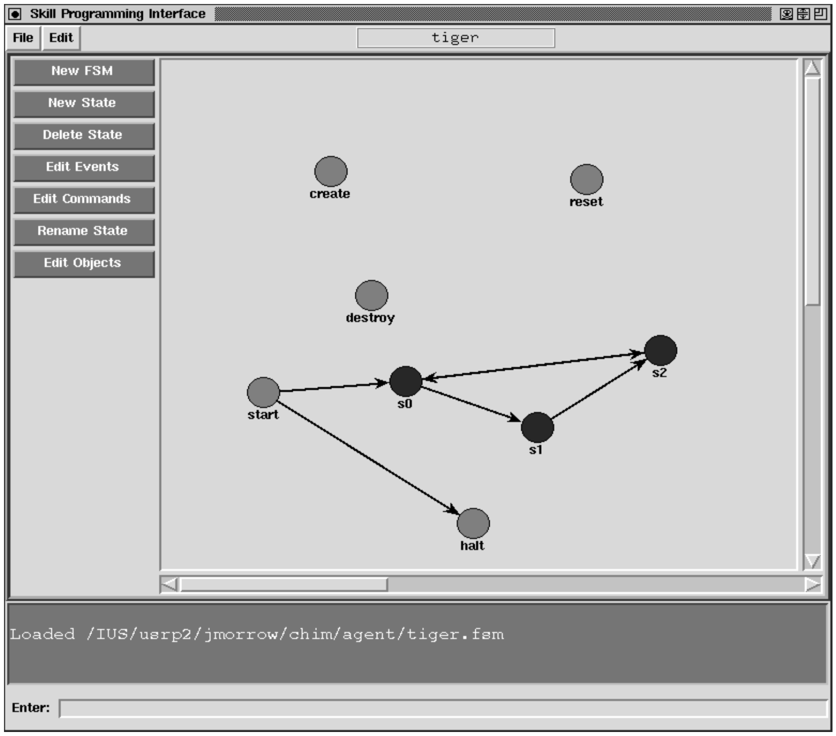


Figure 5-5: Skill Programming Graphical User Interface

5.3 Example Skills

5.3.1 Square Peg Insertion

No robotic manipulation thesis would be complete without a reference to the canonical peg-in-hole task. The task is a square peg insertion into a square cutout hole. The peg/hole clearance is not free -- the fit is actually a slight press fit and releasing the peg will not cause it to fall into the hole. The starting position is above the surface with the insertion axis approximately aligned. Two similar skills are implemented to perform this task and both assume approximately correct insertion axis alignment to start. Skill A also assumes rotational alignment about the insertion axis, while skill B allows error in this rotation. Both use force and vision feedback to execute the task. A guarded move down begins the strategy. After contact, corner features are selected on the two parts and a vision rotation move aligns the peg edge with the hole edge. Once rotational alignment is attained, a translation is executed under visual feedback to bring the two part corners together. This translation is a combined force and vision move -- vision commands motion in the plane while force maintains contact with the surface. The projection visual servoing primitive is used to mitigate the disturbances caused by visually-driven motions on the force-control direction. Once the corners are brought close together (defined on the image plane as a ‘target’ visual error), the peg is tilted into the hole and a guarded move acquires the side contact. The peg is pushed against the hole side while slowly straightening and introducing a small rotational dither about the peg insertion axis. When the rotational error was initially zero the insertion was

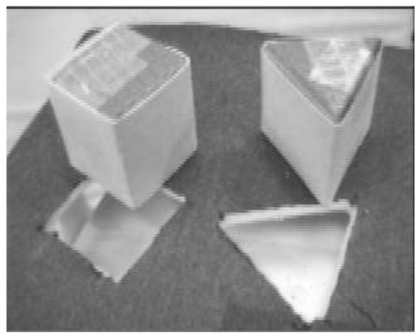


Figure 5-6: Square and Triangular Peg Insertion Tasks

much easier. With rotational error, the error was not always completely removed due to noise in the feature tracking -- the error might still be a few degrees. This was significant enough to prevent mating. So for skill B, additional dithering was introduced both when straightening and during insertion to help correct this additional error.

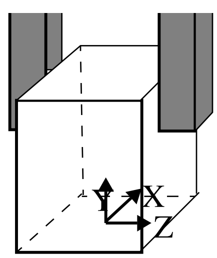


Figure 5-7: Block Task Frame Assignment

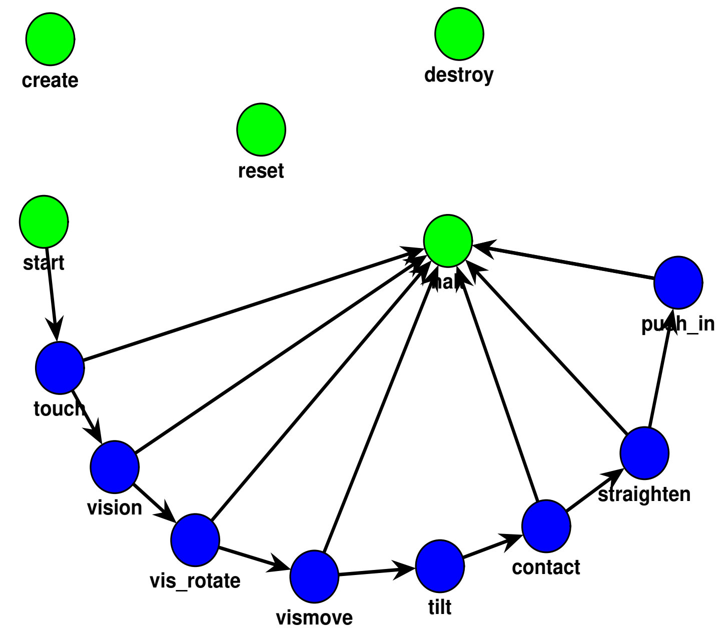


Figure 5-8: Block Insertion Skill

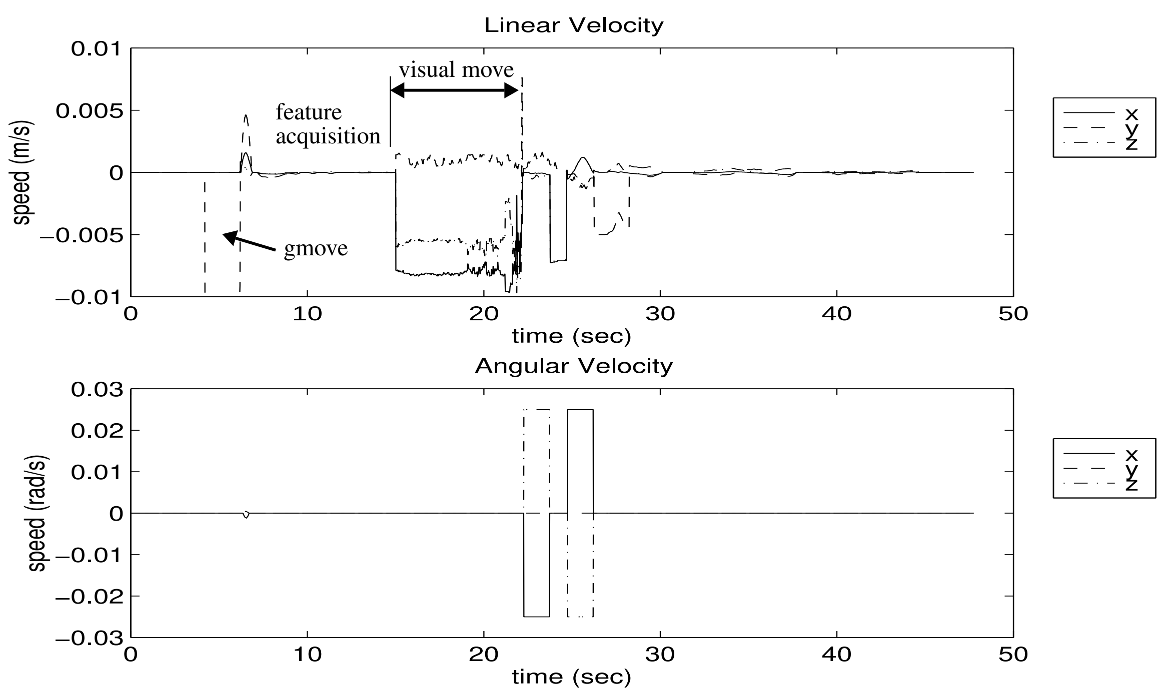


Figure 5-9: Block Skill "A" Velocities

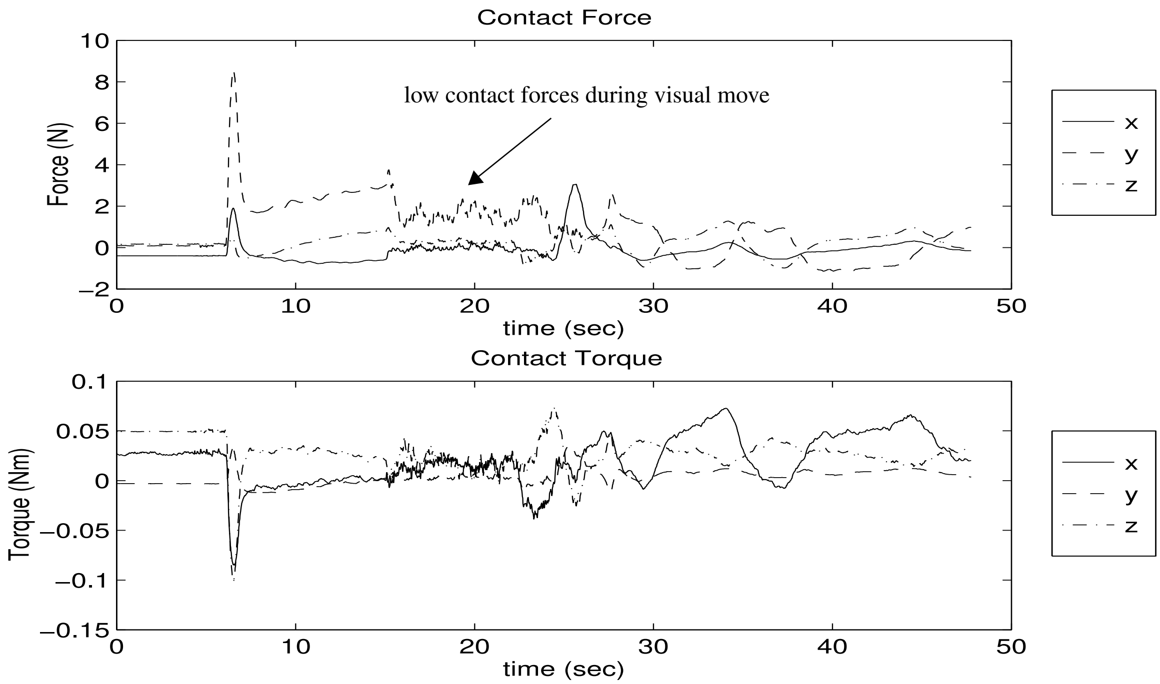


Figure 5-10: Block Skill "A" Forces

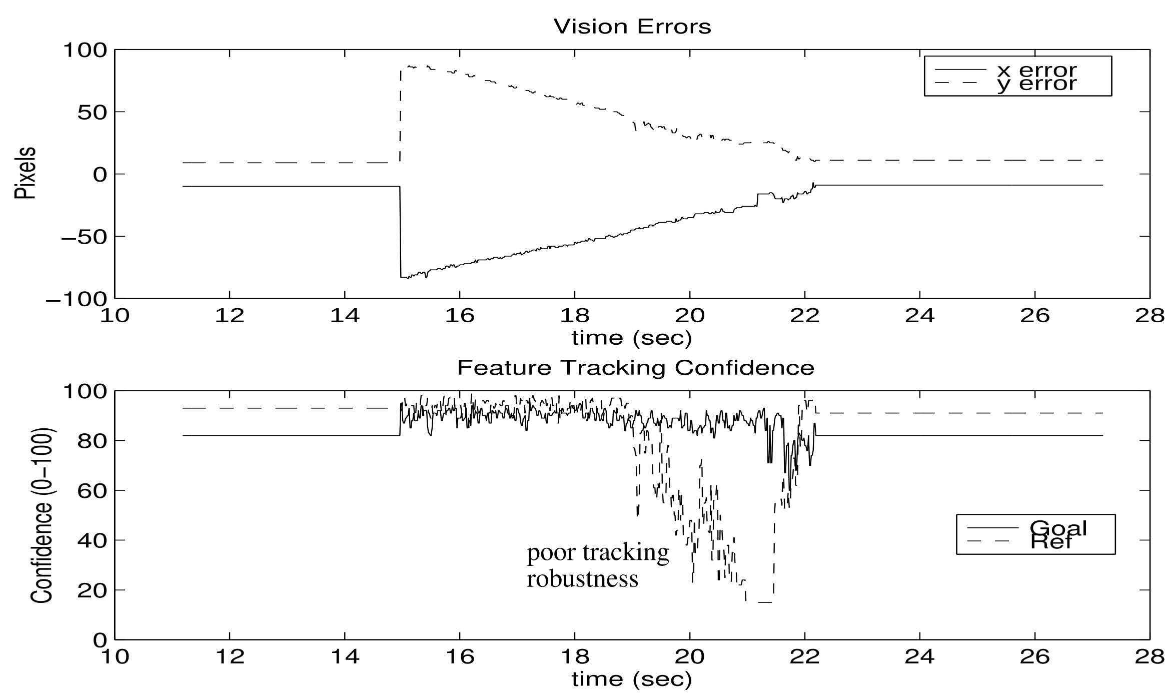


Figure 5-11: Block Skill "A" Pixel errors and confidence

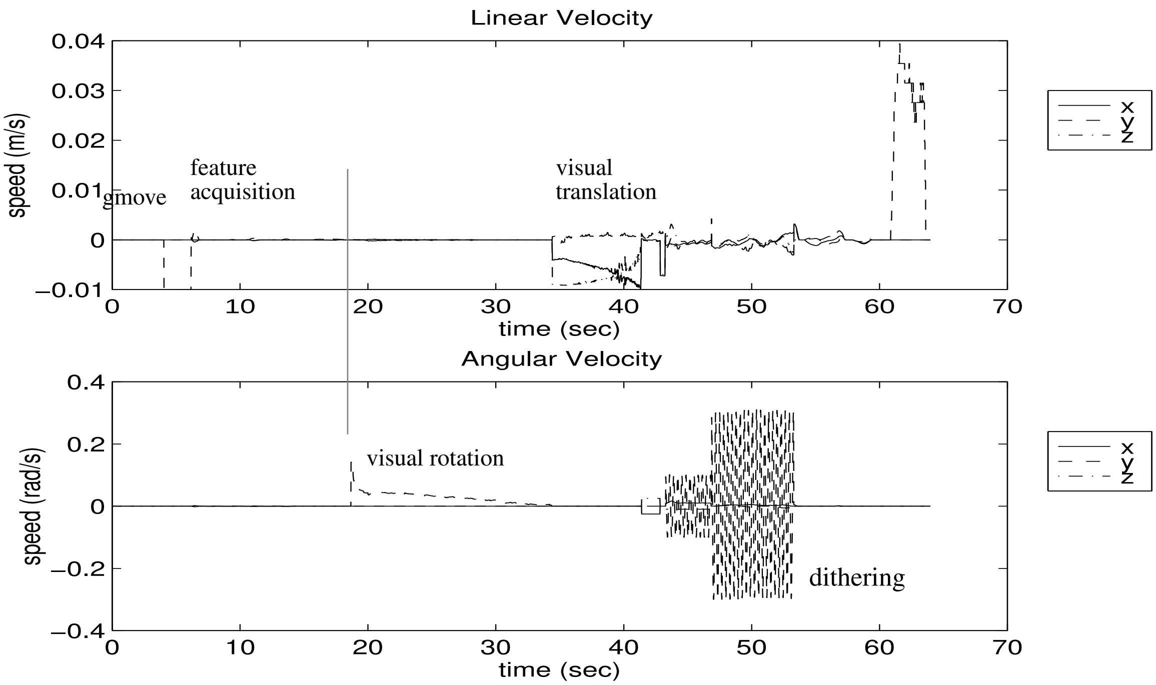


Figure 5-12: Block Skill “B” Velocities

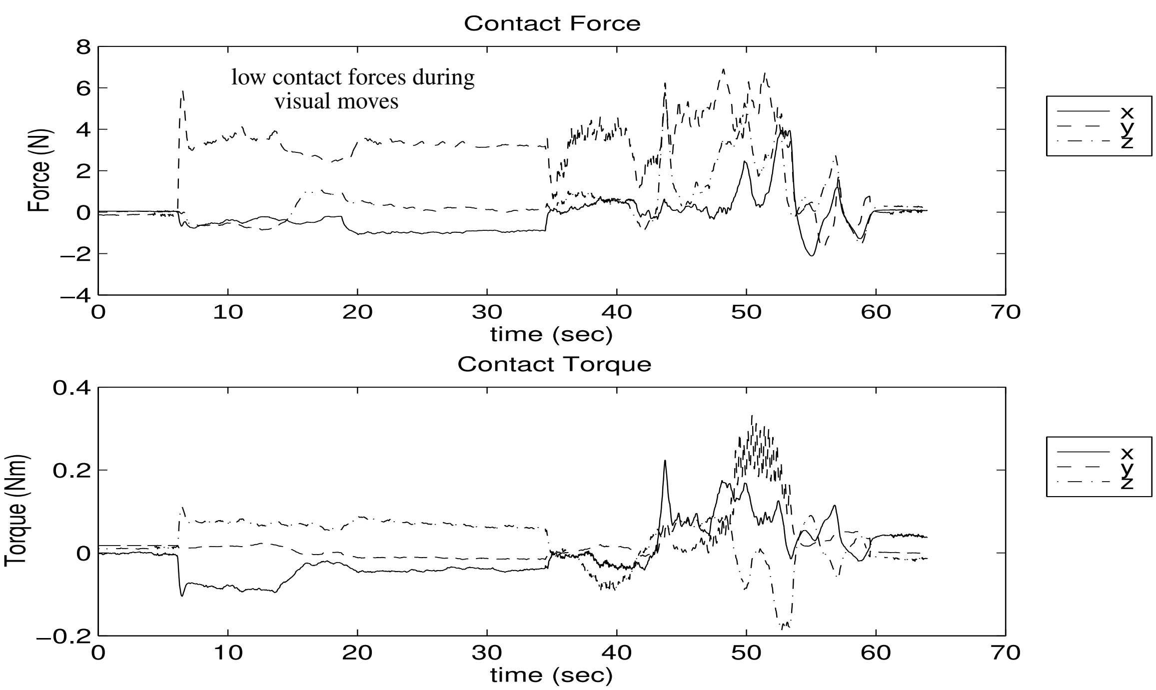


Figure 5-13: Block Skill “B” Contact Forces

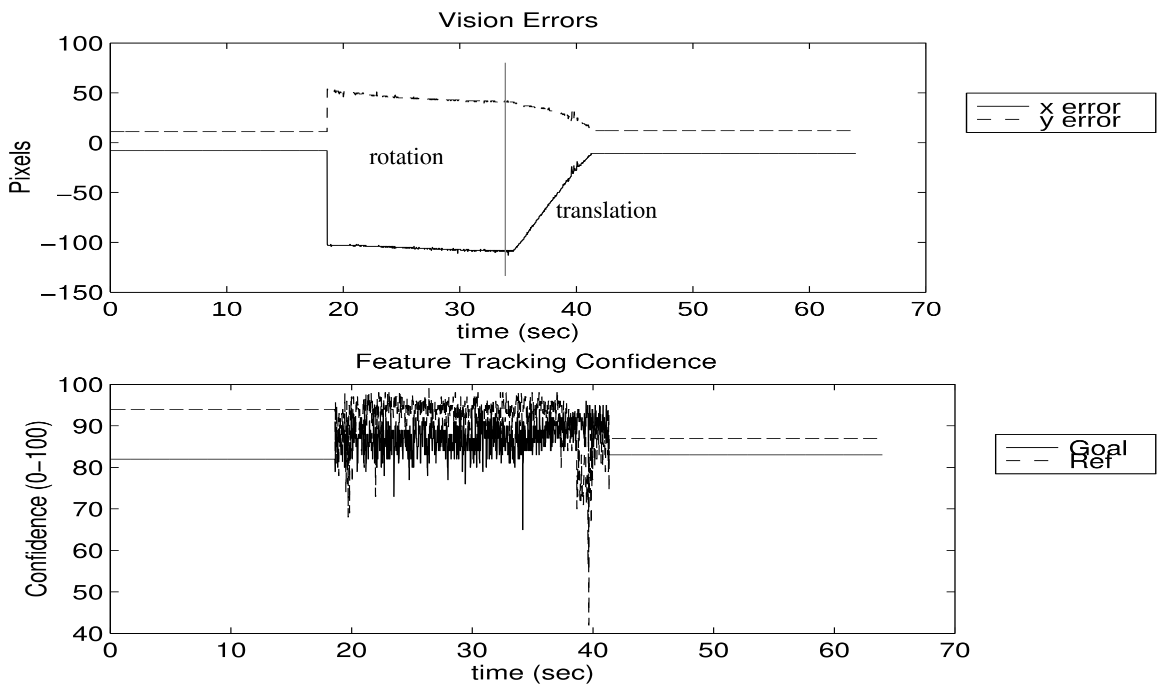


Figure 5-14: Block Skill “B” Visual Error and Tracking Confidence

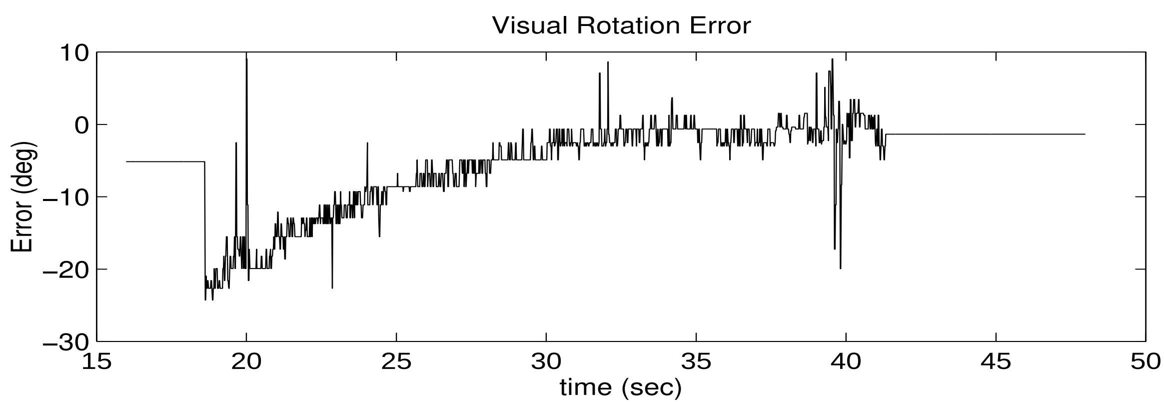


Figure 5-15: Block Skill “B” Rotation Error

5.3.2 Triangular Peg Insertion

The triangular peg is shown in Figure 5-6 along with the square peg. The triangular peg insertion was made easier by slightly enlarging the hole with a file and producing a chamfer on the hole. This gave a couple of millimeters of clearance instead of the slight press fit for the square peg task. We used the exact same skill program for this task as the square peg task.

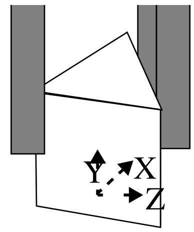


Figure 5-16: Triangular Peg Task

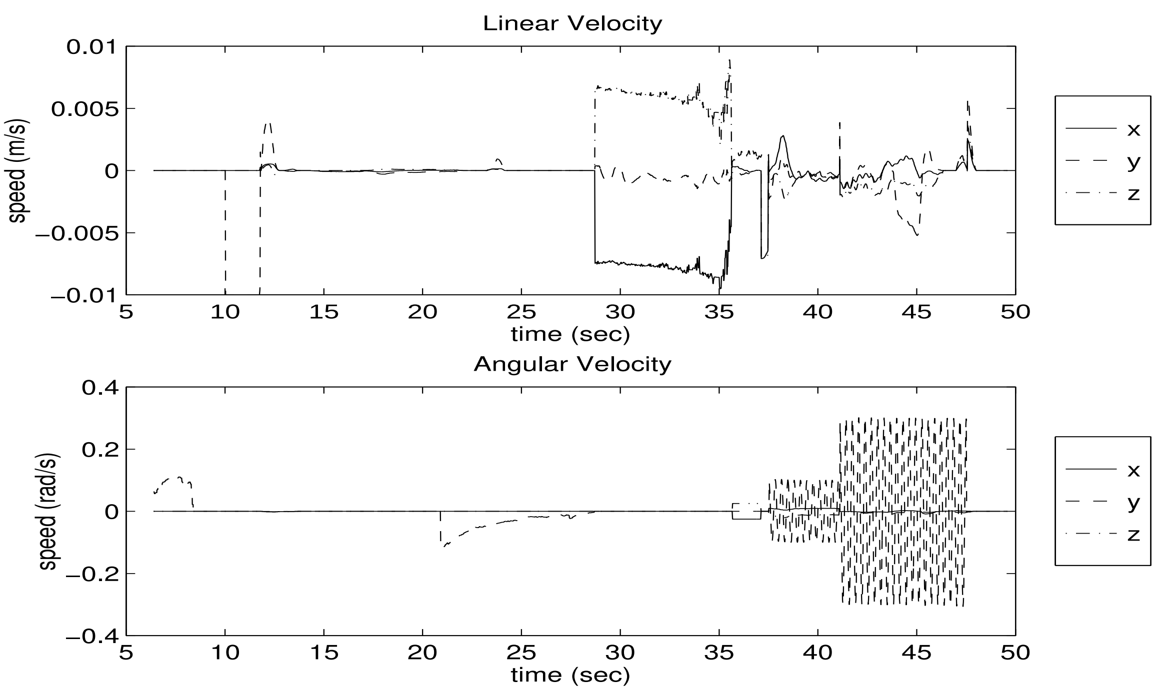


Figure 5-17: Triangular Peg Velocities

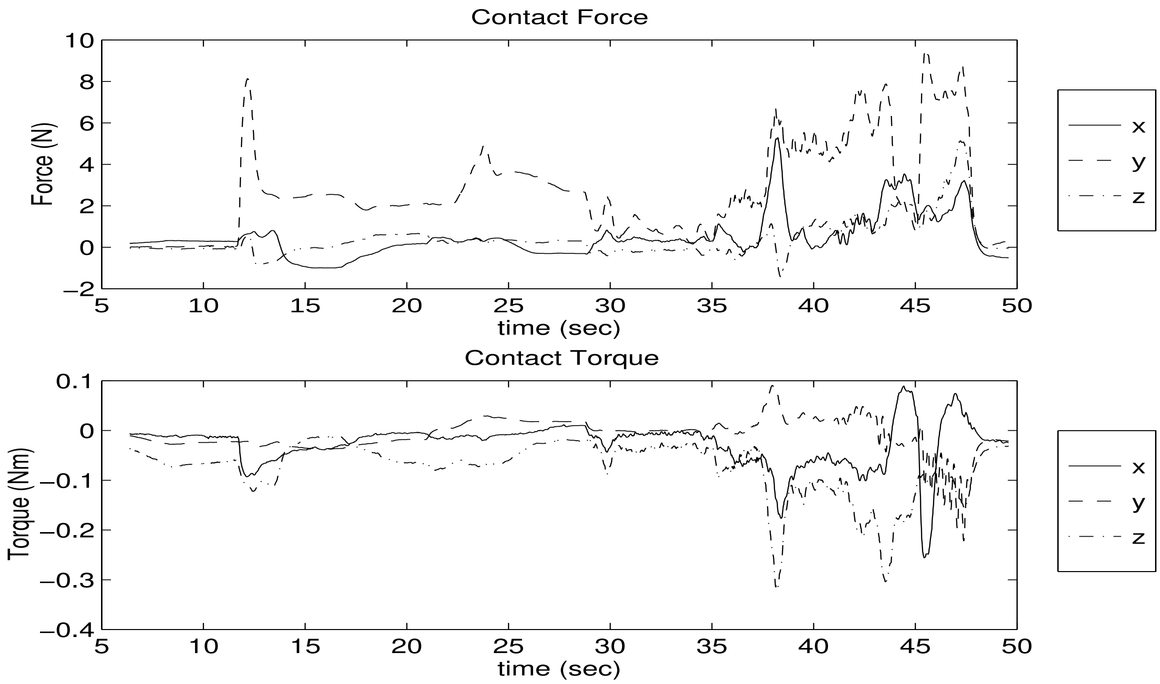


Figure 5-18: Triangular Peg Forces

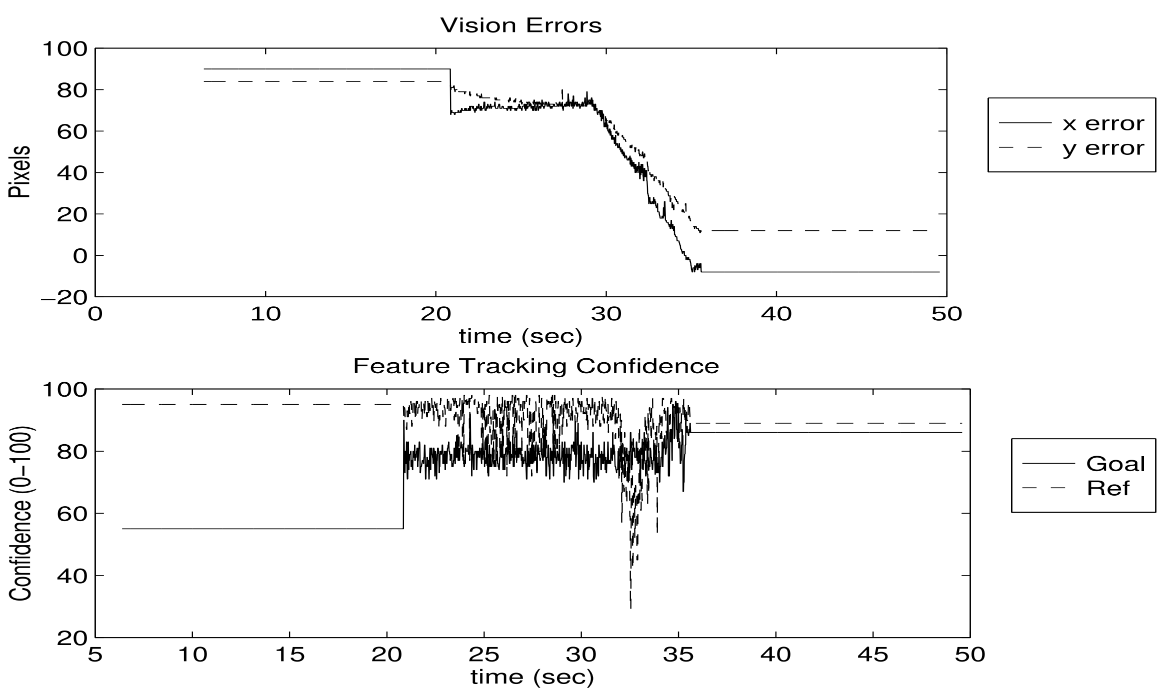


Figure 5-19: Triangular Peg Visual Errors and Tracking Confidence

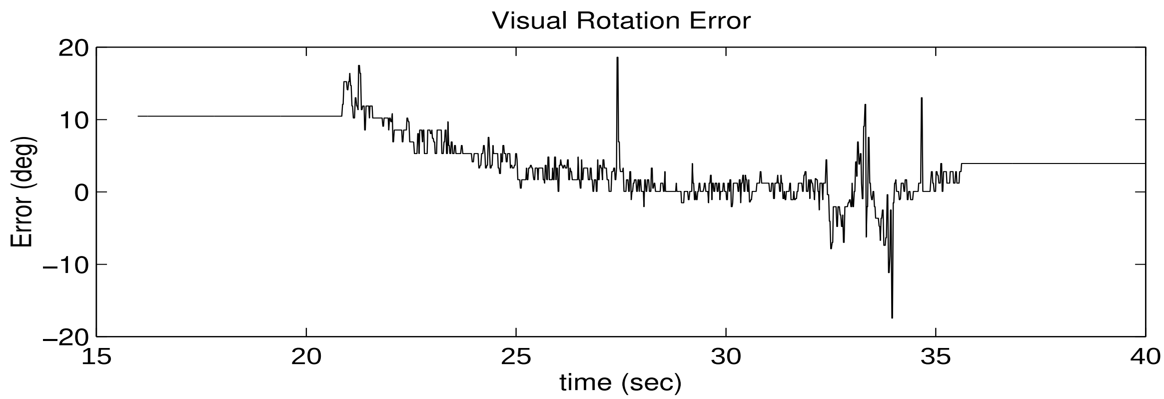


Figure 5-20: Triangular Peg Visual Rotation Error

The peg insertion tasks demonstrate the use of the combined force and vision primitive for constraining all three translation DOF. The next tasks are all connector insertion tasks. The task strategy implemented by primitives results in a command velocity, V_{cmd}, which is perturbed by the accommodation controller in response to contact. The perturbed velocity, V_{ref}, is used to generate joint setpoints for the robot joint controller. The experimental results for the BNC and D connector tasks are shown as plots of V_{ref}.

Before presenting the results of the connector insertion tasks, grasping results from using decomposition in the visual servoing control law to control 3 translation DOF for approaching and grasping a connector are presented. This vision-driven grasp shown in Figure 5-21 was used as part of the connector insertion strategies. The feature acquisition and placement was manually performed for this experiment and SSD feature tracking was used. Two cameras placed approximately orthogonally are used to track features on the parts -- one camera is used to drive two DOF, and the second camera controls the “depth” DOF. There are three distinct phases visible in the grasp: the transport, the approach, and the depart. The transport involves visual servoing along two directions, but not the approach direction to align the gripper above the part. The approach involves mainly a straight-line motion along the approach direction, but all three translations are visually servoed to compensate for calibration-induced errors. Finally, the part is grasped and an open-loop depart move is executed to move away from the table.

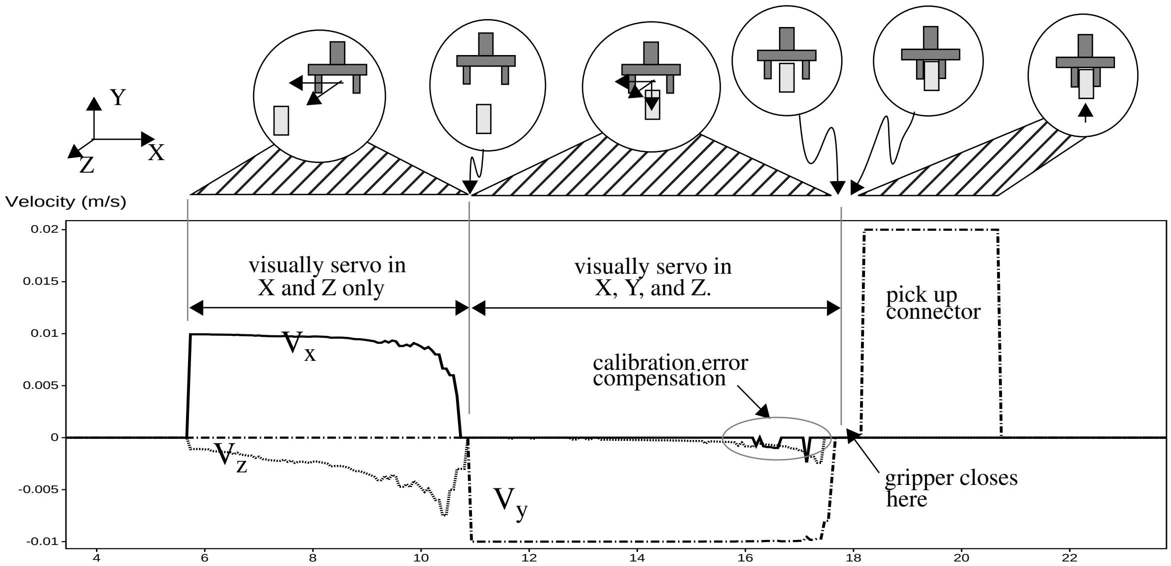


Figure 5-21: Vision-Guided Grasp

5.3.3 BNC Connector Insertion

Figure 5-22: BNC Connector Task

This task is a standard BNC connector insertion. The BNC connector has a very different geometry than the D-connector, but the same primitives are used to implement a task strategy. The BNC insertion strategy has the same three basic steps as the D-connector insertion: 1) grasp connector, 2) transport to the mating connector, and 3) insertion. The grasp and transport phases are essentially the same as those for the D-connector. However, the insertion phase in this case is a more complex, multi-state event, and force-sensing is used to trigger the transitions. The first part of the insertion step is the guarded move followed by dithering (and correlation) to acquire the first constraint: no translation in the plane defined by the insertion axis. Once this constraint has been acquired, the holes in the bayonet must be aligned with the stubs on the connector shaft. The *movedx* primitive performs the rotation while the *stick* primitive maintains contact and monitors for movement along the insertion axis. Once movement along the insertion axis occurs, we know that the connector has mated with the stubs so we terminate the rotation. Another rotation locks the bayonet and finally the connector is released. The *stick* primitive in the D-connector task was used to detect failure since the primary failure mode was losing contact. Here the same primitive is used to signal a task state transition when the bayonet mates with the shaft stubs.

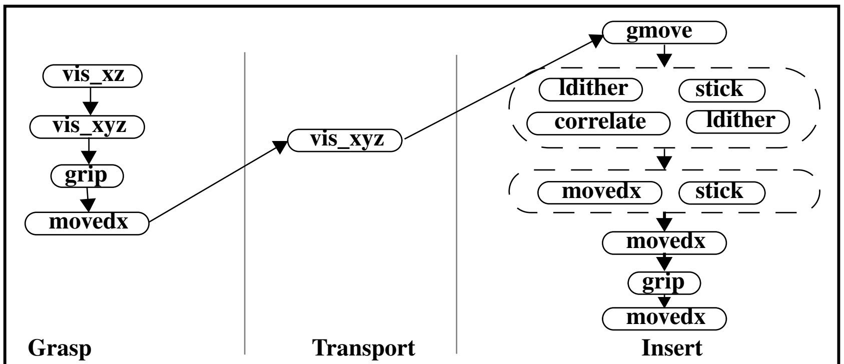


Figure 5-23: BNC Insertion Strategy

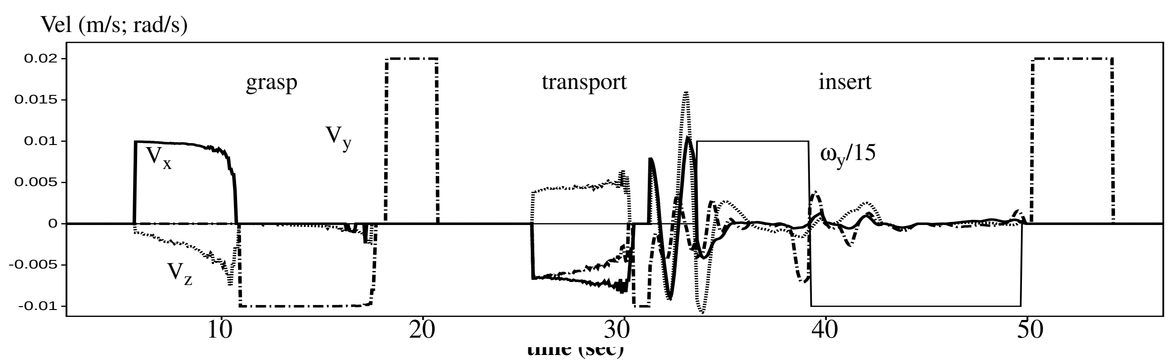


Figure 5-24: BNC Connector Insertion Results

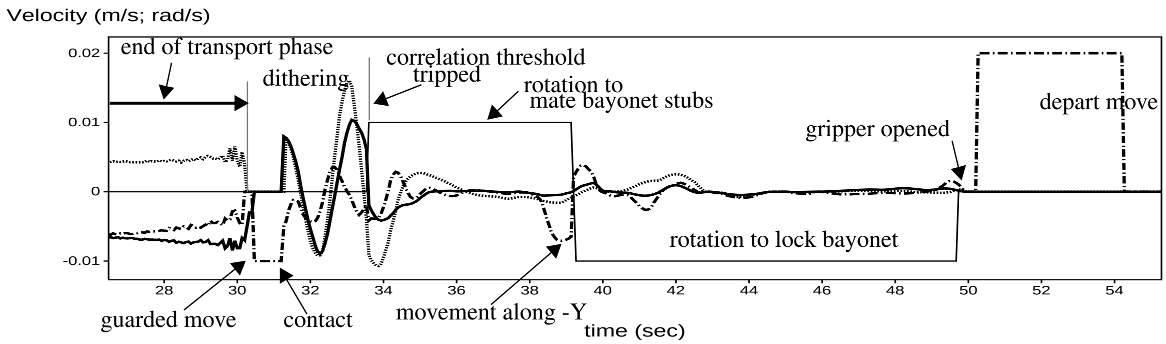


Figure 5-25: BNC Insertion Stage

5.3.4 D-connector insertions

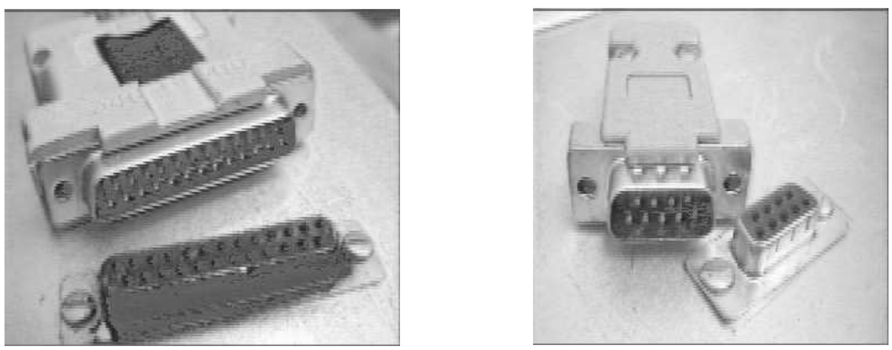


Figure 5-26: 25 and 9 pin D-connector Tasks

This task is a basic 25-pin D-connector insertion. Given the small scale of the contact, we cannot reasonably derive a strategy based on a detailed-contact analysis. Instead, a heuristic strategy was developed based on the available command set and sensing. The strategy is based on the available command primitives (some sensor-driven, some not) and is shown as a finite-state machine (FSM) in Figure 5-27. The basic strategy has 3 steps: 1) grasp the connector, 2) transport to the mating connector, and 3) perform the insertion. The first two steps are dominated by vision-feedback; the third step is dominated by force feedback. The first step, grasping, relies on approximate angular alignment of the connector axes (X, Z) with the camera optical axes. Visual setpoints are identified in the images and controlled through visual feedback. The second step also involves using visual feedback to position the grasped connectors above the mating connector for insertion. The insertion step involves a guarded move, followed by a mixture of "sticking" along with rotational and linear sinusoidal dithering (different frequencies) and correlation monitoring of the linear dithering. The dithering introduces enough variation in the command to resolve small uncertainties left over from initial positioning. The correlation of the commanded and force-perturbed reference velocities provides a means to reliably detect when the connector has seated. Note that this "success-detection" method does not rely on attaining absolute position goals, but rather on attaining particular motion constraints.

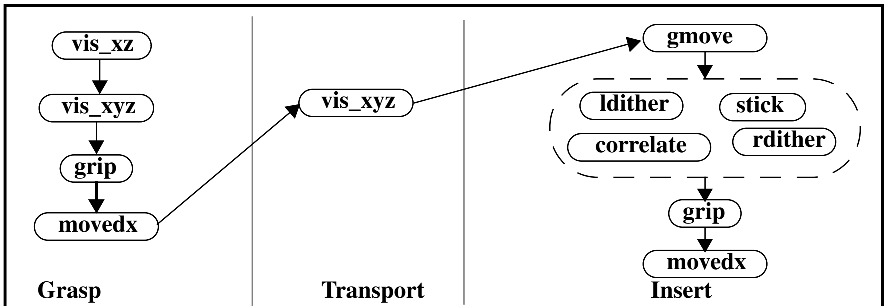


Figure 5-27: D-connector Insertion Strategy

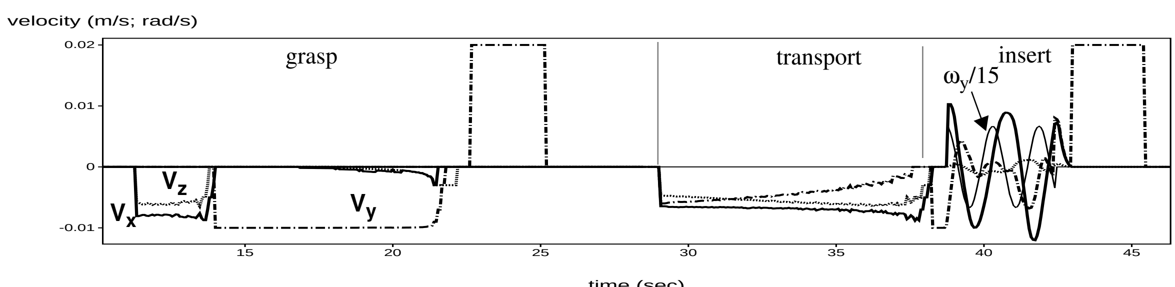


Figure 5-28: D-connector Insertion Results

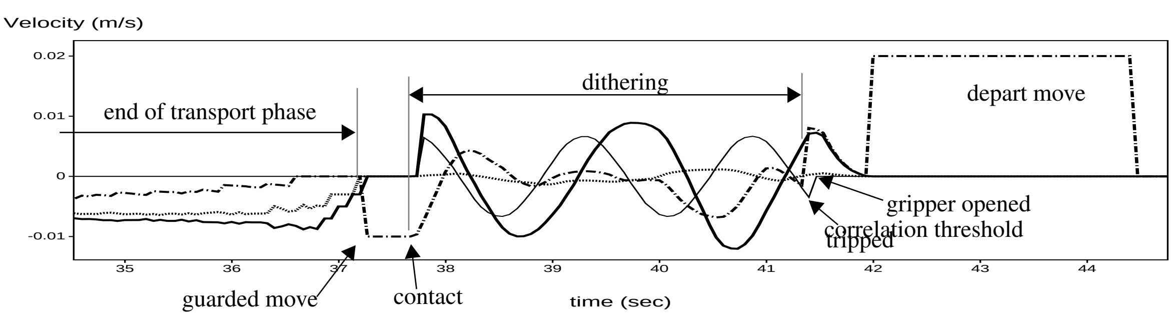


Figure 5-29: D-connector Insertion Stage

5.3.5 Press-fit Connector

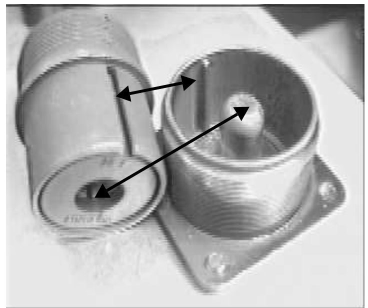


Figure 5-30: Military-style Press Fit Connector

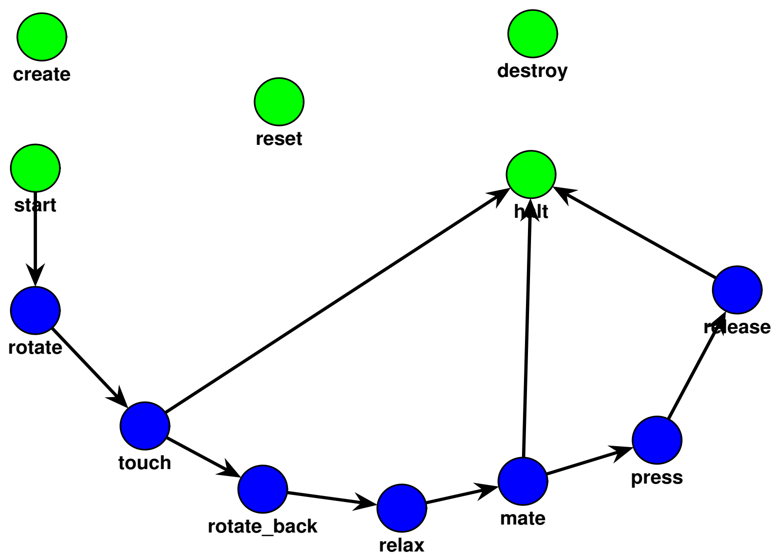


Figure 5-31: Press-fit Connector Skill

This strategy was performed without visual servoing feedback because feature tracking was especially difficult for this task because the threads on the female connector add texture which is not allowed by our feature tracker. Another fiducial mark could be added to improve this. The basic strategy is to tilt the peg by 5 degrees, and move to acquire the contact. Then, while applying a sticking force in Y and Z, rotate slowly to vertical to trap the peg inside the hole. At this point, the peg is resting on top of the ridge inside the hole which is only a mm or so from the top surface. At this point a rotational guarded move is executed to line up the slot. Then the peg is pushed down with 40N of force to accomplish the press fit. We found it necessary to introduce a fast dither in Z to ease the insertion. The values of the parameters are selected by the human and are fairly arbitrary. One of the extensions of this research is to optimize the parameter values to maximize robustness and/or performance.

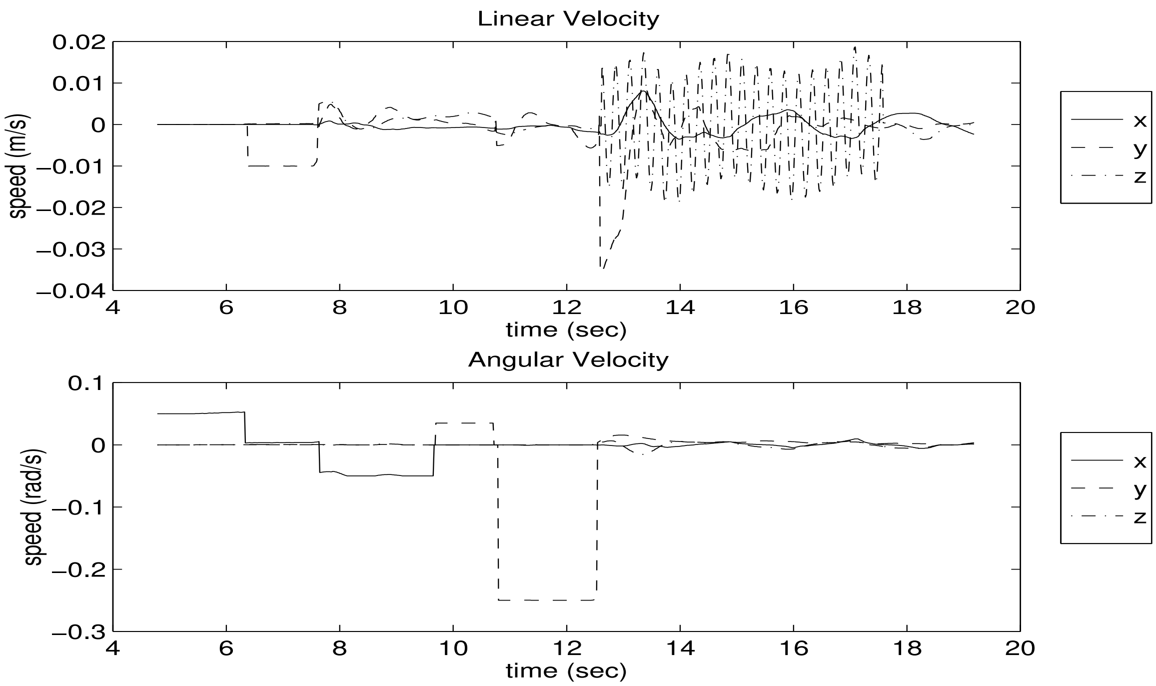


Figure 5-32: Press Fit Connector Velocities

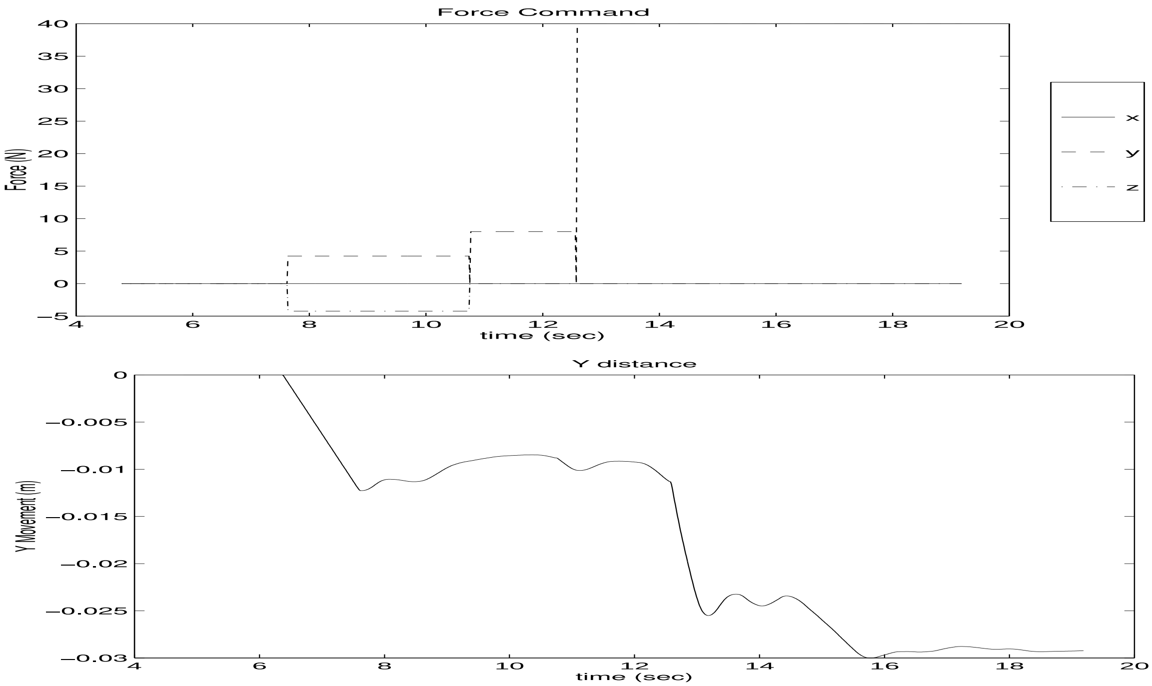


Figure 5-33: Press Fit Connector Force Command and Y Movement

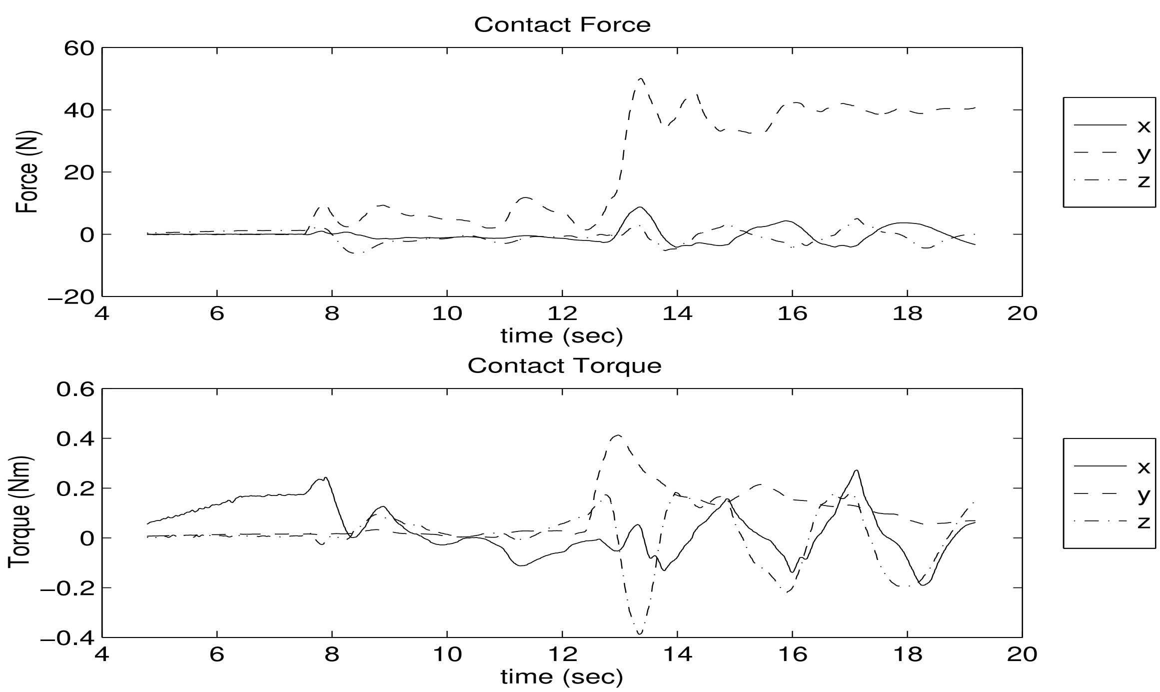


Figure 5-34: Press Fit Connector Forces

5.4 Summary

This chapter presented a finite-state machine representation of a sensor-based skill. Each part of the skill must specify a controller, a set of trajectories (in force and/or velocity) and event detectors to force state transitions. The Chimera Agent 4th Level was introduced for constructing event-driven real-time programs. The Agent level supports objects which process only events (fsm's) and objects which can process both events and data (agents). Agents provide a bridge between the fast data processing occurring on the 3rd level through periodic, port-based modules and the asynchronous event-based state machines in the 4th level.

Six different, but related tasks demonstrated experimental solutions with shared sensor-based primitives based on damping force feedback and visual servoing. The strategies are differentially specified -- no absolute positions are used as cues for execution on termination. This allows the skills to be used wherever a higher-level program can initialize them appropriately relative to the task. This differential nature of skill definition is contrasted with some other skill-building efforts which use absolute position data as input and output -- for example Gullapalli's peg insertion skills [23]. Philosophically the idea is to get away from specifying tasks in terms of absolute robot movement and toward specifying tasks in terms of the robot's view of them -- their projection onto the robot's sensors.

The weakest part of the skill execution is the vision integration -- feature acquisition and tracking. Currently the feature selection is done manually which obviously limits the usefulness of a skill. In the next chapter a very preliminary approach to automatic feature acquisition is presented. More serious is the (relative) lack of robustness in feature tracking. The corner feature tracker is more robust than SSD for tracking occluding boundaries, but the confidence measures clearly decline when the features are brought close together. What is needed are more sophisticated trackers which incorporate task model information to provide additional constraints for robustness. This brings up two problems: 1) making the trackers too task-specific, and 2) increasing the computation time. The obvious first step is to key on fiducials which are designed into the task(s). Keying on 'natural' task features is more challenging but recurring features like edges, vertices, and surfaces can be used. One advantage of fiducials is that both geometric projection and photometric effects can be addressed. The photometric effects are extremely significant and often ignored [19]. Specularity, shadows, and other real-world imaging phenomena caused significant problems with simple feature tracking algorithms. However, fiducials are usually internal targets which necessarily introduce offsets -- these offsets introduce additional sensitivity to camera placement uncertainty.

The experimental results presented here support the central claim that different but similar tasks can share sensor-based commands in their strategies. This has been experimentally demonstrated with different assembly contact tasks including difficult connector insertion tasks.

Chapter 6

Design Agents

6.1 Introduction

This thesis has presented the concept of a sensorimotor primitive as a domain-specific building block for composing sensor-based robot skills. To maximize generality for rigid-body assembly tasks, the primitive derivation is based on the different types of relative motion between two rigid parts. However, we still cannot guarantee that *any* manipulation task can be solved using these primitives. There are clearly manipulation tasks outside of this domain: machining tasks, fabric handling, etc. But even for tasks within the domain, the primitives require specific conditions to be met. For example, the visual servoing primitives are based on corner feature projections. If the task does not produce these projections, then it cannot take advantage of those primitives. The task features are crucial for determining the applicability of a particular primitive. This strong coupling between the task characteristics and the robot capabilities is already evident in robot programming and led to *design for assembly/manufacturing* methods. It is well-accepted that task design is a crucial step in task manufacturing automation. Poorly-designed tasks can be very difficult and prohibitively expensive to automate. Rather than try to devise a set of 'guaranteed' primitives, a more pragmatic approach is adopted. Accepting that primitive applicability is strongly influenced by task design, the focus is shifted from trying to guarantee a primitive sets' applicability toward explicitly trying to exploit the capabilities of a particular set of primitives *during the task design phase*. This goal leads to the expansion of the concept of a primitive from merely a run-time component to one which also possesses a design component.

Representations of the primitive capability in the design environment are called *design agent*s. The primitive has both an instantiation in the real-time system as a collection of real-time software modules, and a representation in the design environment as an agent object (data + methods). Each primitive has several different types of design agents to support different phases of primitive application: selection, parameterization, and execution/monitoring. The design agents interact with both the CAD model and the user to help the user select a primitive for a particular task step, parameterize the primitive based on the task configuration, and predict the primitive performance through simulation of sensor-based primitive execution on a dynamic model of the task.

Supported by the design agents in the design environment, the task designer concurrently produces the task skill along with the task design (geometry). He is able to determine, at design time, whether or not the task is executable with the available primitive set. This affords the opportunity to modify the task design (geometry and mechanics) to exploit the primitive capabilities or to identify new requirements for primitives should the required task design modification(s) not be feasible. Moreover, the C^2T system provides the task designer with the tools to compose an executable sensor-based program through interacting with the task model.

This chapter covers three topics: 1) an overview of the Chimera/Coriolis/Telegrip (C^2T) system which integrates a CAD environment, a simulation environment, and a real-time control environment; 2) how the C^2T system is used in CAD-based skill construction; and 3) specific examples of primitive design agents.

6.2 Chimera/Coriolis/Telegrip (C^2T) System

To support the integration of SMP's into the design environment, three different systems have been integrated: Chimera, the real-time system which executes primitives to control the robot; Coriolis, a 3D mechanics simulation package capable of modelling rigid-body collisions and impacts under Newtonian mechanics assumptions; and Telegrip, a 3D CAD design environment with simulation capabilities. This section only briefly describes the role of each of these systems -- details about the integration are given in the Appendix.

The corner-stone of primitive implementation and execution is Chimera, our real-time operating system running on a multiprocessor VME system. Primitives are implemented as real-time reconfigurable software modules in Chimera. Telegrip is a 3D CAD and simulation environment targeted at modelling and simulating robotic systems. It is used for geometry creation, display, and a menu-driven user interface. Although Telegrip has dynamic

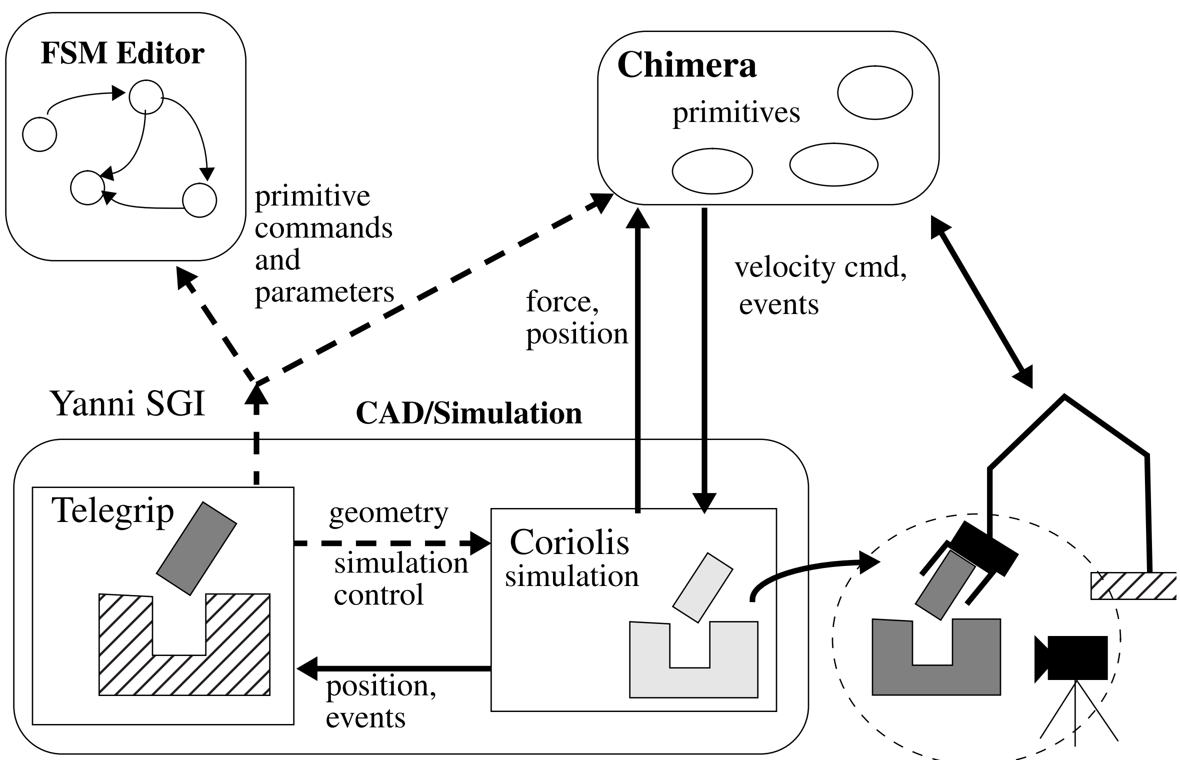


Figure 6-1: C^2T System

simulation capabilities, they are focused on modelling robot dynamics. Telegrip excels at "programming in the large" -- that is, laying out workcells and writing robot programs which can be played in simulation to check for collisions between robots and parts of the workcell. What is missing from this type of tool is "programming in the small" or programming skills. Skills are typically fine-motion or gross/fine transition motions which require sensor feedback to robustly accomplish. Rather than the robot dynamics, skill composition requires modelling the task dynamics, especially of contact tasks.

The task model consists of two parts: one moving and one fixtured. The moving part will be grasped and moved by the robot; the grasp location/transform is determined by placement of a *grip* frame on the part geometry. In addition, a *task* frame is assigned to the moving part which corresponds to the center of compliance. The sensor signals are transformed to the task frame and force control is performed about it. Any velocity commands (e.g. vision) apply to the task frame location and in its coordinate system. After creating the part geometry in Telegrip, the geometry is exported to Coriolis.

Coriolis [6] is a C++ library for simulating rigid-body dynamics including collisions and impact under Newtonian mechanics assumptions. Coriolis allows us to simulate tasks involving contact and provide simulated force feedback.

Besides modelling the task mechanics, Coriolis also replaces the robot and the force sensor during simulation. The robot is modelled as a frame which is kinematically-controlled from a velocity input. For force feedback, a stiff 3D spring models the force sensor in simulation. The task model completely abstracts away the robot to its end-effector frame, and focuses on the interaction of the two task parts. This simplification explicitly ignores robot-related limitations (e.g. singularities and reach). Instead, the focus is on applying the sensor-based commands to solve the task -- the resulting strategy (or skill) can later be tested against specific robot limitations. For example, candidate robot systems must possess the ability to command Cartesian motion and have force feedback through wrist force sensing available. Whether or not robot singularities are violated depends on the specific robot and the location of the task in the robot's workspace. Skills are usually built for 'fine-motion' tasks. Although the displacements involved in a skill (a few cm) might be huge relative to task considerations (e.g. clearance), they are small compared to the robot's capabilities.

The force sensor mechanically connects the robot frame, which is kinematically controlled via a velocity command, and the moving task part (Figure 6-2). Moving the robot frame will move the part along with it. Collisions of the part will result in spring deflections and, under force control via Chimera, perturbations to the robot velocity. The robot commands are expressed in terms of the task frame located in a task-relevant position. The force sensor is modelled as a stiff spring with a diagonal stiffness matrix. Little effort was spent trying to make the simulated force sensor match the real sensor because the simulated force sensor includes all compliance in the system. The linear spring constant is 30,000 N/m, the gripper weight of 0.92 kg, and the damping coefficient is chosen to give critical damping. The rotary spring constant was 300 Nm/rad, with 0.0093 kg/m^2 rotational inertias along each principal direction, and critical damping. With these parameters, the Coriolis time step was 0.001s which is the time quantum (for task scheduling) of the Chimera system. When in contact, the simulation begins taking smaller 0.00025s steps (four are taken before corresponding with Chimera).

The robot receives discrete velocity command updates and each update defines a new segment. When the velocity is strictly a translation (no rotational component) the integration to find the robot's position is easy. For the case when the velocity includes both a translation and a rotation component, it must be integrated to get a closed-form solution. The task frame expressed as a quaternion evolves according to equation (6-1).

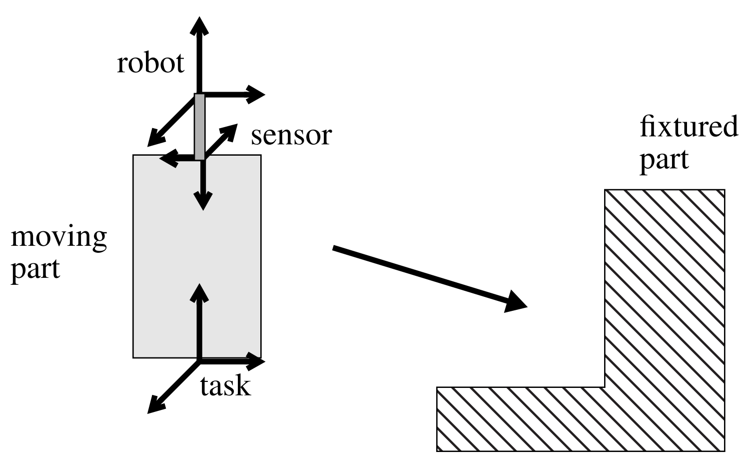


Figure 6-2: Task Model and Frames

$$q(t) = \left[\cos\left(\frac{\omega t}{2}\right), \sin\left(\frac{\omega t}{2}\right)\hat{n}\right] \tag{6-1}$$

The angular velocity remains constant throughout the segment as does $\hat{n}$. The robot linear velocity <u>in terms of the task frame at the beginning of the segment</u> evolves according to:

$$v(t) = q(t)v_0 q^*(t) \tag{6-2}$$

where v_0 is the constant linear velocity specified in the robot frame. Integrating (6-2) gives the equation for the position change of the robot frame as a function of the task velocity in the robot frame as:

$$\Delta x(t) = \frac{1}{\omega}\sin(\omega t) + \frac{1}{\omega}(1 - \cos(\omega t))(\hat{n} \times v_0) - \left(t - \frac{\sin(\omega t)}{\omega}\right)\hat{n}(\hat{n} \cdot v_0) \tag{6-3}$$

where Δx is the change in the robot's position in the <u>frame at the beginning of the velocity command</u>, $\hat{n}$ and ω are the direction and magnitude of the angular velocity, v is the linear velocity all in the robot frame, and t is the time in the current segment (not absolute time). The change in position can be transformed into world coordinates by using:

$$^{W}\Delta x = {}^{W}R_0 {}^{0}\Delta x \tag{6-4}$$

where $^{W}R_0$ is the rotation matrix of the robot frame relative to the world at the beginning of the motion segment.

Visual feedback can be simulated in two ways. First, the features themselves can be simulated through a pin-hole camera model and our world knowledge of the features. However, the scene can also be rendered onto a virtual image plane and used to produce video which is then processed by the image-processing system. In the first case, the video is not produced -- the simulation effectively replaces both the video and the visual tracking algorithms. The second case preserves the use of the visual tracking algorithms.

The SGI has the capability to output video from a window placed on the monitor this capability is used to generate ‘synthetic’ video. It is important to realize that although this video is from our simulated (perfect) world, the mapping is not perfect. First of all, the feature projections require the knowledge of the pixel size which is unknown for this virtual camera. Secondly, the Telegrip virtual camera possesses the same field-of-view in both the X and Y axes unlike the real camera which has a non-unity aspect ratio. Third, the output video window of the SGI is fixed size. To use ‘synthetic’ video, the Telegrip CAD window has to be carefully sized to fit the video output window. Then, because of the different aspect ratios, the output window has to be adjusted to cutoff the top and bottom part of the CAD camera view. All of this introduces uncertainty when using the simulated video. This is apparent because the projected locations of the features (which are known perfectly in the CAD world) clearly have some error when projected onto the virtual video. Because of this, a simple feature acquisition algorithm was built to search a 30x30 pixel window around the initial feature location estimate to lock onto the feature. This algorithm uses a 7x7 template to find a corner of consistent orientation to the feature. It is not very robust as it can be distracted by strong edges which are not corners. One of the primary needs of future work is to improve the feature acquisition (and tracking) algorithms to increase their robustness.

Telegrip provides the main user interface for the system. Telegrip also sends primitive commands and parameters through ethernet to the Chimera system. And Telegrip controls the overall flow by controlling the Coriolis simulation (i.e. beginning, advancing, stopping). Geometry created in Telegrip is exported to build Coriolis models. The Coriolis models receive input from the Chimera system through running actual primitive modules (the same ones which run the robot). The Coriolis simulation outputs force and position information to the Chimera process for use by the primitives and outputs position information on the moving part to Telegrip for display updating. The Chimera system provides velocity updates (based on the primitives) to Coriolis and also forwards events to Coriolis (to be passed onto Telegrip). Forwarding events is important since they represent the break-points in the strategy and a skill state cannot be exited unless an event occurs. Since simulation time is slower than real-time, the Coriolis and Chimera systems must be synchronized. This has been achieved by controlling the timer interrupts on the Chimera system and is described more fully in the Appendix. The result is that primitives can run in simulation or the real system with no changes to source code. Having only one executable removes the mismatch prob-

Figure 6-3: CAD Environment

lems that multiple versions (for simulation and ‘real’ system) introduce.

6.3 CAD-based Skill Composition

An overview of the C^2T system was presented in the previous section. This section describes how a skill is composed using the system. The application programmer builds the skill up interactively inside the CAD environment. To build the state machine requires specifying the individual states and connecting them with appropriate events. Skill states are defined by a list of commands which are executed upon entry to the state along with a list of events which transition to the next state(s). Beginning with a CAD model of the parts involved, the user indicates an intermediate step in the mating strategy by moving a ‘ghost’ model of the moving part. Currently the gripper is not rendered in the CAD environment. To execute this step on the real task, a primitive must be selected and parameterized. After selection and parameterization, the primitive is executed on the real-time system to drive the Coriolis simulation. The simulated sensor feedback is input to the Chimera system. The user can observe the parts move and continues the simulation until a Chimera event occurs. When the event occurs, the user can evaluate the task configuration and decide whether or not to accept the outcome (and primitive) for this step. During the simulation, data is logged into a file of state variables of the user's choosing. When the primitive completes, the user may view these variables in a graphing program to further evaluate the success or failure of the primitive. If accepted, the primitive command and parameters are sent to the FSM Editor to instantiate a new step in the strategy. If rejected, the task state is reset so the user can enter a new task configuration goal, select a new primitive, or modify the current primitive parameters. This procedure continues until the task strategy is complete. The user then saves the skill from the FSM Editor.

Design agents specific to each primitive can contribute to the skill building process in three different phases: selection, parameterization, and execution/simulation. During selection, each primitive's *selection* agent computes a scalar measure of its applicability to achieving the indicated goal by computing a scalar value between -1 (not applicable) and 1 (highly applicable). A score of zero indicates neutrality (the selection agent may not be able to determine the applicability). The user arbitrates between the choices and selects one primitive to invoke (Figure 6-4). Its *parameterization* design agent then extracts the necessary information from the task configuration for run-time execution. For example, a cartesian move primitive would extract the direction and magnitude of motion. Some information may not be evident from the geometric model -- for example, the force threshold for a guarded move. In this case, a default value is provided which is often reasonable in the domain. The user may override this automatically-generated information and explicitly indicates his acceptance of the final parameter set (Figure 6-5). The automatic extraction of primitive parameterization information reduces the burden on the application programmer. After parameterization, the primitive is executed by the real-time system and drives the task simulation. During this simulation, execution agents run to evaluate performance. For exam-

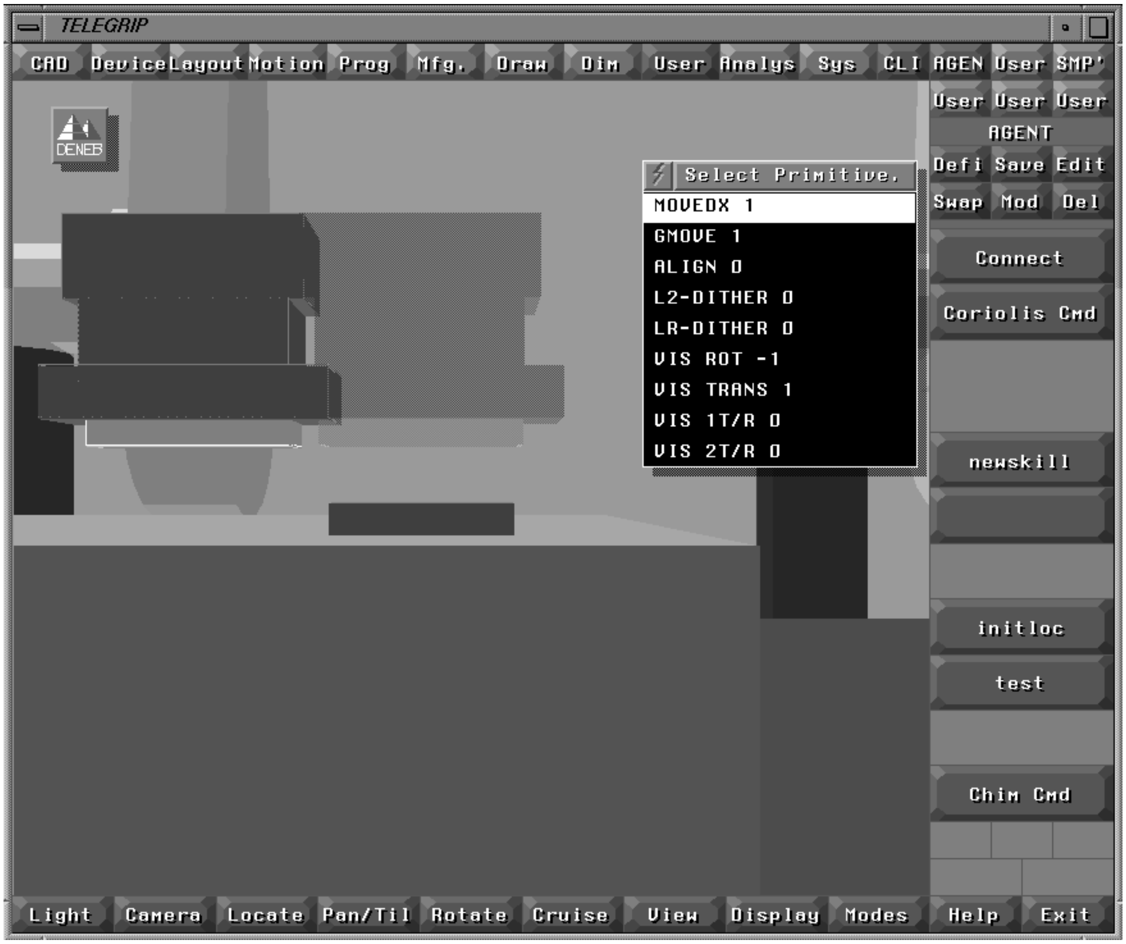


Figure 6-4: Primitive Selection

ple, visual servoing primitives rely on the visibility of the feature during the task motion. A visibility metric can be implemented in an execution agent which queries the CAD model state and camera location, and a warning can be issued should the feature visibility decline below an acceptable level. This can help the user identify problems in the skill strategy during skill composition rather than during real execution.

The result of this construction process is a linear sequence of primitive steps with specific event transitions. Ideally, the strategy would execute this way all the time! Unfortunately, the real world is not so kind. The state-machine itself represents the *skill*, not the task. Differences in task configurations can lead to different outcomes for a particular skill step -- these differences may be different events or they may be different task configurations which generate the same event but cause failure later in the strategy. The two possibilities correspond to an ‘immediate’ failure versus a deferred failure. In either case, the program

Figure 6-5: Parameter Selection

sequence has to be expanded into a 'web' in which different task outcomes are handled. One method to achieve this is to generate task configuration perturbations at different steps and execute the skill to see if either different events or skill failure can be generated. Manually, this would be very tedious but it could be automated by using, for example, Monte Carlo simulation. These cases can be saved and brought to the attention of the user for resolution.

One of the possible jobs of an execution agent is to construct an expected sensor trace for the strategy step during simulation. This can serve as a baseline for error detection during the actual strategy. Of course, executing multiple simulation trials allows this expected trace to be more fully characterized. The trace should probably be expressed qualitatively, perhaps in terms of linguistic or fuzzy variables, rather than quantitatively because of the uncertain task parameters. The goal is to give the primitive a rough idea of the expected sensor values so that significant deviations can be used to flag errors or at least conditions which should be investigated. This is an example of incremental program expansion -- rather than trying to predict and compose responses to all contingencies up front, quickly construct the linear, base program and then rely on simulations of skill performance to detect the abnormal execution conditions and problem areas in the strategy.

6.4 Design Agents

An SMP design agent is a representation in the CAD world of the primitive capability with a particular resource set capability. The motor resource capability is assumed to be fairly generic -- straight-line motion capability and enough workspace to perform the task. However, explicit assumptions are made about the available sensors. This thesis considers only force and vision, but clearly other sensors are germane: tactile and range-sensing are perhaps the most obvious. Since the primitives are sensor-based, the sensors must be modelled in the CAD/simulation environment as well. Deriving from those sensor model objects, the primitive design agents are instantiated as objects with data and a set of methods. The purpose of the design agent is to support the user’s selection, parameterization, and evaluation of a particular primitive application to a task strategy. Of course, the ultimate goal is to completely automate the selection, parameterization, and evaluation of the primitives where possible. But because of the extreme complexity of fully modelling these capabilities, usually some interaction with the human will be necessary. This approach leverages the strengths of man and machine by allowing the insertion of powerful algorithms where they exist along with the ability of the human to guide the process, override agent suggestions, and generally arbitrate between competing agent solutions. The framework is easily extensible by adding additional sensors, primitives, and more sophisticated algorithms.

This section provides some example design agents for primitives introduced earlier. The primitives can be grouped according to whether they use force feedback, vision feedback, or no feedback (open-loop). The Cartesian differential movedx is an example of an open-loop primitive. The guarded move (gmove), compliant move (accommodation), L2 dither, LR dither, and align are examples of force-based primitives. Vis_acc and vis_aab are examples of vision-based primitives.

6.4.1 A Simple Example: movedx

The movedx primitive supports a straight-line differential motion in either pure translation or pure rotation. The selection agent matches the initial, goal positions to this motion template. In addition, the selection agent looks at the minimum distance between the parts involved -- if it gets very small, then movedx disqualifies itself in naive anticipation of contact. The parameterization follows from the selection computation with additional information needed for speed -- a default one second move duration is used provided it does not exceed the allowable maximum speed. The evaluation agent monitors the minimum distance to the part during the move to be sure it does not fall below an acceptable threshold. Other functions for the evaluation agent could be to monitor the executed path for collisions -- although the run-time system will be monitoring for excessive force. The movedx algorithm is similar to current robot programming primitives -- open-loop. Next, visual feedback primitives are discussed which allow positioning moves to be task-driven by defining them on the image plane.

6.4.2 Vision Primitive Design Agents

The vis_acc and vis_aac primitives involve specifying two or one position goals on the image plane for controlling a corner; vis_acc is considered since it subsumes vis_aac. All visual selection algorithms key on anticipated projections of 'key' task features. The thesis has focused on corner feature projections on the image plane. The current implementation requires that the operator indicate pairs of task features (defined on the parts) to monitor as part of the task model. The simplest example of this type of feature is a 2D corner defined by 3 points on the part. This could be a fiducial mark or related to actual task edge features. An actual (3D) corner in the task can be defined by using 4 points. Pairs are provided to be computationally efficient, but one could imagine having algorithms which examine the geometry of each part to extract candidate features and form the cross product of the feature sets for each part. In the author's opinion, the feature pairs idea is more attractive and it fits in with feature-based CAD which is discussed later in this chapter.

Before computing selection agents, the set of feature pairs for the task must be 'imaged' for the initial and final task positions -- that is, projected onto the virtual image plane. From this imaging, the corner projections on the virtual camera are determined for each feature pair. Feature robustness criteria like requiring a minimum length projection of each corner edge are also applied. When the feature projections are computed, the basic errors associated with initial/final feature pairs are computed and stored -- these include the x,y translation error and four measures of rotation error (one for each edge combination).

Armed with a list of visible feature pairs, the selection agents can compute their applicability indices. The visual translation primitive selection agent is interested in whether significant translation occurred between the initial and final positions accompanied by little rotation. The vis_acc selection primitive looks at x,y errors between two features in a pair along with minimal angular error. The current selection agent implements a crisp function returning -1 or +1. More sophisticated algorithms could return a confidence value (e.g a stochastic or fuzzy measure) of the applicability.

$$S = (x_e > \varepsilon_x) \wedge (y_e > \varepsilon_y) \wedge (\theta_e < \varepsilon_\theta) \tag{6-5}$$

The parameters of the visual translation primitive include: the camera pose relative to the task frame at the beginning of the visual motion, the estimated initial locations of the features in the image for acquisition purposes, the nature of the corner foregrounds (inside/outside), and the goal position referenced to the non-moving part feature (xe,ye). The camera pose is important since it defines the image jacobian for the control law -- it defines the expected location of the camera to the task. The estimated corner features are necessary to invoke the primitive autonomously -- i.e. without a human manually selecting the features. The last parameter is important since the motion will often not involve actually bringing a feature error to zero. Thus, the xe,ye parameters refer to an acceptable error offset (Figure 6-6). The key point is that the goal location has been defined explicitly by this offset relative to the fixed part. Thus, the amount of initial placement uncertainty which can be resolved is largely driven by the robustness of the feature acquisition algorithm. The current acquisition algorithm is very simple -- a more robust algorithm would use more CAD information to

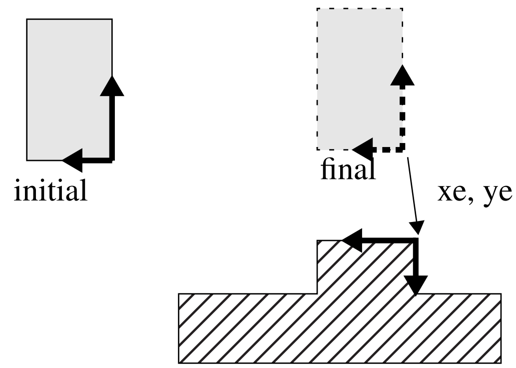


Figure 6-6: Translation Offset Definition

provide a robust acquisition of the feature. For example, other visual features project onto the image (e.g. lines) -- if this information were also exported, the acquisition algorithm could use more constraints to acquire the feature. The idea is to make the acquisition algorithm general and instantiate the specific parameters it needs from the CAD environment. Thus, even if that part location is slightly different during execution, the primitive will still succeed. Significant camera pose changes will change the relationship of the offset error to the fixed part. This is one advantage of using zero offset errors is that they are insensitive to camera pose while larger offset errors are more sensitive. Fortunately, larger offset errors are usually tolerant of these errors since they often define approach directions.

6.4.2.1 vis_aab

The selection agent implements nearly the complement function of vis_acc for selection because it applies when rotation is significant but translation is not. However, the allowable translation is reasonably large to account for induced translation of the feature under rotation.

$$S = (x_e < \delta_x) \wedge (y_e < \delta_y) \wedge (\theta_e > \varepsilon_\theta) \tag{6-6}$$

The parameters are similar to the visual translation primitive: camera pose relative to the task, initial feature locations in the image, feature foregrounds, the edge pair defining the rotation error and the rotation error offset. The edge pair is selected based on edges lining up (error of 0 or 180). This takes more task knowledge since there is no guarantee that the perceived edges are actually the ones which the user is referencing. The edge selection algorithm also favors edges which are longer -- these give more accurate orientation information. One more parameter is possible and that is the task axis to project the rotation vector onto. The rotation is naturally defined along the optical axis. However, due to other controllers (or previous knowledge), only one rotation DOF may be free and the camera may have an oblique view of it. An optical axis velocity vector will introduce disturbances to the other rotation DOF. Projecting the optical axis rotational velocity onto the estimated desired rotation axis will mitigate the disturbances.

6.4.2.2 vis_bcc

Controlling both rotation and translation simultaneously may require a different control law because they are highly coupled. The selection criteria are similar to the above except they recognize the rotation. The parameters are similar with respect to locating features. However, a choice of control laws exist. The simplest is a superposition of the vis_acc and vis_aab control laws. With the proper starting location, the superposition will work fine. But the corners may easily begin in a configuration that will lead to occlusion if straight-line motion is used to align the corners. The chances of this happening increase as the allowable uncertainty increases. A non-linear control law may be designed by constructing a non-symmetric attractor around the goal corner projection on the image plane.

6.4.3 Force Primitive Design Agents

6.4.3.1 Guarded move

The selection algorithm looks for a straight-line move along with a decrease in the minimum distance between the two parts. The guarded move primitive should enable when the movedx primitive disables. The key part of the selection algorithm is to match the directionality of the motion with the directionality of the minimum distance decrease. This will prevent the primitive from enabling when the motion is parallel to a near-contact constraint. The motion parameters (direction and magnitude) follow from the selection computation. In addition, default values of contact force (~1N) and approach speed (~1 cm/s) are suggested -- their correctness is strongly dependent on the task model. Task mechanics can be used to set these parameters as well. The evaluation agent can monitor the impact forces and suggest a lower approach velocity if excessive impact forces result. The guarded move is relatively easy to select and parameterize because 1) it is a relatively straight-forward primitive, and 2) the availability in the CAD environment of a function reporting the minimum distance between two parts.

Generally, automated selection and parameterization force-driven primitives requires the assessment of the contact states between the initial and final task configurations. With the current method of input involving only initial and final task states, the constraint directions which are the same will be assumed to remain the same throughout the move. This thesis does not consider the analysis of these contact states for selection and parameterization of force-based primitives. Some relevant approaches might be the polyhedral convex cone contacts analysis of Asada and Hirai [5] in which instantaneous motion constraints can be determined from a geometric model (assuming rigid objects). The automatic recognition of force-based primitives may be better performed with a richer set of input data. For example, Voyles et al [74] apply principal components analysis technique to recognize guarded moves and align moves from teleoperated demonstration. This is not applicable to our system since the demonstration phase involves only indicating a task state change and does not involve a continuous task motion. In light of the difficulty of integrating automatic selection and parameterization agents for the force-driven primitives, we have provided input dialogs for the user to easily enter the information. This is still very useful since it automates the parameter formatting which is unique to each primitive. The ability to replay the move in simulation allows the user to iterate to arrive at a good parameter set. The framework easily allows the addition of automated selection and parameterization algorithms.

6.4.3.2 L2 and LR Dithers

These dither combinations are particularly appropriate for acquiring motion constraints in the plane through coordinated sinusoidal dithers plus correlation functions. The LR primitive, for example, has the following parameters: the axis of rotational dithering (3), the magnitude of rotation, frequency of rotation, number of cycles, rotary correlation threshold, the axis of linear dithering and linear correlation threshold. While it would be preferable to set these parameters automatically, it is still very useful to have simple parameter entry dialog box along with the task model (with task axes shown) to help the designer set them. In the absence of an algorithm to indicate the selection index, a value of 0 is returned indicating neutrality.

6.5 Sensor-based Simulation Results

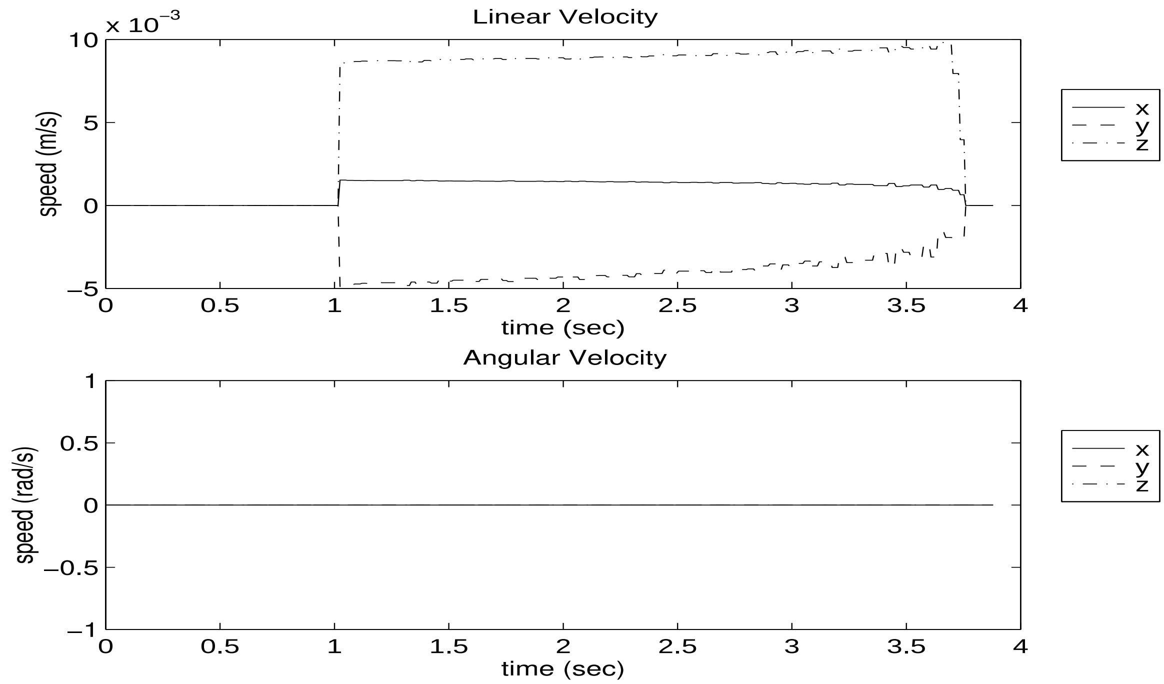


Figure 6-7: Visual Move in Simulation

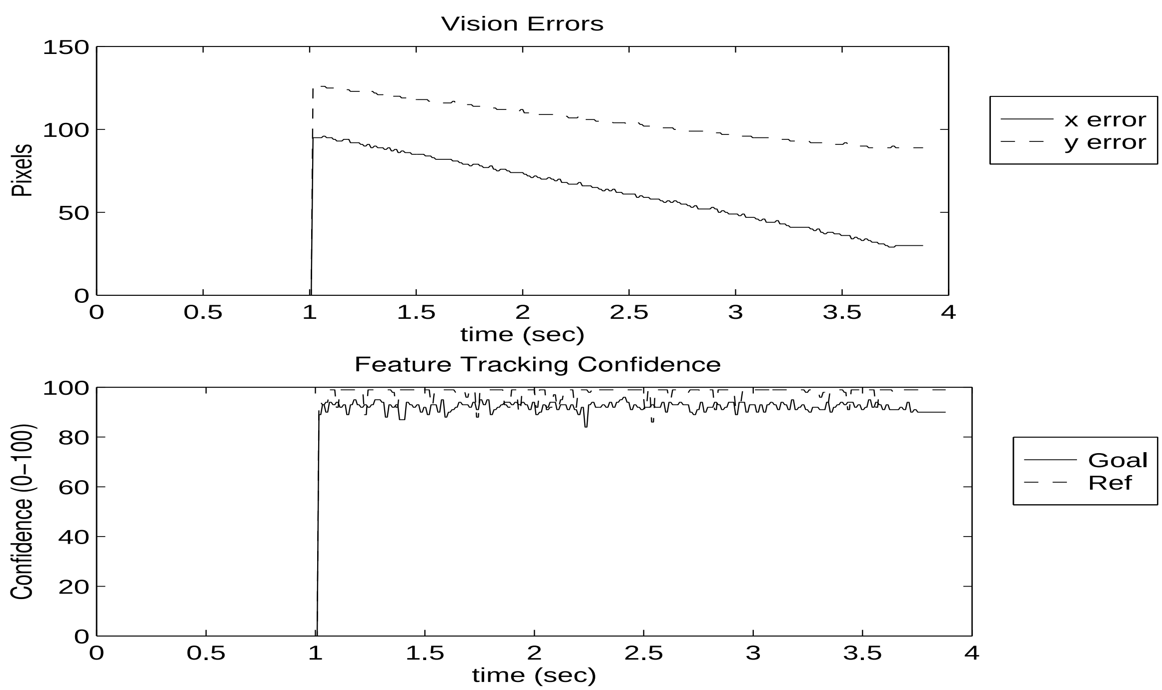


Figure 6-8: Visual Move in Simulation

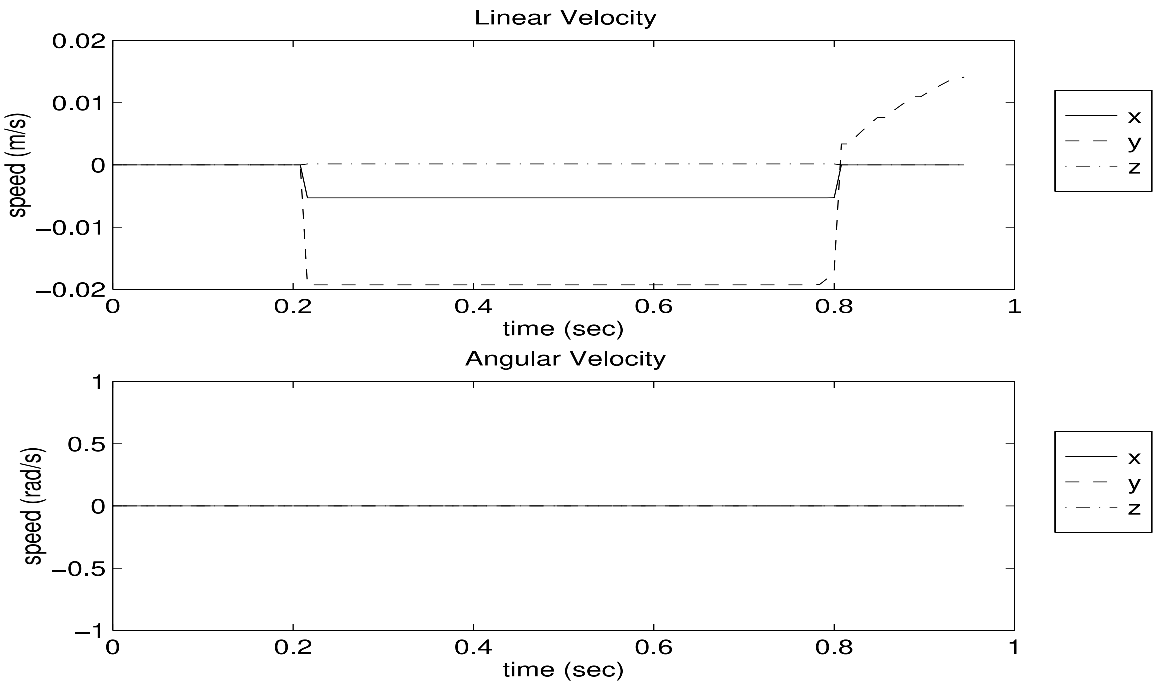


Figure 6-9: Guarded Move in Simulation

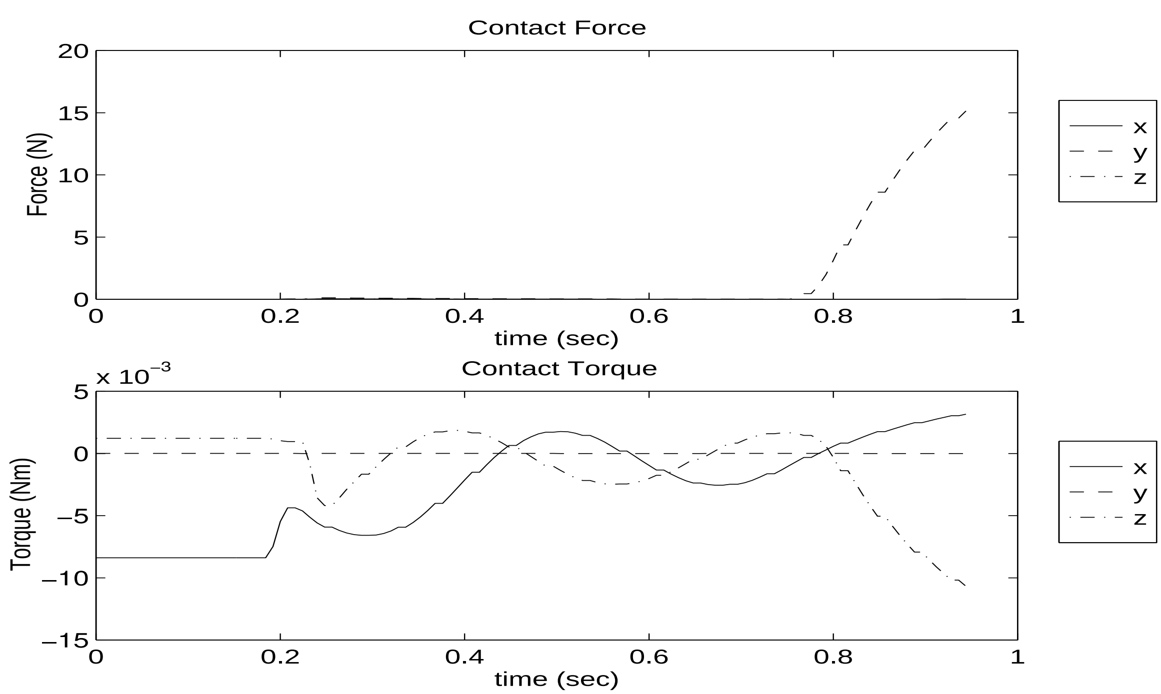


Figure 6-10: Guarded Move in Simulation

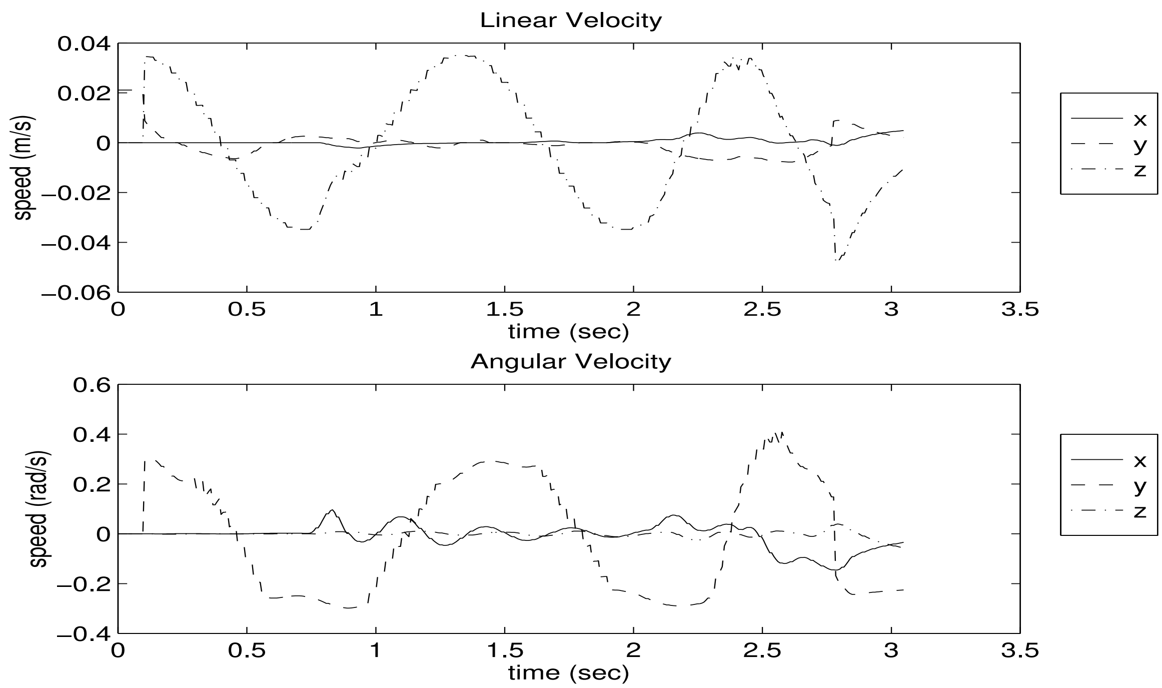


Figure 6-11: Dither Move in Simulation

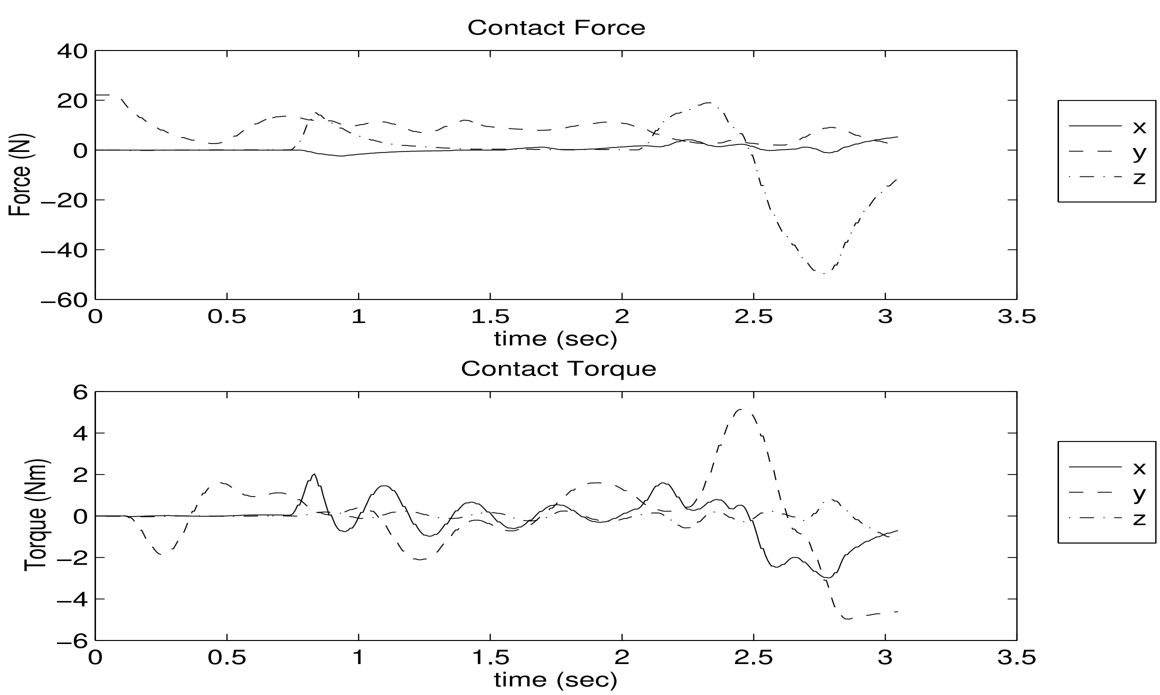


Figure 6-12: Dither Move in Simulation

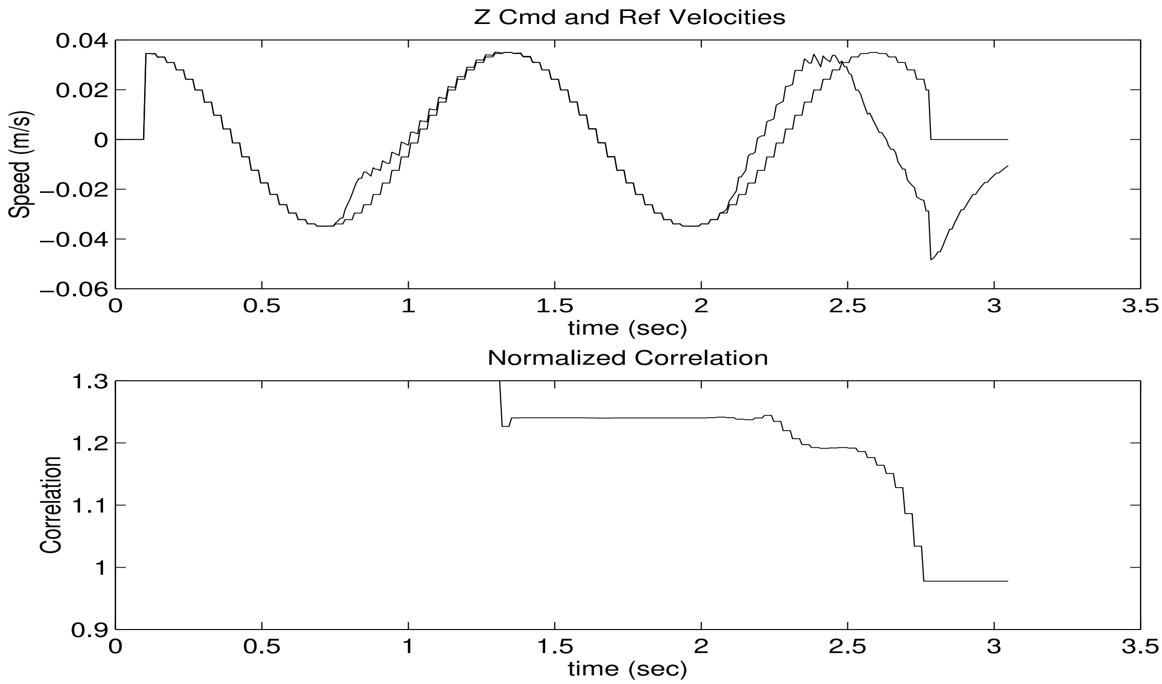


Figure 6-13: Dither Move in Simulation

The simulation results show that vision and force feedback produced in the CAD/simulation environment are used by the sensorimotor primitives. The force signals are much cleaner than real force signals, but stochastic noise can easily be added to the simulation. During visual moves, the feature tracking confidence is quite good but not perfect because the feature tracking algorithms are operating on the synthetic video image of the CAD world as seen through a camera placed in the world. The guarded move and dithering/correlation primitives are executed as examples of force-based primitives. The event detection for each works in simulation as it does in the real world.

6.6 Interactive Software Components

Design agents were introduced to support primitive design integration. This is a simple implementation of a more powerful concept of *interactive software components*. The idea is that a software component has not only the *execution* component, but a *design* component which interacts with the software designer during the programming phase. This idea extends the programming primitives from being passive components which the user must select to 'active' components which have some understanding of their capabilities, can interact with an explicit task model, and make suggestions to the software developer. Interactive software components transform software development into a *collaborative* effort between the software designer and the software components themselves. The key to this approach is the presence of an explicit task model of the programming problem to which both the user and the software components have access. The task model provides: 1) an explicit, though approximate, representation of the task, 2) an intuitive view of the representation for the programmer, 3) an intuitive interface for specifying desired task state changes, 4) the ability to fully execute the software components, 5) a source of information to the software components which they use to evaluate their selection, parameterization, and execution criteria.

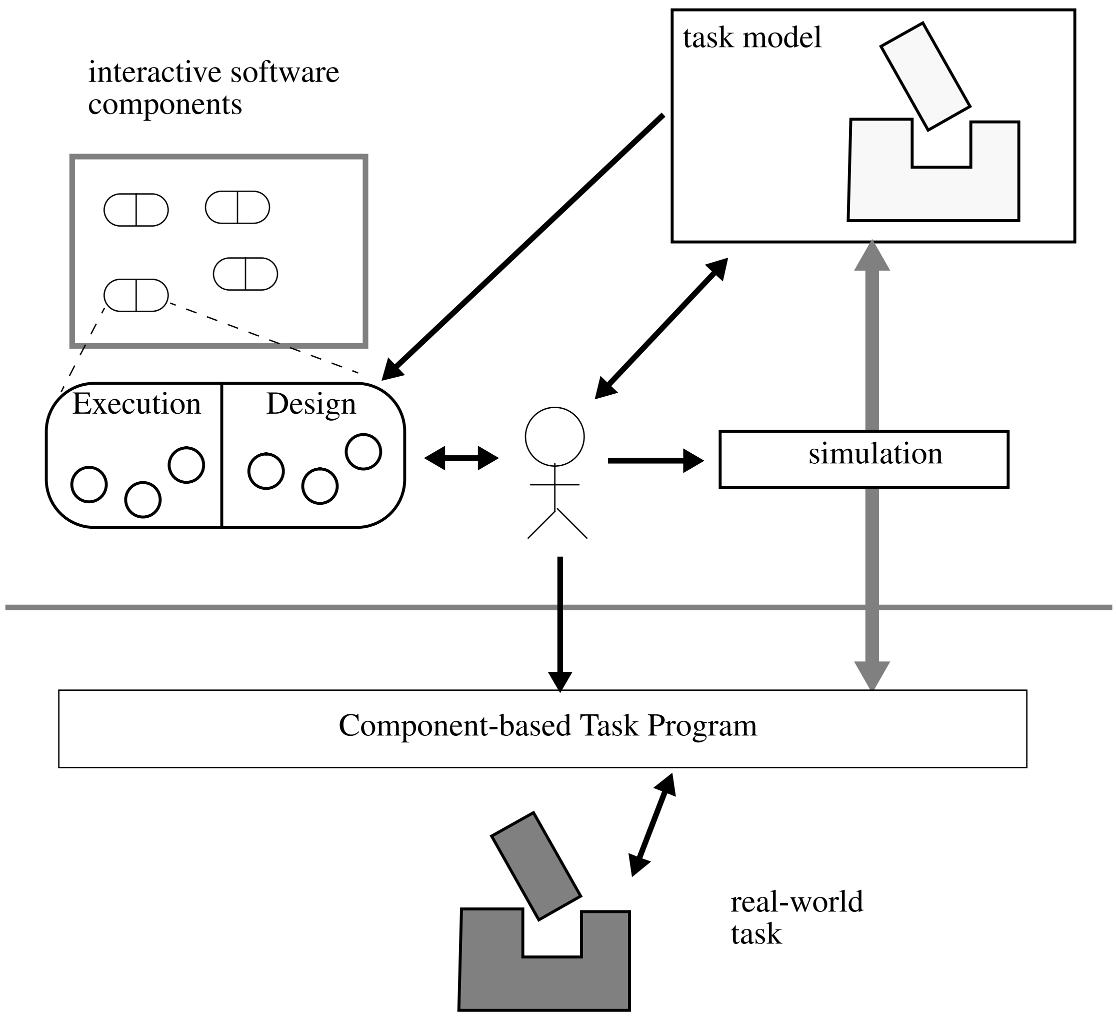


Figure 6-14: Interactive Software Components

6.7 Summary

The CAD/simulation integration of the sensorimotor primitives was described in this chapter. To focus on modelling the task mechanics, the robot is abstracted away to a kinematic frame connected to the force sensor. The force sensor is modelled as a stiff spring/damper and the visual feedback is taken from a virtual camera view in the CAD world using the SGI output video for our Datacube vision processing system. The actual visual feature tracking algorithms running on the Datacube vision processing hardware are used to drive the visual feedback primitives. Force-based primitive execution was demonstrated as well, showing that the simulation produces forces on contact which are used by the Chimera force control algorithms drive the simulation. Simulation results included a demonstration of a visual servoing move, a guarded move, and a dithering/correlation primitive move all run with simulated sensor feedback. Because of the stiffness of the dynamic system equations, the simulation speed is much slower than real-time (estimated ~1:20 worst case). Fortunately, this problem will go away with faster computing.

To support the exploitation of primitives in the design environment, the concept of Interactive Software Components (ISC) was introduced. ISC's include both an execution and a design component to help the user deploy the software, and transform the software development process to a collaboration between the software components and the user. As examples of simple ISC's, several 'design agents' were discussed which help the user select and parameterize the sensorimotor primitives inside the CAD environment. A couple of examples were given for visual feedback primitives which are naturally (and fully) defined by the initial/final task configuration input. The difficulty of automating the selection and parameterization of force-based primitives beyond the simple guarded move was outlined. A couple of approaches which might apply to automating the selection and parameterization of force-based primitives including PCC analysis and principal components analysis were mentioned.

Chapter 7

Conclusions

7.1 Summary

The goal of this project was facilitating the construction of sensor-based robot skills by simultaneously increasing sensor integration and reducing programming complexity. More general sensor integration was driven by a relative motion classification defining the types of possible relative motion between two rigid parts. Although the motion constraints are expressed in terms of part frame motion, they are defined in terms of specific task features. We reviewed how specific task features enforce those constraints and related those task features to their projection on force and vision sensor spaces so that task-driven execution primitives could be implemented. Wrist force sensing and visual servoing were used separately and in some cases together to implement algorithms for implementing or observing these motion constraints.

A skill was defined as a finite-state machine built out of these primitives. This representation allows use to reuse portions of the skill (primitives) which apply to different, but related tasks. This represents one of the chief differences in our approach compared with other researchers who typically use some form of 'black-box' to represent the skill. The Chimera reconfigurable software framework was extended to a 4th level which processes events and data to implement a meta-control level over the SBS module (3rd) level. The run-time system was built to support the Agent level along with an off-line graphical programming environment to facilitate the rapid construction of sensor-based skills.

Because the sensorimotor primitive set depends on specific task features, the primitives were integrated into a CAD design environment to facilitate concurrent design in which the available primitive set could be exploited during task design. Instead of considering the task as "given" and trying to build a skill for it, the environment allows the skill to be co-designed with the task. For tasks involving fine motion this is very important since poor design decisions can make skill strategy synthesis extremely difficult. This approach is also consistent with DFM/DFA approaches which state that manufacturing and assembly concerns should be largely solved during design.

Since sensorimotor primitives use sensor feedback, the integrated system tightly couples the real-time system with a task CAD/mechanics model which includes force and vision sensor simulations. The skill is constructed by manipulating the task model with the assistance of design agent representations of the primitives. The design agents are simple implementations of a more interesting concept: *Interactive Software Components*. The basic idea is applicable to programming in domains like robotics, where the task strategy requires significant translation to express in the execution programming language. Using an explicit model of the task to be programmed, the user induces desired task state changes and the software primitives which are the elements of the program interact with the user and the task model to select and parameterize themselves for the step. This is different from automatic planning in that the human is the central decision-maker and arbiter. The role of the software "design" components is to assist the human in building the software.

The human arbitrates between different primitives by selecting one. He also may override the parameters which the primitive's design agent suggests. The primitive then executes to update the task state via simulation and the user may verify the proper task state change.

Realizing this system required a large integration effort involving a 3D CAD environment, a rigid-body simulation package, and the real-time system. An example CAD-based skill program was presented to demonstrate the sensor-based nature of the simulation control during the task update. One of the significant advantages of the integration is that the actual code which drives the robot also drives the simulation. There is no danger of mismatch between the robot version and simulation version of code since they are the same. This also lets the CAD integration system be used as a (safe) development tool for additional primitives. The explicit representation of both the task and the strategy let the programmer consider modifications in either to generate a robust solution.

Finally, the ideas were experimentally demonstrated. Six different assembly tasks requiring significant contact were performed to demonstrate the construction of sensor-based assembly skills built from common sensorimotor primitives. A CAD-built skill provided simulation results to demonstrate that both force and vision feedback are used inside the CAD integration system to update the task state. The concept of design agents which support the primitive use inside the design environment was demonstrated as well.

7.2 Development Extensions

This section discusses a number of the system short-comings which can be fixed through additional engineering development -- evolutionary improvements. The next section discusses “revolutionary” improvements -- future research. The virtual demonstration interface needs to be improved to allow fine-motion tasks to be adequately specified. Currently the CAD demonstration interface allows only one size of step increment -- this can easily be extended to a user-specifiable level to allow both “large” and “small” steps to be specified.

Next, the CAD-generated program is a *linear* program-building process -- that is, the resulting program is a simple sequence, not a web or tree. But robot programs need loops and non-linear flow which are not easily specified with the current CAD version, though the state machine implementation clearly supports them. A first step is to allow the user to accept a primitive outcome without advancing the task state inside the CAD environment. This way perturbations to the task state could be used to generate a different event for the command. The basic approach is to introduce task state perturbations to discover different outcomes of the same primitive step. Another improvement is to allow loading and executing of completed skills inside the CAD environment. The current system supports simulation of a single skill step as part of building a skill. The tools exist for simulating a complete skill, but their integration must be completed.

Visual feature tracking needs to be improved in a number of ways. Clearly a larger set of visual features must be considered for a real system including edges, regions, and ellipses. Espiau et al [19] present different features in their *task-function* approach to visual servoing which focuses on eye-in-hand formulation. Improved robustness is needed to apply vision-based primitives to real tasks. The simple corner tracker allowed tracking of occluding boundaries but additional constraints are necessary for improved robustness. Fiducials provide robustness but are not "natural" task features and increase the sensitivity to camera pose errors between programming and execution. The problem is that developing robust feature trackers is very difficult and most robust trackers use a great deal of model-based information to provide constraints so that they are not easily distracted. This introduces two problems: 1) increased computational effort slows down tracking (which makes control more difficult), and 2) the trackers become very specific to a particular task model. Since clearly more model information is needed to improve tracking robustness, the question is how to balance the feature generality with computation speed. What is needed is to efficiently incorporate model-based information to improve tracking robustness without making the feature tracker development a task-specific ordeal. Exploiting the CAD environment's ability to render the scene realistically will be important for investigating the photometric effects on feature tracking algorithms. Surface properties, colors, and lighting all have a tremendous effect on the robustness of different feature tracking algorithms and the ability to test the robustness of algorithms by perturbing these parameters will help designers generate robust task/skill designs which rely on visual feedback.

The Agent level is the first comprehensive attempt to extend the Chimera execution layer above the 3rd level. One way it could be improved is by providing a deeper definition of a FSM state. For example, allowing command lists to be attached to different events would be useful for cleaning up the state before moving onto the next one. Also allowing the FSM to do simple logical operations on events -- for example to force a transition when two events occur (an “AND” function) -- would make it more powerful as well. Adding hierarchy capability in the state machine would be very useful to combat one of the main problems with using state-machines for real-problems -- the proliferation of states. Hierarchies allow one to hide complexity in the state machine. Although the state machine model is a good one for representing the segmented nature of complex skills, it hides all the real-time aspects of the implementation. While this is attractive for the application programmer, the control system engineer or real-time systems engineer may want this view. Since there are multiple agents usually executing in parallel, it would be interesting to construct a graphical view of the various agents with their states as parallel lines or tracks. Finally, it became very clear after using the state-machine editor that parameter entry dialog boxes are really important for efficient primitive use. Without parameter entry boxes, it is very difficult to remember the primitive formatting which is unique for each skill.

Finally, were I to continue this work, I would abandon Telegrip as the CAD interface. The skill-programming system does not exploit its robot-simulation capabilities and is hampered by the lack of access to primitive graphics functions like selecting graphic elements (lines, points, etc.) of the model. Instead, I would explore using a graphics package like OpenGL or Open Inventor which may better support graphics-based parameter specification.

7.3 Future Research

7.3.1 Task Uncertainty

The current CAD integration is focused on developing the ‘nominal’ strategy, but error recovery is an important part of any robot program. Related to this is a more explicit representation of uncertainty in the CAD environment. Uncertainty may be better expressed from the primitive's point of view than being arbitrarily assigned to the task. In fact, the bounds of resolvable uncertainty might be a very useful way to arbitrate between different primitives competing for a task step. Building in semi-automated methods for extending the state-machine are necessary. Algorithmic methods are preferable, but they might not always be possible. For investigating robustness, both *task and strategy parameters* may be modified to detect strategy failures or performance degradation. As a first step, Monte Carlo simulation could be incorporated to detect problems and bring them to the attention of the programmer/ designer for resolution.

7.3.2 Parameter Adaptation and Optimization

The parameters in the skill are fairly arbitrarily set by the programmer during the iterative skill-building process and are not optimized. A key extension is to try to optimize the parameters to maximize robustness or performance (or some combined objective function which includes both). The best parameter set (and even the best skill strategy) will be a strong function of the specific objective function selected. In addition, since the strategy and task can be co-developed, the optimization can apply to not just the strategy (or program), but to the task design features as well. Clearly, the parameter search space becomes very large when including both the primitive parameters and the task parameters so efficient heuristics to prune and direct the search are necessary.

7.3.3 Parametric (Feature-based) CAD

The idea of primitive design agents fits in nicely with state-of-the-art CAD systems which use parametric modelling techniques. In fact, the idea of defining assembly features which relate geometric primitives on different parts has been proposed [59] as a method of facilitating downstream processes like assembly sequence planning. This thesis focuses on the generation of an executable program from the CAD environment instead of an abstract assembly sequence. As discussed earlier, mating two parts involves constraining relative motion and these constraints are implemented by specific part features. Primitives would be associated with feature pairs belonging to different parts. By connecting these features in the CAD environment, we can effectively attach the execution capability to them. For example, one feature used in CAD programs is a hole -- when this feature is added to a part the hole "knows it's a hole." If the part is made wider, the hole lengthens to still go through. Imagine now defining a feature set of a hole and a peg -- their relative dimensions deliver important assembly information such as whether the fit is loose, tight, or a press-fit which helps in the selection of the proper execution primitive. Primitive design agents can also extract information such as required sensor placement relative to each feature so that the important attributes (to execution) are visible. The design environment may have "preferred" feature pairs from the point of view of executability since not all feature pairs might have execution support. This approach tightly integrates the execution (assembly/manufacturing) requirements with the design requirements.

7.3.4 Design for Sensor-based Assembly

The primary goal in designing for sensor-based assembly is satisfying the primitive constraints with respect to task feature projection onto the sensor space. The constraints fall into feature visibility, uniqueness, and applicability. Visibility is the most basic and requires at least that the geometric projection of the feature fall in the sensor's field-of-view. Visibility also encompasses ideas about feature quality (is it large enough?). Visibility must be satisfied through proper sensor placement relative to each part as well as specific assumptions about the environment (e.g. light source direction). Uniqueness requires that the feature is identifiable and trackable which imposes specific environmental demands like adequate lighting and limited distractions. Finally, the features must be *applicable* to the task motion -- that is, they must measure task information which is usable for controlling the task. The trackability issues of visibility and uniqueness typically dominate. Fiducials are often used because all three can be adequately addressed but it requires engineering the task in ways which are not always feasible. "Natural" features are preferable for applicability, but can be difficult to track.

7.3.5 Interactive Software Components

Finally, Interactive Software Components have been barely touched and should be more fully explored. ISC's are related to gesture-based programming by demonstration which is a 'natural' method for humans to program by demonstrating the task. The idea is to capture the programmer's intent (which in robotics is a particular world state change) and robustly execute it later using the robot. Opportunities exist to improve and enrich the user interface -- we have used a very simple user demonstration method. Integrating touch/tactile feedback and virtual reality technologies would make the source of demonstration data much richer. Formalizing the interaction between the human, primitives, and task model is important. The design component of the primitives must be able to recognize when the human is trying to perform that primitive by observing the human's actions as well as the task state change. Voyles et al [74] present some preliminary results from applying principal components analysis to primitive recognition for simple primitives. As the number of primitives grows, the user interface will have to evolve to efficiently manage them. With ISC's, software development becomes a collaborative effort between the human and the software components, balancing the strengths of each. As more autonomous software components are developed, they can be incrementally added. This approach allows a useful programming environment for developing software with complex primitives which require significant parameterization and the environment can evolve in capability as the software components themselves evolve.

Chapter 8

References

[1] N. Ahlbehrendt, H.-J. Horch, and A. Tentschew, “Techniques of Teaching Skills to Sensor Guided Robots,” *IFAC Skill Based Automated Production*, pp. 203-208, Vienna, Austria, 1989.

[2] D.S. Ahn, H.S. Cho, K. Ide, F. Miyazaki, and S. Arimoto, “Strategy Generation and Skill Acquisition for Automated Robotic Assembly Task,” pp. 128-133, *Proceedings of the IEEE International Symposium on Intelligent Control,* Arlington, VA, 13-15 August, 1991.

[3] P. Anandan, “Measuring visual motion from image sequences,” Tech. Rept. COINS-TR-87-21, Amherst, Mass., University of Massachusetts COINS Department.

[4] H. Asada, “Representation and Learning of Nonlinear Compliance Using Neural Nets,” *IEEE Transactions on Robotics and Automation*, pp. 863-867, Vol. 9, No. 6, December 1993.

[5] H. Asada and S. Hirai, “Towards a Symbolic-level Force Feedback: Recognition of Assembly Process States”, in Robotics Research, Fifth International Symposium, Tokyo, Japan, 28-31 Aug, 1989.

[6] D. Baraff, “Interactive Simulation of Solid Rigid Bodies,” in IEEE Computer Graphics and Applications, v. 15, no. 3, pp. 63-75, 1995.

[7] A. Blake and A.L. Yuille, *Active Vision*, MIT Press, 1992.

[8] R.W. Brockett, "On the computer control of movement," in the *Proceedings of IEEE Internationall Conference on Robotics and Automation*, Philadelphia, PA, pp. 534-540, 24-29 April, 1988.

[9] B. Brunner, K. Arbter, G. Hirzinger, and R. Koeppe, "Programming robots via learning by showing in a virtual environment," Virtual Reality World '95, Stuttgart, Feb 21-23, 1995.

[10] H. Bruyninckx and J. De Schutter, "Specification of Force-Controlled Actions in the "Task Frame Formalism" -- A Synthesis," IEEE Transactions on Robotics and Automation, vol. 12, no. 4, August 1996, pp. 581-589.

[11] M. Buss and H. Hashimoto, "Skill Acquisition System for the Intelligent Assisting System (IAS)," pp. 157-171, *Advanced Robotics*, Vol. 8, No. 2, 1994.

[12] J. Canny and K. Goldberg, "A RISC approach to robotics," *IEEE Robotics and Automation Magazine,* vol. 1, no. 1, March 1994, pp. 26-28.

[13] W.F. Carriker, "A Rapid Prototyping System for Flexible Assembly," Ph.D. Thesis, Electrical and Computer Engineering Dept., Carnegie Mellon University, Pittsburgh, PA, 1995.

[14] A. Castano and S. Hutchinson "Visual Compliance: Task-Directed Visual Servo Control," *IEEE Transactions on Robotics and Automation*, vol. 10, no. 3, pp. 334-342, June 1994.

[15] D.C. Deno, R.M. Murray, K.S.J. Pister, and S.S. Sastry, "Control Primitives for Robot Systems," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1866-1871, Cincinnati, OH, 1990.

[16] B. Donald, "Planning multi-step error detection and recovery strategies," *International Journal of Robotics Research,* vol. 9, no. 1, pp. 3-60, February 1990.

[17] S.D. Eppinger and W.P. Seering, "Understanding bandwidth limitations in robot force control," Proceedings of the 1987 IEEE Int'l Conf. on Robotics and Automation, Raleigh, NC, 31 March - 3 April, 1987.

[18] M. Erdmann, "Understanding Action and Sensing by Designing Action-Based Sensors," *International Journal of Robotics Research*, 14(5), 1995:483-509.

[19] B. Espiau, F. Chaumette, and P. Rives, "A New Approach to Visual Servoing in Robotics," *IEEE Transactions on Robotics and Automation*, Vol. 8, No. 3, pp. 313-326, June, 1992.

[20] J. Feddema, "Visual Servoing: A Technology in Search of An Application," Proc. of Workshop M-5: *Visual Servoing: Achievements, Applications, and Open Problems*, at *IEEE International Conference on Robotics and Automation*, San Diego, CA, May 1994.

[21] M.W. Gertz, "A Visual Programming Environment for Real-time Control Systems," Ph.D. Thesis, Electrical and Computer Engineering Dept., Carnegie Mellon University, Pittsburgh, PA, 1994.

[22] M.W. Gertz, D.B. Stewart and P.K. Khosla, "A Software Architecture-Based Human-Machine Interface for Reconfigurable Sensor-Based Control Systems," in *Proceedings of the 8th IEEE Symposium on Intelligent Control*, Chicago, IL, August 1993.

[23] V. Gullapalli, R. Grupen, and A. Barto, "Learning Reactive Admittance Control," in *Proceedings of IEEE International l Conference. on Robotics and Automation*, Nice, France, 1992.

[24] T. Hasegawa, T. Suchiro, and K. Takase, "A Model-Based Manipulation System with Skill-Based Manipulation," *IEEE Transactions on Robotics and Automation*, Vol 8, No. 5, pp. 535-544, October 1992.

[25] L.S. Haynes and G.H. Morris, "A Formal Approach to Specifying Assembly Operations," *Int. J. Mach. Tools Manufact.*, Vol. 28, No. 3, pp. 281-298, 1988.

[26] S.H. Hopkins , C.J. Bland, and M.H. Wu, "Force sensing as an aid to assembly," *Int. J. Prod. Res.*, vol. 29, no. 2, pp 293-301, 1991.

[27] S. Hutchinson, G.D. Hager, and P.I. Corke, "A Tutorial on Visual Servo Control," IEEE Transactions on Robotics and Automation, vol. 12, no. 5, October 1996, pp. 651-670.

[28] S.B. Kang, "Automatic Robot Instruction from Human Observation," Ph.D. Thesis, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, December, 1994.

[29] J.C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, Norwell, MA, 1991.

[30] S. Lee and H. Asada, "Assembly of Parts with Irregular Surfaces Using Active Force Sensing," pp. 2639-2644, *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, CA, 1994.

[31] S. Lee and M. H. Kim, "Learning Expert Systems for Robot Fine Motion Control," pp. 534-544, *Proceedings of the IEEE International Symposium on Intelligent Control*, Arlington, VA, 1988.

[32] S. Liu and H. Asada, "Transferring Manipulative Skills to Robots: Representation and Acquisition of Tool Manipulative Skills Using a Process Dynamics Model," *ASME Journal of Dynamic Systems, Measurement, and Control,* Vol. 114, No. 2, pp. 200-228, June 1992.

[33] T. Lozano-Perez, "Task Planning," Ch. 6, *Robot Motion: Planning and Control*, MIT Press, Cambridge, MA, 1982.

[34] T. Lozano-Perez, M.T. Mason, and R.H. Taylor, "Automatic Synthesis of Fine-Motion Strategies for Robots", *Int. J. of Rob. Res.*, Vol 3, No 1, pp. 3-23, Spring, 1984.

[35] D. Lyons and A.J. Hendriks, "Planning by Adaptation: Experimental Results," *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 855-860, San Diego, CA, May 1994.

[36] D. Lyons, "RS: A Formal Model of Distributed Computation For Sensory-Based Robot Control," COINS Tech. Rept. 86-43, University of Massachusetts at Amherst, Sept. 1986.

[37] M.T. Mason, "Compliance and Force Control for Computer Controlled Manipulators," IEEE Trans. on Sys., Man, and Cybernetics, SMC-11, 6 (June, 1981), pp. 418-432.

[38] R.S. Mattikalli, "Mechanics-based Assembly Planning," Ph.D. Thesis, Department of Mechanical Engineering, Carnegie Mellon University, Pittsburgh, PA, 1994.

[39] B.J. McCarragher and H. Asada, "Qualitative Template Matching using Dynamic Process Models for State Transition Recognition in Robotic Assembly," Trans. ASME J. of Dynamic Systems, Measurement, and Control, vol. 115, no. 2A, June 1993, pp. 261-269.

[40] B.J. McCarragher, "Force Sensing from Human Demonstration Using a Hybrid Dynamical Model and Qualitative Reasoning," *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 557-563, San Diego, CA, May 1994.

[41] P. Michelman and P. Allen, "Forming Complex Dextrous Manipulations from Task Primitives," pp. 3383-3388, *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, CA, 1994.

[42] G.H. Morris and L.S. Haynes, "Robotic Assembly by Constraints," pp. 1507-1515, *Proceedings of IEEE International Conference on Robotics and Automation*, Raleigh, NC, 1987.

[43] J.D. Morrow and P.K. Khosla, "Sensorimotor Primitives for Robotic Assembly Skills," in *Proceedings of the 1995 IEEE International Conference on Robotics and Automation*, Nagoya, Japan, May 1995.

[44] J.D. Morrow, B.J. Nelson, and P.K. Khosla, "Vision and Force Driven Sensorimotor Primitives for Robotic Assembly Skills," to appear in *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IROS), Pittsburgh, PA, August, 1995.

[45] B.J. Nelson and P.K. Khosla, "Integrating Sensor Placement and Visual Tracking Strategies," *Proceedings of the 1994 IEEE Conf. on Robotics and Automation*, pp. 1351-1356, San Diego, CA, May 1994.

[46] B.J. Nelson and P.K. Khosla, "The Resolvability Ellipsoid for Visual Servoing," in *Proceedings of the 1994 IEEE Conf. on Computer Vision and Pattern Recognition* (CVPR94), pp. 829-832, 1994.

[47] B.J. Nelson, J.D. Morrow, and P.K. Khosla, "Robotic Manipulation Using High Bandwidth Force and Vision Feedback," to appear in *Mathematical and Computer Modelling Journal,* 1995.

[48] B.J. Nelson, J.D. Morrow, and P.K. Khosla, "Improved Force Control Through Visual Servoing," in Proceedings of the 1995 *American Control Conference*, Seattle, WA, June, 1995.

[49] B.J. Nelson, N.P. Papanikolopolous, and P.K. Khosla, Visual servoing for robotic assembly. *Visual Servoing - Real-Time Control of Robot Manipulators Based on Visual Sensory Feedback*. ed. K. Hashimoto. River Edge, NJ: World Scientific Publishing Co. Pte. Ltd. pp. 139-164, 1993.

[50] , J. O'Sullivan, T. Mitchell, and S. Thrun, "Explanation-Based Learning for Mobile Robot Perception," in *Symbolic Visual Learning*, Ikeuchi and Veloso(eds.), 1995.

[51] J. K. Ousterhout, *Tcl and the Tk Toolkit*, draft, Addison-Welsey, 1993.

[52] W. Paetsch, and G. von Wichert, "Solving Insertion Tasks with a Multifingered Gripper by Fumbling," pp. 173-179, *Proceedings of IEEE International Conference on Robotics and Automation*, Atlanta, GA, 1993.

[53] R. Palm and H. Fuchs, "Skill Based Robot Control for Flexible Manufacturing Systems," *IFAC Skill Based Automated Production*, pp. 189-195, Vienna, Austria, 1989.

[54] N.P. Papanikolopoulos, B.J. Nelson, and P.K. Khosla , "Full 3-d tracking using the controlled active vision paradigm," *Proc. 1992 IEEE Int. Symp. on Intelligent Control (ISIC-92)*. New York:IEEE, pp. 267-274, 1992.

[55] C.J.J. Paredis, "An Agent-based Approach to the Design of Rapidly Deployable Fault Tolerant Manipulators," Ph.D. Thesis, Electrical and Computer Engineering Dept., Carnegie Mellon University, Pittsburgh, PA, August, 1996.

[56] D. Pomerleau, *Neural Network Perception for Mobile Robot Guidance*, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, 1992.

[57] J.M. Schimmels and M.A. Peshkin, "Admittance Matrix Design for Force-Guided Assembly," *IEEE Transactions on Robotics and Automation*, Vol. 8, No. 2, pp. 213-227, April 1992.

[58] K.K.W. Selke, "A framework for intelligent robotic assembly," *Mechatronic Systems Engineering*, vol. 1, no. 1, pp. 41-51, 1990.

[59] J.J. Shah and M.T. Rogers, "Assembly Modeling as an Extension of Feature-Based Design," in *Research in Engineering Design*, vol 5., 1993, pp. 218-237.

[60] K. Shimokura and S. Liu, "Programming Deburring Robots Based on Human Demonstration with Direct Burr Size Measurement," pp. 572-577, *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, CA, 1994.

[61] D.A. Simon, L.E. Weiss, and A.C. Sanderson, "Self-Tuning of Robot Program Primitives", in the Proceedings of the *1990 International Conference on Robotics and Automation*, Cincinnati, OH, vol. 1, pp. 708-713, May 1990.

[62] J. Simons, H. Van Brussel, J. De Schutter, and J. Verhaert, "A Self-Learning Automaton with Variable Resolution for High Precision Assembly by Industrial Robots," *IEEE Transactions on Automatic Control*, Vol AC-27, No. 5, October, 1982.

[63] R. N. Singer, *Motor Learning and Human Performance: An Application to Motor Skills and Movement Behaviors*, pp. 102-103, MacMillan Publishing Co., New York, 1980.

[64] T. Smithers and C. Malcolm, “Programming Robotic Assembly in terms of Task Achieving Behavioural Modules,” DAI Research Paper No. 417, University of Edinburgh, Edinburgh, Scotland, December 1988.

[65] T.H. Speeter, “Primitive-Based Control of the Utah/MIT Dextrous Hand,” *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 866-877, Sacramento, CA, April 1991.

[66] D.B. Stewart, “Real-time Software Design and Analysis of Reconfigurable Multisensor based Systems,” Ph.D. Thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1994.

[67] D.B. Stewart, R.A. Volpe, and P.K. Khosla, “Integration of Real-Time Software Modules for Reconfigurable, Sensor-Based Control Systems,” *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems,* Raleigh, NC, pp. 325-333, July 1992.

[68] D. Strip, “Technology for Robotics Mechanical Assembly: Force-Directed Insertions,” *AT&T Technical Journal,* Vol. 67, Issue 2, pp. 23-34, March/April 1988.

[69] B. Stroustrup, *The C++ Programming Language*, p. 8, Addison-Wesley Publishing Co., Reading, MA, 1991.

[70] K. Toyama and G.D. Hager, “Keeping Your Eye on the Ball: Tracking Occluding Contours of Unfamiliar Objects without Distraction,” pp. 354-359, Vol. 1, *Proceedings of 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Pittsburgh, PA, 1995.

[71] W.O. Troxell, “A robotic assembly description language derived from task-achieving behaviors,” *Proceedings of Manufacturing International ‘90*, Atlanta, GA, 25-28 March 1990.

[72] E.G. Vaaler and W.P. Seering, “A Machine Learning Algorithm for Automated Assembly,” *Proceedings of IEEE International Conference on Robotics and Automation*, 1991.

[73] R. Volpe and P. Khosla, “A Theoretical and Experimental Investigation of Explicit Force Control Strategies for Manipulators”, *IEEE Trans. of Automatic Control*, vol. 38, no. 11, Nov. 1993, pp. 1634-50.

[74] R.M. Voyles, J.D. Morrow, and P.K. Khosla, “Gesture-Based Programming, Part 2: Primordial Learning,” in *Intelligent Engineering Systems through Artificial Neural Networks, Volume 6; Smart Engineering Systems: Neural Networks, Fuzzy Logic and Evolutionary Programming*, ASME Press 1996.

[75] R.M. Voyles, Jr., G.F. Fedder, and P.K. Khosla, “A Modular Tactile Sensor and Actuator based on an Electrorheological Gel,” in the *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, Minneapolis, MN, April, 1996.

[76] L. Weiss, "Dynamic Visual Servo Control of Robots: An Adaptive Image-based Approach", Ph.D. Thesis, Electrical and Computer Engineering Dept., Carnegie Mellon University, Piitsburgh, PA, 1984.

[77] D. Whitney, "A Survey of Manipulation and Assembly: Development of the Field and Open Research Issues," in *Robotics Science*, ed. M. Brady, MIT Press, 1989.

[78] D. Whitney, "Quasi-Static Assembly of Compliantly Supported Parts," *ASME Journal of Dynamic Systems, Measurement, and Control*, Vol 104, pp. 65-77, March 1982.

[79] J. Yang, Y. Xu, and C.S. Chen, "Hidden Markov Model Approach to Skill Learning and Its Application to Telerobotics," *IEEE Transactions on Robotics and Automation*, Vol. 10, No. 5, pp. 621-631, October 1994.

Chapter 9

Appendix

9.1 Chimera Agent Level

Developing robust robot skills would significantly impact the robot programming problem. Robust skills must be expressed in terms of the task representation and driven by task events. A combination of continuous and discrete control layers is necessary to implement control strategies for complex tasks. The Chimera real-time operating system provides a strong foundation for this work in the reconfigurable subsystem (SBS) library. This architecture makes reusable real-time software a reality and hides many real-time programming details from the application developer. The Chimera reconfigurable library has 3 levels defined: IOD, SAI, and SBS (Figure 1). The SBS level is based on periodic, port-based objects which implement data processing and control functions. What is missing, however, is a "meta-control" method -- that is, controlling the SBS module sets which implement continuous controllers. This meta-control is usually driven by discrete events. In robotics, for example, the event of contacting the environment requires the transition from a position controller to a force controller. In addition, sensor-based control requires the on-line computation of setpoints and/or trajectory parameters for input to the controllers as well as the monitoring and detection of different events in different parts of the control strategy.

To support event-driven, sensor-based robot programs, the Agent level provides two new objects for managing SBS module configurations. The first object is an "agent," which is similar to SBS modules in implementation. The second object is a finite-state machine for implementing event-driven control flow. Unlike the agent object, the finite-state-machine does not support general-purpose computation, but it does support the processing of events (i.e. the connection of specific events to state transitions).

A graphical programming environment is available to support the construction of finite-state machine objects which allows an intuitive method of viewing and editing the state-machine structure.

Agent Level SBS module management & event-driven control
meta-control, on-line setpoint generation,
response to asynchronous discrete events

D. Stewart

SBS Level -- reconfigurable, port-based objects -- “continuous” control

SAI Level -- sensor/actuator interface level (force sensors, etc.)

IOD Level -- card-level i/o (e.g. serial ports, parallel ports, etc.)

Figure 1: Chimera Reconfigurable Levels

This report describes the agent level in two sections. The first section briefly describes the design at a high level. It covers how the agent level is implemented on the real-time system, the structure of the agent and fsm objects, event processing and generation, the connection of the source code files to the high level description, and a comparison to Onika. The second section is a user’s guide which describes how to get started, the agentcmdi and SPI (skill programming interface) user interfaces, and listing of agent library functions useful in implementing agent objects.

9.1.1 Software Design Issues

9.1.1.1 High-Level Design

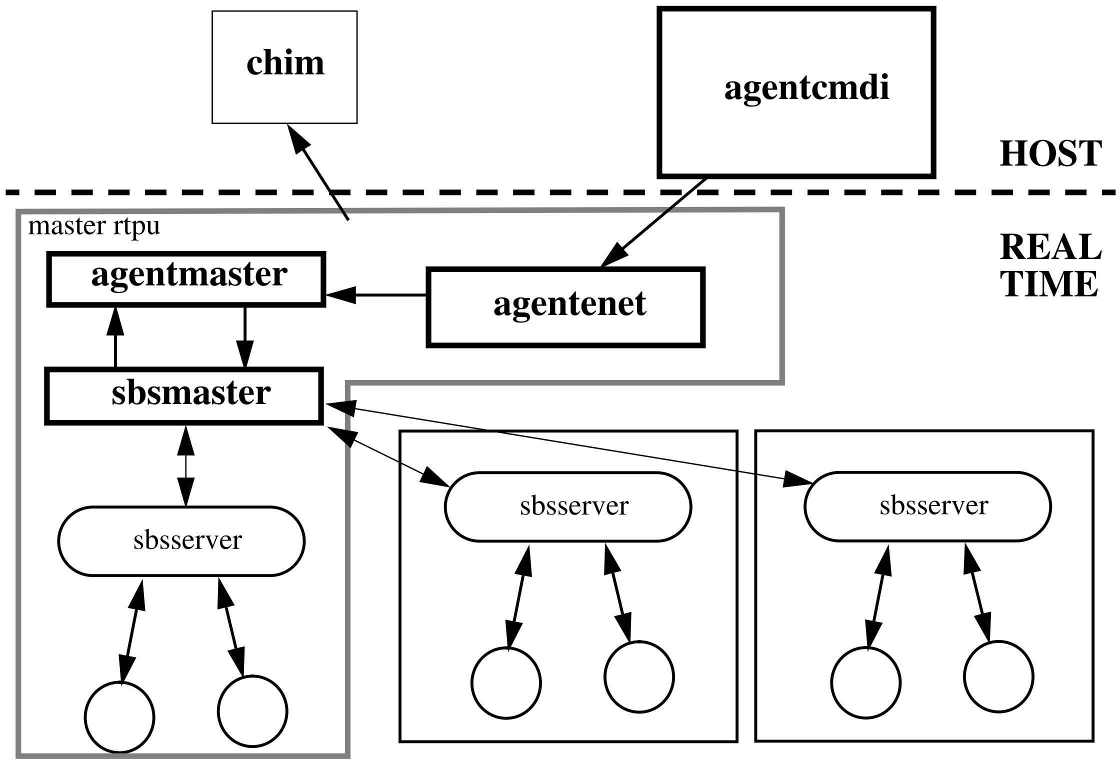


Figure 2: Agent High Level Design

The agent implementation has focused on the real-time side to provide low-latency response to real-time events. Three processes implement the agent level on the master rtpu: sbsmaster, agentmaster, and agentenet. Agents are executed in the sbsmaster process, which also receives the sbs events and issues sbs commands. FSM's are executed in the agentmaster process and agent and sbs commands at the FSM level are sent to the sbsMaster for execution. The user interface (agentcmdi) has been moved to the host side, since it is fundamentally non-real-time. The agentenet process is an ethernet interface to the user interface (or to other command generating processes -- e.g. CAD-driven planners). The execution of agent objects (i.e. methods) occurs inside the sbsmaster process, which issues the sbsInit and is thus the only process which can execute sbsXXX commands. Note that because agents are executed in one process that there is the potential to tie up this process indefinitely. Care should be taken to make agents do limited computation to preserve the ability to react quickly to events. Larger computational burdens should be encapsulated as separate processes with the appropriate priority for real-time scheduling. FSM's are executed in the agentmaster process. Agent and sbs commands inside FSM states are sent to the sbsmaster for execution.

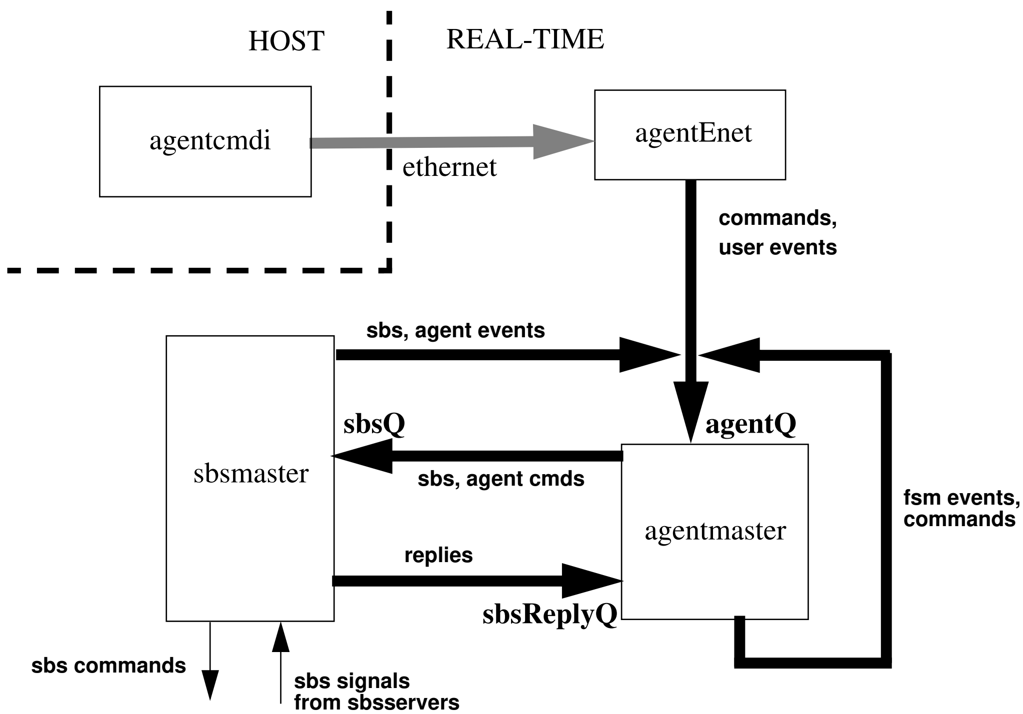
Agents and FSM's are designed to be persistent. For example, the control agent exists across state changes in FSM's -- this way different FSM state's can request a particular control mode from the control agent. Note that careful management of this capability is called for. One can develop dangerous programs where multiple co-existing objects demand incompatible modes from a single object. Objects can be written to be safe under these conditions, but coordination is up to the programmer.

Unlike the sbs level where it is considered an error to spawn and already spawned module, in the agent level this is allowed and even commonplace. A connect counter is maintained on each object and create/spawn calls increment this counter. Similarly, destroy/kill calls decrement this counter; an object is only destroyed/killed (its destroy method run) if the connect counter decrements to zero. In this way, an object can be made persistent across objects which may create it and then themselves go away. However, it is important to ensure that create/spawn calls are balanced with destroy/kill calls for a particular object.

9.1.1.2 Messaging and Control Flow

The sbsmaster process blocks on any sbs signal. Thus, an sbsSigSend from any module will unblock the sbsmaster process. In addition, a special synchronous sbs module (sbsmsg) exists to inform the sbsmaster process when a command from the agentmaster process is pending. The communication between the various agent level processes (agentmaster, sbsmaster, agentEnet) is via Chimera's message queue facility.

Three message queues are used implement the agent level interprocess communication: sbsQ, sbsReplyQ, and agentQ. The sbsQ is the input queue for the sbsmaster process; sbs and agent commands from the agentmaster are sent via this queue. The sbsReplyQ is used to indicate when the command is complete; the agentmaster process waits at the sbsReplyQ after sending a command -- this implements handshaking so that the next command is not sent until the current command has finished. AgentQ is the prioritized input queue for the agentmaster process; the agentEnet sends commands from the user (or other host-based programs which arrive other ethernet), the sbsmaster sends sbs and agent events, and the agentmaster sends (itself!) fsm events.



There are two ways of accessing the agent level: through the agentEnet process and from the sbs level. The sbs level provides a path for real-time events detected by sbs monitoring modules to generate a configuration change. The agentEnet process allows slower, non-real-time processes (like the user or a planner) to provide commands to the system and force configuration changes.

9.1.2 Agent Object Definition

A agent is defined by a source file and a configuration file, like an sbs module. It has three methods (create, destroy, and exec) and a user-definable local data structure. In addition, the agent object accepts parameters and can access the SBS state variable table. The following agent capabilities are immediately useful:

1. A set of interrelated modules which require tight (and careful) management can be effectively encapsulated inside an agent. Clients can then simply request a particular function or service from the agent without worrying about internal details. A collection of n modules will have generally (at least) 2^n states, only a fraction of which are actually needed. We have used this capability to manage the transitions between various control modes (joint, cartesian, and force) with a robot manipulator.
2. Often, a collection of SBS modules defines a higher-level function with its own parameterization. These agent parameters must be distributed to the individual SBS modules. The agent object provides support for this and allows general purpose computation for checking, computing, or generating parameters for these modules at run-time.

Appendix A shows an example of an agent source file -- it is obviously heavily based on the Chimera sbs module framework. *It is the user's responsibility to correctly set several parameters in the agtTask_t (pointed to by atask) data structure*: **statenames** (an ordered array of strings defining the states of the agent), **nstates** (the number of states defined for the agent -- must match the dimension of the statenames array), and finally the user must initialize the **state** variable. This must all be done inside the create method.

For any agent, the necessary components in the source file are:

1. local data structure definition
2. AGENT_MODULE(foo) statement
3. create, destroy, and exec methods
4. Inside create method:
    - set atask->statenames to a static array of state strings for this agent
    - set atask->nstates to the number of states (must match dimension of statenames above)
    - set atask->state to an integer value of the initial state (index into statenames)

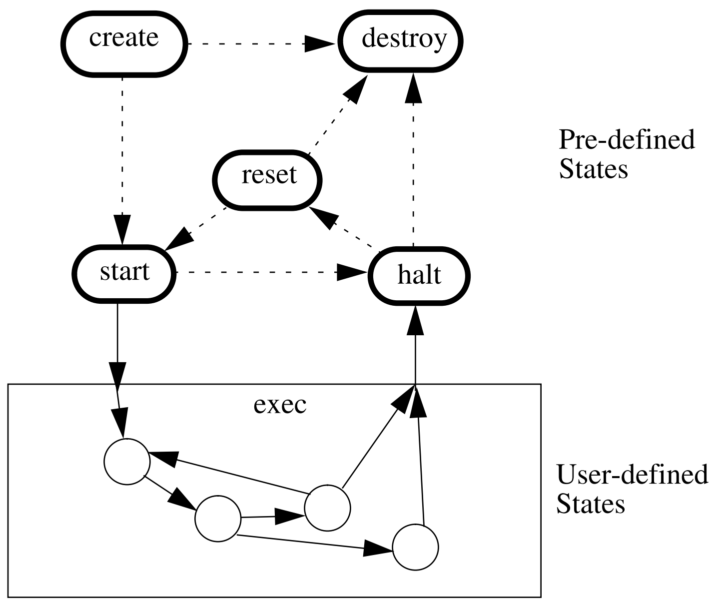
Typically, the user will have a switch statement inside the exec method and a set of user functions to implement transitions into each state from every other state. The dithcomb agent shows how to access the state variable table and use agent parameters.

The only two required lines in an agent configuration file are AGENT (which points to the object code of the agent) and EOF, which ends the file. Often a series of MODULE lines will be included to specify sbs module configuration files that the agent uses. Note that connectivity is still determined by the sbs config files, so developing a custom rmod set for the agent is a good idea to ensure that connectivity is not inadvertently modified (through changing an rmod file which is shared with other applications). Parameter configuration lines are also possible (see dithcomb.agt).

9.1.3 FSM Object Definition

FSM objects are collections of states connected by transitions triggered by object-generated events. Each FSM state has two lists associated with it: a command list and an event list. The command list is executed when entering the state and can consist of commands to any type of object.

There are five (5) pre-defined states: create, destroy, halt, reset, and start. Other states are executed with the 'exec' command. Generally, resources (sbs modules, agents, and other fsm's) which are required by this fsm are 'created' (or spawned) during create. The fsm is 'started' with the start command.



FSM's are stored as text configuration files which are executable in the agent library. Fortunately, FSM's can be graphically created and edited using the SPI (Skill Programming Interface), a menu-driven tcl/tk graphical programming interface. There is no source code to write for an FSM, and no general computation possible. FSM's return value is their exitcode and is generated when the FSM is 'halted.' Thus any event which leads to the halt state should specify the exitcode (if not provided, the default exitcode is 2). A halted FSM informs its FSM parent of its exitcode.

The FSM is instantiated with the create command, and removed with the destroy command (see Software Description for more subtle details). The start command begins execution. Generally the halt command stops execution and puts the FSM into a 'safe' state. The reset command get the FSM ready to start again. Generally one will put the first state of the user's program into start and then an event will transition into another user state. On completion, the user code executes the halt state. On exiting the halt state, the exitcode is returned to the FSM's parent. A fixed number of FSM's can exist at any one time.

The command syntax is:

type	**command**	**object**	**[args]**

Example commands:

sbs	**spawn**	**cisxdot**	**control**
agt	**create**	**control**	
fsm	**create**	**foobar**	

The event syntax is:

type	**object**	**signal**	**nextstate**	**[exitcode]**

Example events:

sbs	**cisxdot**	**103**	**s0**	
agt	**control**	**203**	**halt**	**2**

9.1.4 When should I use an agent or an fsm?

Agents are good for encapsulating the management of a collection of sbs modules. They offer two advantages over FSM's: 1) they support general computation which means parameters can be computed and distributed to sbs modules and state variables can be read and written; and 2) they allow slightly more complete state machine connectivity to be specified (e.g. a four state machine will have 12 transitions between the various states possible). Their disadvantage is that their creation is totally text-based. Thus agents should be reusable objects which can justify their development cost. Note also that agents can only use sbs objects and not other agents or FSM's.

FSM's, on the other hand, are very easy to create, but are less capable in terms of computation. An FSM can reference sbs modules, agents, and other FSM's. But there is no general purpose computation available with an FSM, it is strictly an event responder. Because it has no computational ability, a FSM does not support parameters. It is possible to combine an (FSM, agent) pair to embed computation inside an FSM. FSM's are generally sequencing control structures which are driven by real-time events, which are typically generated by sbs objects and agents (but may also come from other fsm's).

9.1.5 Event Generation and Processing

Fundamentally, all events begin at the sbs level as sbs signals. An SBS object may generate an event by returning a particular integer value from xxxCycle. In addition, several defines and macros exist for specifying event values. When the appropriate event values are *returned* from xxxCycle, then an sbsSigSend call is made which is transparent to the user. This is the “normal” method of event generation at the SBS level -- when an SBS module finishes its task (such as a trajectory or monitoring for an event), then it turns itself OFF and returns an event value indicating the status. If data is involved, then it is written to the svar table and the object processing the event retrieves it from there.

Agent events are generated inside the agent source code by setting atask->signal to a non-zero value inside the exec method. If the stask argument is NULL, then the exec function has not been called due to an sbs event. A non-zero atask->signal value will cause the agent signal to be sent to the agentmaster process where fsm's are executed. An fsm event is generated when the fsm enters the halt state. Any fsm state can transition to the halt state, and an exitcode can be specified to indicate the fsm status when entering halt.

When running, each object has a parent which started it running. SBS objects can have either agents or fsm's as a parent. Agents can have only FSM's as their parent. An FSM can have only another FSM as its parent. Though many objects might *connect* to a particular object, only one object at a time is its parent (the one that ‘start’ed or ‘on’ed it). Events are specified as hexadecimal bit patterns. The value of an event for an object should be well-documented for that object but for sbs signals can depend on the version of Chimera which is used. Returning a SBS_FINISHJOB(x) value is an easy way to generate events when a module turns itself off. Alternatively, a module may issue an sbsSigSend call to send a signal as well.

Table 2: Event/Signal Macros

Event/Macro	signal value
`SBS_FINISHJOB(x), x>=0`	`0x0B + (1<<(9+x))`
`SBS_OFF_OK or SBS_FINISHJOB(0)`	`0x20B`
`SBS_OFF_FAIL or SBS_FINISHJOB(1)`	`0x40B`

Sbs events are directed to their parent: either an agent or an fsm/user. If the sbs module is owned by an agent, the sbsmaster traps the event and calls the exec method of the agent with the sbs signal and task arguments. The agent can then process the event and potentially generate another event to its parent (an fsm/user) by setting atask->signal to a non-zero value. Or the agent may “absorb” the event without generating another event. If an sbs event is not trapped by an agent, then it is propagated to the agentmaster where it filters through the currently running fsm’s. The order of event processing in the fsm hierarchy is “most-recent first.” The most recent fsm is the latest to be started. Currently up to 32 state machines can be running at once (the user is considered fsm #0). If an event is unmatched to any object it is printed for user information; unmatched events cause no particular action to occur (i.e. nothing is halted or aborted due to an unmatched event). User events may also be generated with the ‘event’ command inside agentcmdi.

9.1.6 Comparison to Onika

The current agent level has the following advantages relative to Onika:

1. Event-switching decisions are made entirely on the real-time side. Onika made these decisions on the host side, which was separated by non-real-time ethernet link from the process controlling the SBS modules (sbsmaster).

2. Module management objects (agents) with general-purpose computation capability. Onika supported no general-purpose computation ability.

3. Support for potentially the entire state-space ($\sim 2^n$) of an n-module configuration. In Onika, all states were considered either off or on in a particular configuration.

4. Explicit support for event-driven control, with state machines allowing complex, event-driven behavior to be easily specified.

5. Agents can actually generate/define setpoints and trajectories at run-time, which is

a key requirement in developing sensor-based control algorithms.

The agent level has the following disadvantages relative to Onika:

1. No graphical support for configuring sbs modules -- this is especially apparent when trying to verify the connectivity of state variables which is determined solely by the rmod files. Onika displays this graphically and allows editing of rmod files to adjust the connectivity.
2. No constraint-enforcing interface at the high-level like Onika's puzzle pieces. Onika's implementation essentially enforces data types which is a pretty small constraint in robot programming. There is a very small space of correct setpoints for a particular task goal out of a fairly large set which satisfy the data type constraint.

9.1.7 Code Distribution

There are four source files implementing the agent library:

1. sbsMaster.c -- generally functions which run in the sbsmaster process
2. fsm.c -- fsm support functions
3. agentmaster.c -- generally functions which run in the agentmaster process
4. agentenet.c -- function for agentenet process

In addition, two header files are used:

1. agent.h -- data structure definition, defines
2. agtfuncs.h -- function prototypes

9.2 Agent User's Guide

9.2.1 Getting Started

New setup:

1. create an 'agent' directory in your chimera path (i.e. ~jmorrow/chim/agent). This directory will hold the configuration files (*.agt, *.fsm) for both new types of objects.
2. Link your main program with the 'agent' library, but *the agent library must be linked before the recfg library*.
3. Save the source code for the agent objects in your chim/src/module chimera directory -- Put a marker (_agent_) in the file so that you can use 'grep' to find 'agent' files quickly.
4. Add agent source code files to your modules and main Makefiles.
5. Construct a symbolic link in your chim/bin directory to ~troikbot/chim/bin/agentcmdi, which is the command interface executable file.

6. Construct a main file with the only line in the master rtpu main **if** statement as: **agentStart("d5");** /* or whatever chimera host you're using */. Notice that unlike sbsCmdi, agentStart does not return, so don't put any code after it.
7. Make a host-dependent sbs file: mainE5.sbs for running on E5 (host must be in caps).

To run:

1. Open two windows into chimera host (d5 or e5).
2. start chim in one; download your "new" main and execute it
3. in the other window, run **agentcmdi d5** (wait for the **SBS >** prompt)
4. Type your commands into the agentcmdi window

9.2.2 Agentcmdi User Interface

The agentcmdi user interface is largely implemented with the cmdi() function provided in Chimera; as such, it is relatively easy to add additional commands (of course the agentenet and agentmaster processes are also usually affected). A couple of other functions from sbscmdi() were copied and slightly modified as well. The result is a multiple-mode textual interface for sending commands to the real-time system. This user interface could easily be upgraded to a graphical user interface (e.g. with tcl/tk). In addition, other software packages running on the host or on the real-time side (e.g. a CAD-based planner) could generate commands for the real-time system and send them via ethernet.

Agentcmdi has 4 modes: sbs, agt, fsm, and user; the mode can be changed by typing the desired mode name. The first three correspond to what type of object can be sent a command. In user mode, all three object types can be sent commands, but all commands must be preceded by the type identifier (i.e. sbs, agt, or fsm). The primary use of the user mode is to allow batch files to be used which mix object commands. All modes support the display and status commands. Display allows you to display the value of different state variables and the status command gives status on all three types of objects in the system. In addition, the event command allows the user to generate user events which can drive the state transitions of an fsm.

The sbs mode supports (spawn, on, off, clear, reinit, and kill). It does not (yet) support other (familiar) sbs commands (notably the 'module' command). It also does not have a default module, so you must type in the module name. The agt mode supports (create, destroy, and exec) commands. The fsm mode supports (create, destroy, reset, halt, start, and exec) commands. On-line help is generally available for the commands. Note that currently the fsm exec command takes a state *number*, not a state name like the agent object. Thus, it can be inconvenient to user-direct the fsm execution outside of the predefined states which can be invoked by their respective names (create, destroy, halt, etc.).

9.2.3 Skill Programming Interface (SPI)

Run with: **/usr/vasc/local/pkg/tcl/bin/wish /IUS/usrf1/troikbot/tcl/spi.tcl**

SPI is a tcl/tk program for graphically constructing the FSM configuration files. It is not fool-proof, but merely an aide to an informed programmer. The process of FSM creation is one of creating states and then filling in individual state command and event lists. Predefined states are green, user states are blue. The selected state is red. The events are associated with transitions to other states and are shown as black arrows. Only one FSM can be displayed at a time; however, multiple copies of the SPI can be run to edit FSM's in parallel (although there is no cut/paste support between different SPI processes).

In addition to editing state event and command lists, one can also edit the fsm object list. The object list displays a list of objects (sbs, agt, and fsm) referenced in the FSM. Entering a spawn/create command will add an object to this list; deleting a spawn/create command will remove the object from this list. This adding/deleting is not intelligent -- an object which may still be referenced and yet deleted from the list if a spawn/create command of this object is removed somewhere in the fsm. One can also change an object name and propagate this change throughout the FSM (useful for modifying an sbs module name).

The SPI has several areas. Along the top is a menu bar and a field to display the fsm name.Along the left are buttons implementing various functions described below. A large blank area is the canvas where state machines are drawn; the canvas is also sensitive to mouse events. Finally, at the bottom is a 4 line message area which provides information and prompts the user. Below the message area is a 1 line data entry field.

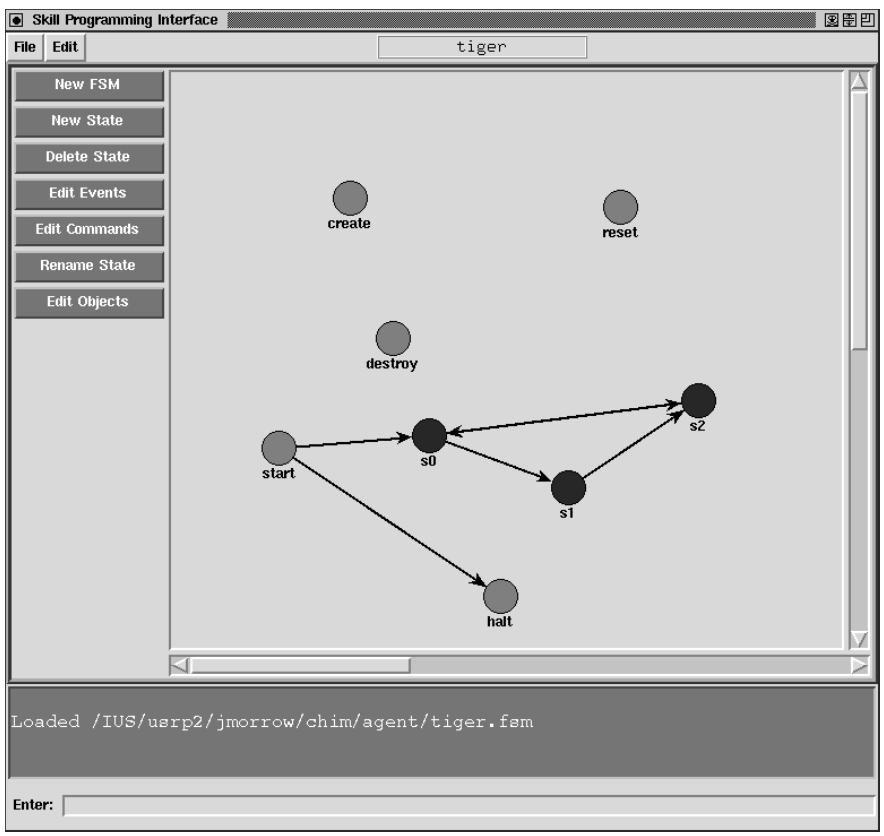


Figure 3: SPI Main Window

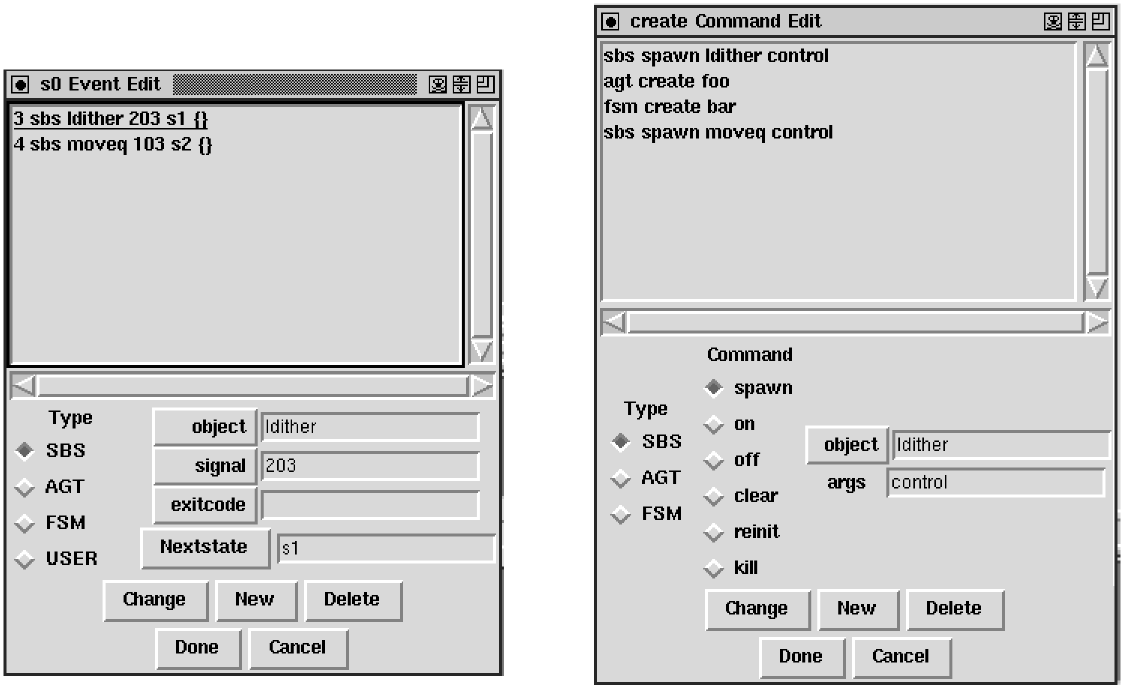


Figure 4: SPI Command and Event List Edit Windows

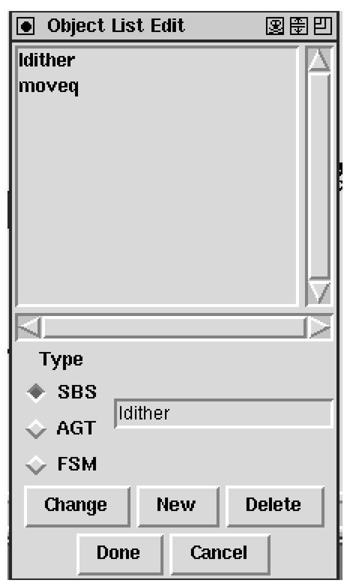


Figure 5: SPI Object List Edit Window

Besides typing in the fields, the user may often click on buttons to get drop menus full of values for placement into fields. Objects are from the object list, which may also be edited. The signals are connected to SBS_FINISHJOB values. The nextstate button sends the user to the canvas to click (B1 or B3) on a state as the nextstate. The command values change in response to the type (sbs command selections are different from agt and fsm selections). One edits the displayed list and then clicks ‘done’ to save this list. Cancel will leave the window without saving the changes.

Table 3: SPI Keyboard Shortcuts

`B1 click on state`	select the state (highlights in red)
`B1 click on state, hold, and drag`	move the selected state (events are updated on B1-release)
`B1 click on arrow`	display the event in message area
`B2 click on state`	display the event and command lists in the message window
`B2 click in canvas`	cancel the current command (if waiting on a pick in canvas)
`B3-click on state`	if a state is currently highlighted, will create an event transition ('blank') from the currently selected state to the B3-clicked state. Note that the user *must* edit the Events of the origin state to specify the object and signal of the created event (event is created with '?' object and signal placeholders).
`shift-B1 on arrow`	move arrow down a level in the display (useful for displaying multiple, overlapping events)
`shift-B2 on state`	edit event list of state
`shift-B3 on state`	edit command list of state

Table 4: SPI Commands

New FSM	Prompts user to entry new name in the message area; instantiates new FSM
Clear FSM	Clears the current FSM from the SPI
Save FSM	Saves the current FSM in the first directory in your CHIMERA_LOCAL/agent list
Save As	Same as above except user is prompted to enter a new name
Load FSM	Allows selection of a FSM from your CHIMERA_LOCAL/agent list. You must select which path and then the particular file (you may not select a path or file not in your CHIMERA_LOCAL list)
Print FSM	Prints <fsmname>.ps file of the canvas in your agent dir
New State	Prompts for name entry; user clicks B1 to place new state on the canvas
Edit Events	Edits events of selected state; or prompts user to B1 click on a state if none is selected; B2 clicking on canvas cancels the command
Edit Commands	Edits commands of selected state; or prompts user to B1 click on a state if none is selected; B2 clicking on canvas cancels the command
Edit Objects	Allows user to edit the object list: add/delete objects and rename objects
Rename State	Allows user to rename a state; propagates that change to all states which reference the changed state
Delete State	Deletes the currently selected state (or prompts user to pick a state to delete); it will confirm the deletion *if* the state has events defined for it.

List Editing:

1. Changes are not saved unless you click 'done'.
2. Generally menu buttons allow quick entry of options (e.g. objects, signals, exit-codes, etc.)
3. To reorder an item in the event or command lists: select item to move with B1, click B2 on item *above* the desired location.

9.2.4 Useful Agent Function Listing

```
int cfigReadMods(cinfo,sbslist,nmods,modnames)
          cfigInfo_t    *cinfo;
          sbsmod_t      *sbslist;
          int           nmods;
          char          **modnames;
```

Reads a set of modules along with their rtpu arguments from a cfig file. Modules must be described in the following syntax:

```
MODULE    NAME          RMOD             RTPU
```

Note that NAME must be identical to that string appearing in modnames. Also, the order of MODULE statements should match the order of NAME's in modnames. RMOD is the rmod filename and RTPU is the name of the rtpu to run the module on.

```
int spawnSbsMods(atask,nmods,sbslist,tasklist)
          agtTask_t     *atask;
          int           nmods;
          sbsmod_t      *sbslist;
          sbsTask_t     *tasklist;
```

Spawns the set of modules in their sbslist order. tasklist holds the sbsTask_t pointers in the same order as the modules are defined in sbslist (so the same set of defines can be used to reference individual modules).

```
int killSbsMods(atask,nmods,tasklist)
          agtTask_t     *atask;
          int           nmods;
          sbsTask_t     *tasklist;
```

Kills the set of sbs modules in their sbslist order.

```
int print_agt_states(atask)
          agtTask_t     *atask;
```

Prints the statenames for the agent -- can be useful in xxxCreate to remind user of valid agent state names.

```
int get_state(statestr,nmods,statenames)
        char          *statestr;
        int           nmods;
        char          **statenames;
```

Returns the index of the statestr in statenames. Useful to use in xxxExec to map the state string identifier into an index which can be used inside switch statements, etc.

```
sbsTask_t *SBS_spawn(agt, rtpu, rmod)
        agtSystem_t   *agt;
        char          *rtpu;
        char          *rmod;
```

Spawns the particular module on the given rtpu. Increments the module's connect counter if module is already spawned. Will issue a warning message is secondary spawn is on different rtpu from original.

```
int SBS_clear( agt, atask, stask, args )
int SBS_on( agt, atask, stask, args )
int SBS_off( agt, atask, stask, args )
int SBS_kill( agt, atask, stask, args )
int SBS_reinit( agt, atask, stask, args )
        agtSystem_t   *agt;
        agtTask_t     *atask;
        sbsTask_t     *stask;
        char          *args;
```

Implement the various sbs functions. These should be called in lieu of the more direct sbs versions inside agent source code.

```
int getsvars(svartable,ixlist,line)
        svarTable_t   *svartable;
        int           *ixlist;
        char          *line;
```

Parses a line of svar names and returns the array of indexes in the svartable for those svars.

```
int read_svars(svartable,ixlist)
        svarTable_t   *svartable;
        int           *ixlist;
        char          *line;
```

Reads the list of svars with svarReadIx calls.

```
int write_svars(svartable,ixlist)
        svarTable_t   *svartable;
        int           *ixlist;
        char          *line;
```

Writes the list of svars with svarWriteIx calls.

Note that svaraliasing is not currently supported at the agent level.

9.2.5 Limitations

Remembering to save is the user's responsibility (no prompting is done).

It is possible to create FSM's which will not execute. Thus the user must be informed.

Agent-level error handling is primitive -- starting again is usually the best option.

The number of FSM's created is currently limited to 32 (the user counts as one).

The number of objects referenced in an FSM is currently limited to 32.

9.3 Control Agent Example

9.3.1 Configuration File

```
#
#  control.agt
#
AGENT     control

MODULE    GRAVCOMP     grav_comp       control
MODULE    PUMAPIDG     puma_pidgC      control
MODULE    PIDLOG       pumalog         second

MODULE    GRIPTOOL     griptool        second
MODULE    FWDKIN       fwdkin          control
MODULE    PUMAXFORM    pumaxform       control
MODULE    IJAC         ijac            second
MODULE    CARTCNTL     cartcntl        third

MODULE    FORCE        aftC            second
MODULE    FMEZFIL      fmezfil         second
MODULE    DAMPCNTL     dampcntl        third
MODULE    VMERGE       dampmerge       third
EOF
```

9.3.2 Source Code

```
/***********************************************************************
 *  Created:  JDM  23-Apr-96
 * Modified:  JDM  23-Apr-96  fixed sequencing omissions
 *
 *  control.c is an agent which implements three controllers and
 *  the transitions between them: a joint controller, a cartesian
 *  hand-based velocity controller, and a damping controller built on
 *  the cartesian hand-based velocity controller.
 *
 *  _agent_
 * ----------------------------------------------------------------------
 *
 *  states:
 *   off -- all modules are off
 *   joint -- joint controller running
 *   cvel -- cartesian velocity controller running
 *   damp -- damping velocity controller running
 *
 ***********************************************************************/
#include <chimera.h>
#include <sbs.h>
#include <cmdi.h>
#include <agent.h>
#include <agtfuncs.h>

#define OFF        0
#define JOINT      1
#define CVEL       2
#define DAMP       3
#define _UNKNOWN   4
#define NUM_STATES _UNKNOWN
static const char *statenames[] = {"off","joint","cvel","damp","unknown" };

#define GRAVCOMP  0
#define PUMAPIDG  1
#define PIDLOG    2
#define GRIPTOOL  3
#define FWDKIN    4
#define PUMAXFORM 5
#define IJAC      6
#define CARTCNTL  7
#define FORCE     8
#define FMEZFIL   9
#define DAMPCNTL  10
#define VMERGE    11
#define NUM_MODS  12
static const char *modnames[] = {"GRAVCOMP","PUMAPIDG","PIDLOG","GRIPTOOL",
                                 "FWDKIN","PUMAXFORM","IJAC","CARTCNTL",
                                 "FORCE","FMEZFIL","DAMPCNTL","VMERGE" };

/* ------------------------------------------------------------------- */

typedef struct {
```

```
  sbsTask_t *tasklist[NUM_MODS];
  sbsmod_t  sbslist[NUM_MODS];

} controlLocal_t;


AGENT_MODULE(control);  /* see sbs.h for SBS_MODULE def'n */



/************************************************************************
 *                    U S E R   F U N C T I O N S
 ************************************************************************/

static int off(local, atask, args)
controlLocal_t  *local;
agtTask_t       *atask;
char            *args;
{
          /* ... code to implement the off state from any other state */
          return AGT_OK;
}

static int joint(local, atask, args)
controlLocal_t  *local;
agtTask_t       *atask;
char            *args;
{
          /* ... code to implement the joint state from any other state */
          return AGT_OK;
}

static int cartvel(local, atask, args)
controlLocal_t  *local;
agtTask_t   *atask;
char        *args;
{
          /* ... code to implement the cartesian state from any other state */
          return AGT_OK;
}


static int dampcntl(local, atask, args)
controlLocal_t  *local;
agtTask_t   *atask;
char        *args;
{
          /* ... code to implement the damping control state from any other state
*/

return AGT_OK;
}

/************************************************************************
 *                      A G E N T   M E T H O D S
 ************************************************************************/

/************************************************************************
```

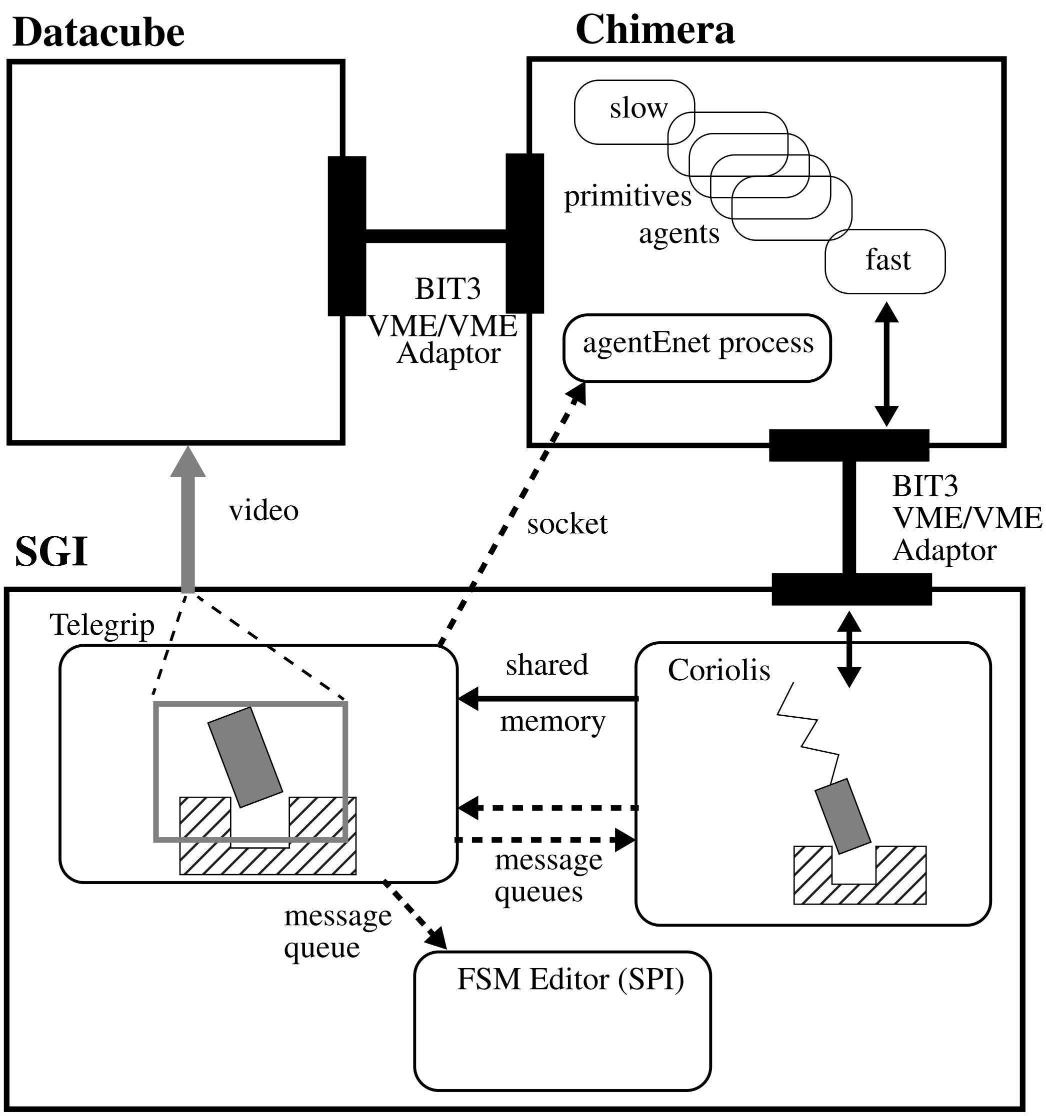
```
 *   controlCreate Method
 ***********************************************************************/
int controlCreate(cinfo,local,atask)
cfigInfo_t *cinfo;
controlLocal_t *local;
agtTask_t *atask;
{
  int i,rv;
  char rmod[MAXNAMELEN];
  char rtpu[MAXNAMELEN];

  kprintf("controlCreate\n");

  rv=cfigReadMods(cinfo,local->sbslist,NUM_MODS,(char **)modnames);
  if(rv==AGT_ERROR) {
    kprintf("Problem reading modules in control\n");
    return AGT_ERROR;
  }
  rv=spawnSbsMods(atask,NUM_MODS,local->sbslist,local->tasklist);
  if(rv==AGT_ERROR) {
    kprintf("Problem spawning modules in control\n");
    return AGT_ERROR;
  }

  if(rv==AGT_ERROR) {
    kprintf("Problem reading sbs modules\n");
    return AGT_ERROR;
  }

  /* read the event list */

  /* write parameter values to svar table ? */
  atask->state = OFF;  /* initial user state */
  atask->statenames = (char **)statenames;
  atask->nstates=NUM_STATES;

  print_agt_states(atask);

  return AGT_OK;
}


/**********************************************************************
 * Destroy Method
 ***********************************************************************/
int controlDestroy(local,atask)
controlLocal_t *local;
agtTask_t  *atask;
{
  /* depending on current state, do some sort of shutdown */

  /* then kill all sbs modules */
  killMods(atask,NUM_MODS,local->tasklist);

  kprintf("controlDestroy\n");

  return AGT_OK;
}
```

```
/*********************************************************************
 *    Exec Method
 *********************************************************************/
int controlExec(local,atask,stask,signal,arglist)
controlLocal_t *local;
agtTask_t  *atask;
sbsTask_t  *stask;
unsigned   signal;
char       *arglist;
{
  char *statestr,*args;
  int nextstate, rv;

  statestr=args=arglist;
  cmdiArg(&statestr,&args,CMDI_BLANKCHARS);

  if(stask==NULL) {   /* a command */

    kprintf("control: statestr='%s'  args='%s'\n",statestr,args);
    nextstate=get_state(statestr,NUM_MODS,statenames);

  } else {            /* an sbs event */

    /* do something here to map the stask,signal pair into
       a nextstate */

    /* for now: clear the event, print, and return */
    kprintf("control: stask=%p  signal=0x%x\n",stask,signal);

    /* clear the signal so it doesn't propagate */
    atask->stask = NULL;
    atask->signal = 0 ;

    return AGT_OK;
  }

  switch(nextstate){

  case OFF:
    rv=off(local,atask,args);
    break;

  case JOINT:
    rv=joint(local,atask,args);
    break;

  case CVEL:
    rv=cartvel(local,atask,args);
    break;

  case DAMP:
    rv=dampcntl(local,atask,args);
    break;

  default: /* invoke error here */
    rv=AGT_ERROR;
```

```
      break;
    }
    return rv;
  }
```

9.4 C^2T: CAD Integration

The CAD integration of primitives integrates the Chimera real-time system via the Agent level, the Telegrip CAD environment, and the Coriolis simulation package. Telegrip connects to Chimera via the agentEnet socket and spawns a process (p2) which runs Coriolis.



9.4.1 Communications

Telegrip is the central software for the user interface and simulation rendering.

9.4.1.1 Telegrip/Chimera

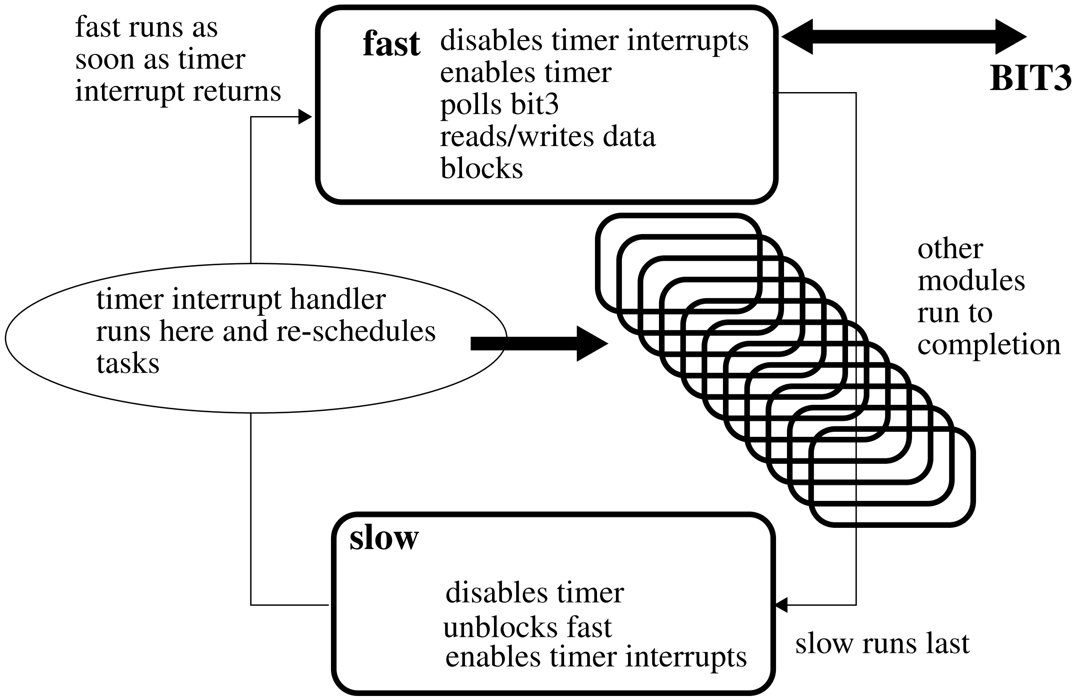
Telegrip connects to Chimera via the Agent Level's agentEnet process. This is a Unix socket-based communication mechanism. The communications protocol allows commands for SBS objects, agents, and fsm's to be sent. This is the path by which primitives are instantiated and run via the CAD integration package.

9.4.1.2 Telegrip/Coriolis

We use Telegrip llti capability to update the position of the moving part from the Coriolis simulation via shared memory. The Chimera tbuf mechanism is used to implement triple buffering so that the consumer/supplier never block and so that the consumer (Telegrip) always uses the latest information. To send discrete messages, two message queues are used. Telegrip sends messages to define geometry, initialize the location, begin the simulation, advance it, and stop it via the message queue. The Coriolis process reports events (received from Chimera) to Telegrip via the other message queue.

9.4.1.3 Coriolis/Chimera

This is the most critical communication and synchronization. The mechanism is via the BIT3 which looks like shared memory on the two systems. But the simulation proceeds at a different time rate than real time, so the two systems must be synchronized. We do this as follows. We spawn two tasks on the Chimera system -- fast and slow -- which have the highest and lowest priorities in the system. These tasks are synchronous tasks which block on semaphores. We start fast and disable the timer interrupt so that no task scheduling is done. Fast polls the bit3 for a time increment written by the Coriolis process. When this increment occurs, fast blocks and all tasks on the ready queue execute. The last one to execute is slow. Slow disables the timer, unblocks fast by giving the semaphore, and enables the timer interrupt to cause the timer interrupt handler to run which runs the task scheduler. When the task scheduler returns (from interrupt), the fast process again runs. It immediately disables the timer interrupts and enables the timer and begins polling again. In this way we are able to "slow down" real time from the VME perspective to the rate of simulation real-time. The current implementation is restricted to running on one RTPU on th real-time system.



9.5 Skill Listings

9.5.1 Square Peg Insertion

#***************				
STATE	0	create	111.0	48.0
COMMAND	agt	create	control	
COMMAND	sbs	spawn	gmove	fourth
COMMAND	sbs	spawn	gripC	control
COMMAND	sbs	spawn	cornerIP	fourth
COMMAND	sbs	spawn	projVS	fourth
COMMAND	sbs	spawn	fstick	fourth
COMMAND	sbs	spawn	movedx	third
COMMAND	sbs	spawn	projRVS	fourth
COMMAND	sbs	spawn	rdither	fourth
COMMAND	sbs	spawn	ldither	fourth
#***************				
STATE	1	destroy	365.0	45.0
COMMAND	agt	destroy	control	
COMMAND	sbs	kill	projVS	
COMMAND	sbs	kill	cornerIP	
COMMAND	sbs	kill	fstick	

```
COMMAND    sbs    kill        gmove
COMMAND    sbs    kill        gripC
COMMAND    sbs    kill        movedx
COMMAND    sbs    kill        projRVS
COMMAND    sbs    kill        rdither
COMMAND    sbs    kill        ldither
#***************
STATE      2      reset       242.0      100.0
#***************
STATE      3      halt        358.0      164.0
COMMAND    sbs    off         projVS
COMMAND    sbs    off         projRVS
COMMAND    sbs    off         fstick
COMMAND    sbs    off         cornerIP
COMMAND    sbs    off         movedx
#***************
STATE      4      start       107.0      153.0
COMMAND    agt    exec        control    damp
EVENT      agt    control     100        5    ?
#***************
STATE      5      touch       117.0      237.0
COMMAND    sbs    on          gmove      0 -1 0 0.01 0.05 1
EVENT      sbs    gmove       10B        6
EVENT      sbs    gmove       20B        3    2
#***************
STATE      6      vision      151.0      294.0
COMMAND    sbs    on          fstick     0 1 0 3 0.01
COMMAND    sbs    on          cornerIP   1 -1
EVENT      sbs    cornerIP    100        12
EVENT      sbs    cornerIP    200        3    2
#***************
STATE      7      vismove     275.0      357.0
COMMAND    sbs    off         projRVS
COMMAND    sbs    on          projVS     -10 10   0 0 1   -0.7071 -0.7071 0  0.7071  -0.7071  0
EVENT      sbs    cornerIP    20B        3    2
EVENT      sbs    projVS      100        8
#***************
STATE      8      tilt        353.0      351.0
COMMAND    sbs    off         projVS
COMMAND    sbs    off         cornerIP
COMMAND    sbs    off         fstick
COMMAND    sbs    on          movedx     A -1 0 1 0.05 0.025
EVENT      sbs    movedx      10B        9
#***************
STATE      9      contact     419.0      328.0
COMMAND    sbs    on          gmove      -1 0 -1 0.01 0.02 1
EVENT      sbs    gmove       10B        10
EVENT      sbs    gmove       20B        3    2
#***************
STATE      10     straighten  480.0      286.0
COMMAND    sbs    on          fstick     0.3 1 0.3 6 0.04
COMMAND    sbs    on          movedx     A 1 0 -1 0.05 0.01
COMMAND    sbs    on          rdither    0 1 0 0.1 20 16
EVENT      sbs    movedx      10B        11
EVENT      sbs    fstick      20B        3    2
#***************
STATE      11     push_in     501.0      188.0
COMMAND    sbs    off         fstick
```

```
COMMAND    sbs    off         rdither
COMMAND    sbs    on          fstick     0.3 1 0.5 8 0.01
COMMAND    sbs    on          rdither    0 1 0 0.3 20 20
EVENT      sbs    fstick      20B        3    1
EVENT      sbs    rdither     10B        3    2
#***************
STATE      12     vis_rotate  201.0      336.0
COMMAND    sbs    on          projRVS    0 2 2  0 0 1  -0.7071 -0.7071 0   0.7071 -0.7071 0
EVENT      sbs    projRVS     100        7
EVENT      sbs    cornerIP    20B        3    2
EOF
```

9.5.2 Press Fit Connector

```
#***************
STATE      0      create      121.0      29.0
COMMAND    agt    create      control
COMMAND    sbs    spawn       gmove      fourth
COMMAND    sbs    spawn       rgmove     fourth
COMMAND    sbs    spawn       movedx     fourth
COMMAND    sbs    spawn       fstick     fourth
COMMAND    sbs    spawn       ldither    fourth
COMMAND    sbs    spawn       gripC      control
#***************
STATE      1      destroy     421.0      27.0
COMMAND    agt    exec        control    off
COMMAND    agt    destroy     control
COMMAND    sbs    kill        fstick
COMMAND    sbs    kill        gmove
COMMAND    sbs    kill        movedx
COMMAND    sbs    kill        rgmove
COMMAND    sbs    kill        gripC
#***************
STATE      2      reset       266.0      70.0
#***************
STATE      3      halt        420.0      116.0
COMMAND    sbs    off         fstick
COMMAND    sbs    off         gmove
COMMAND    sbs    off         movedx
COMMAND    sbs    off         rgmove
#***************
STATE      4      start       117.0      112.0
COMMAND    agt    exec        control    damp
EVENT      agt    control     100        8    ?
#***************
STATE      5      touch       184.0      275.0
COMMAND    sbs    on          gmove      0 -1 0 0.01 0.02 2
EVENT      sbs    gmove       10B        9
EVENT      sbs    gmove       20B        3    2
#***************
STATE      6      mate        415.0      310.0
COMMAND    sbs    off         fstick
COMMAND    sbs    on          fstick     0 1 0 8 0.01
COMMAND    sbs    on          rgmove     0 -1 0 0.25 6.3 0.1
EVENT      sbs    rgmove      10B        7
EVENT      sbs    rgmove      20B        3
#***************
```

```
STATE        7      press       497.0     284.0
COMMAND      sbs    off         fstick
COMMAND      sbs    on          fstick    0 1 0 40 0.01
COMMAND      sbs    on          ldither   0 0 1 0.015 25 40
EVENT        sbs    ldither     10B       11     1
#***************
STATE        8      rotate      117.0     202.0
COMMAND      sbs    on          movedx    A 1 0 0 0.1 0.05
EVENT        sbs    movedx      10B       5
#***************
STATE        9      rotate_back256.0      315.0
COMMAND      sbs    on          fstick    0 1 -1 6 0.02
COMMAND      sbs    on          movedx    A 1 0 0 -0.1 0.025
EVENT        sbs    movedx      10B       10     ?
#***************
STATE        10     relax       341.0     327.0
COMMAND      sbs    on          movedx    A  0 1 0 0.035 0.035
EVENT        sbs    movedx      10B       6      ?
#***************
STATE        11     release     546.0     189.0
COMMAND      sbs    on          gripC     1
EVENT        sbs    gripC       10B       3      1
EOF
```