Following Recipes with a Cooking Robot

by

Mario Attilio Bollini

B.S., Massachusetts Institute of Technology (2009)

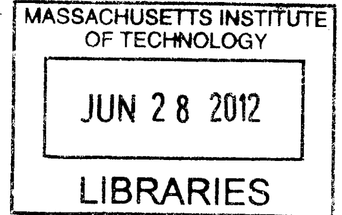
Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 28 2012
LIBRARIES
ARCHIVES

© Massachusetts Institute of Technology 2012. All rights reserved.

Author . . . .
Department of Mechanical Engineering
May 23, 2012

Certified by . . . .
Daniela Rus
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . .
John Leonard
Professor of Mechanical Engineering
Mechanical Engineering Faculty Reader

Accepted by . . . .
David E. Hardt
Chairman, Department Committee on Graduate Students

Following Recipes with a Cooking Robot

by

Mario Attilio Bollini

Submitted to the Department of Mechanical Engineering
on May 23, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

In this thesis, we present BakeBot, a PR2 robot system that interprets natural language baking recipes into *baking instructions* which it follows to execute the recipe, from *mise en place* presentation of the ingredients through baking in a toaster oven. We developed parameterized motion primitives for baking. The motion primitives utilize the existing sensing and manipulation capabilities of the PR2 platform and also our new compliant control techniques to address environmental uncertainty. The system was first implemented as a static finite state machine, which was tested through 27 baking attempts, 16 of which successfully resulted in edible cookies. The system was then implemented as a dynamic state machine, in which the robot estimated the world state and planned sequences of motion primitives to follow the *baking instructions* inferred from the natural language recipe, which was tested thorough five baking attempts of two different recipes, all of which resulted in edible cookies.

Thesis Supervisor: Daniela Rus
Title: Professor of Electrical Engineering and Computer Science

Mechanical Engineering Faculty Reader: John Leonard
Title: Professor of Mechanical Engineering

Acknowledgments

This work was supported by a National Defense Science and Engineering Graduate Fellowship. Thank you for this support. The kitchen is the first step towards my goal of getting robots out of the lab and into the real world.

I would like to thank my advisor, Professor Daniela Rus, for her support and encouragement to continue with what started as a three-month project to cook with the PR2 and spiraled into this thesis. The people I had a chance to work with in the Distributed Robotics Lab were great. I'm thankful to have worked in such an exciting environment filled with cool robots of all shapes and sizes. Kathy Bates was a huge help: from getting receipts of butter, flour, and sugar approved to appreciating the absurd, she was there.

I'd like to thank Jenny Barry and Annie Holladay for their help getting the PR2 to open the oven and for their companionship in the PR2 testing chamber. Sorry for the repeated loud noises (when dropping bowls) and the incessant robot baby-talk. Thanks to Stefanie Tellex for teaching the PR2 to read and for helping me to understand the bigger picture. Thanks to my UROP Tyler Thompson for his help implementing the Recipe Processing.

I want to thank my friends and family for their continued support, I couldn't have done this without you all. You helped me to put everything in perspective and relax after a long day of watching the robot drop things. You were always there when I needed you. The M-Lab / GRIT / Grime team were a huge well of support, moonlighting opportunities, and engineering fun. Thanks for the great times, I'm looking forward to many more.

This work would not have been possible without the Willow Garage PR2 Beta Program: thanks for letting me work on your awesome robot! ROS and the PR2 are making waves in the robotics community thanks to the dedication and skill of the Willow team. Sachin Chitta was a great source of advice and Kaijen Hsiao's manipulation demos were a tremendous starting point for BakeBot development.

Finally, I'd like to thank the PR2 robot itself. Its general reliability, relatively easy to understand software system, and cheerful good nature made this thesis both possible and fun. Its software and hardware malfunctions, frequent and seemingly arbitrary software updates, and occasional obstinance made this thesis something I'll remember forever.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **17**

1.1 Motivation 17

1.2 Problem Statement 18

1.3 Related Work 20

1.3.1 Planning and Manipulation 20

1.3.2 Compliant Control 22

1.3.3 Natural Language Processing 22

1.4 Contributions 22

2 System Setup **25**

2.1 Hardware 25

2.1.1 PR2 25

2.1.2 Hardware Modifications 26

2.2 Environment Setup 29

2.3 Software Substrate 30

3 Key Solution Components **31**

3.1 Object Recognition 31

3.1.1 Object Detection 31

3.1.2 Broad Tabletop Object Detection 32

3.1.3 Object Identification 32

3.2 Compliant End Effector Control 37

3.2.1 Point Compliance 37

3.2.2 Trajectory Interpolation 39
3.2.3 ROS Implementation 39
3.2.4 Discussion 40
3.3 Mixing and Scraping 41
3.3.1 Mixing Trajectories 41
3.3.2 Mixing Experimentation 45
3.3.3 Scraping Trajectories 46
3.3.4 Scraping Experimentation 49
3.4 Recipe Parsing 50
3.4.1 Baking Instruction Set 50
3.4.2 State/Action Space for the Kitchen 50
3.4.3 Reward Function 51
3.4.4 Experimentation 52
3.4.5 Discussion 53
3.5 Dynamic State Machine Framework 54
3.5.1 SMACH 54
3.5.2 Fast-Forward 57
3.5.3 System Design 57
3.5.4 Discussion 61
3.6 Logging 63
3.6.1 Framework 63
3.6.2 Activity Representation 63
3.6.3 Discussion 64

4 System Design Methodology **67**
4.1 Hardware/ROS Abstraction Layer 68
4.1.1 Arm Client 68
4.1.2 Base Client 69
4.1.3 Mix Client 69
4.1.4 Torso Client 69

4.1.5 TF Client 70
4.1.6 Controller Manager 70
4.2 Task Discretization 71
4.2.1 Finding the Ingredients 72
4.2.2 Collecting an Ingredient 74
4.2.3 Mixing the Mixing Bowl 79
4.2.4 Scraping Onto the Cookie Sheet 81
4.2.5 Putting Cookie Sheet Into Oven 83
4.3 Static Finite State Machine 87
4.3.1 Collecting an Ingredient 88
4.3.2 Mixing the Mixing Bowl 92
4.3.3 Scraping Onto the Cookie Sheet 94
4.3.4 Putting Cookie Sheet Into Oven 94
4.3.5 Discussion 96
4.4 Dynamic State Machine 102
4.4.1 Recipe Compiler 105
4.4.2 Collecting an Ingredient 106
4.4.3 Mixing the Mixing Bowl 108
4.4.4 Scraping the Cookie Sheet 110
4.4.5 Putting Cookie Sheet Into Oven 113
4.4.6 Discussion 114

5 Experimental Results 117
5.1 Testing the Static Finite State Machine 118
5.2 Testing the Recipe Processing 120
5.3 Testing the Dynamic State Machine 121

6 Discussion 125
6.1 Compliant Control Techniques 125
6.2 Task Planning 126
6.3 Large Robotic Software Systems 127

6.3.1 ROS 127
6.3.2 Programming Languages 129
6.3.3 Best Practices 129
6.3.4 Feature Requests 130
6.4 Scope and Generality 131
6.5 Future Research 131
6.6 Conclusion 132

A Running BakeBot **133**

A.1 System Requirements 133
A.2 Running the System in Simulation 133
A.2.1 Running the simulator locally 134
A.2.2 Running the simulator on a server 134
A.3 Running the System on the PR2 134
A.3.1 Covering the PR2 135
A.3.2 Executing a Baking Instruction Set 137
A.3.3 Executing Plain-Text Recipes 137

B Software Overview **139**

List of Figures

1-1 The PR2 in the protective garment. 19

1-2 The human interaction with the BakeBot system for recipe execution. . . . 24

2-1 The PR2 in its protective covering with attached spatula standing in the *standard configuration* pose. 26

2-2 The spatula attachment to the right end effector. 28

2-3 The environment setup for the baking task. 29

3-1 A screenshot of the PR2 visualizer showing the bounding boxes for eight objects on the work surface. 34

3-2 The segmentation steps for the HSB framework. 35

3-3 A point in the workspace with independent stiffnesses in x and y and a desired force along the negative z axis. 38

3-4 Top and side views of the compliant trajectories followed by the center of the end effector during circular 3-4(a) and linear 3-4(b) mixing routines. . . 43

3-5 A screenshot of the mixing testing interface that was used to refine the mixing trajectories. 45

3-6 Joint torques during execution of the circular mixing trajectory in a mixing bowl full of cookie batter. 47

3-7 Text from a recipe in our dataset, paired with the inferred action sequence for the robot. 53

3-8 A standard static finite state machine. 55

3-9 A dynamic state machine analogous to the static state machine in Figure 3-8. 56

3-10 A static finite state machine with a nested dynamic state machine. 58

3-11 The architecture of the `smachforward` system. This system is nested within a single state in a larger static finite state machine, such as is shown in Figure 3-10. 59

3-12 Typical output from the `htlogger`. 65

4-1 The baking task was broken into high level subtasks. 72

4-2 Each high level subtask was broken into lower level action sequences. . . . 73

4-3 The eight left end-effector position discretizations around the mixing bowl circumference. 77

4-4 The four steps of the pour maneuver, corresponding to the pour orientations in Table 4.2: (1) the pre-pour position offset from the center of the mixing bowl (Pour Offset), (2) the Pour pose, (3) the Steep Pour pose, and (4) the Super Steep Pour pose. 78

4-5 The segmentation plane (shown on the left) is coincident with the front of the oven and the hanging tablecloth in front of the table. To open the oven (shown on the right), a force-compliant trajectory is executed with stiffnesses in the X direction and a downward force in the Z direction. . . . 84

4-6 The high level BakeBot static state machine. 89

4-7 The state machine that pours an ingredient into the mixing bowl. 90

4-8 The state machine that mixes the mixing bowl. 93

4-9 The state machine that scrapes the batter out of the mixing bowl and into the cookie sheet. 95

4-10 The state machine that puts the cookie sheet into the oven. 97

4-11 The motion primitive set (left) mapped onto the lower level action sequences (right). 101

4-12 The end-to-end BakeBot system. The natural language recipe is processed into *baking instructions* which are compiled into a hierarchical dynamic state machine (as represented by Figure 4-13) and executed. 103

4-13 The high level BakeBot state machine for Chocolate Afghan cookies. . . . 104

5-1 Failure modes of baking attempts using the static finite state machine to execute *baking instructions* to produce Chocolate Afghan cookies. 120
5-2 The steps of executing the Chocolate Afghan recipe with the BakeBot system. 123

A-1 The steps of covering the PR2 in its protective garmet. 136

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

3.1 A mapping between the bowl order and ingredient name for two recipes. . . 34

4.1 The high level subtasks of the baking task with their respective requirements and effects. 71

4.2 For each cardinal discretization around the bowl circumference (shown in Figure 4-3), the left end effector: grasp orientation; pour offset direction from mixing bowl center; and orientations for the pour, steep pour, and super steep pours. 78

5.1 Average runtimes for each of the baking subtasks for Chocolate Afghans. . 119

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

This thesis presents BakeBot, an end-to-end robotic system which is able to read and execute simple recipes. The robot is initialized with a set of ingredients laid out on the table and a set of natural language instructions describing how to use those ingredients to cook a dish. BakeBot parses the text of the recipe and infers a robot action sequence corresponding to the instructions. It executes the action sequence on the PR2 robotic manipulation platform, performing the motion and task planning necessary to follow the recipe to create the appropriate dish. If BakeBot is unable to execute an instruction, it asks its human partner for assistance and then continues with recipe execution.

The BakeBot system combines compliant control techniques, motion planning, task planning, natural language processing, and object recognition to execute recipes. The end-to-end system and its subsystems were extensively tested. An intermediate implementation of BakeBot was tested through 27 baking attempts, 16 of which successfully resulted in edible cookies. The final BakeBot, with task planning and recipe processing capabilities, was tested through five attempts covering two recipes downloaded from the Internet, all of which successfully resulted in edible cookies.

1.1 Motivation

Cooking is one of the most important activities that takes place in the home; a robotic chef capable of following arbitrary recipes has many applications in both household and

industrial environments. We envision a robotic chef, the RoboChef, which when presented with ingredients can find a recipe to utilize them on the Internet and autonomously follow it to prepare a dish for its human partners. The RoboChef is a grand challenge in the field of robotics, requiring advancements in object recognition, motion and task planning, manipulation, and robotic system design.

The kitchen environment in which the RoboChef performs its duties is a semi-structured proving ground for algorithms in robotics. It provides many computational challenges, such as accurately perceiving ingredients in cluttered environments, manipulating objects, and engaging in complex activities such as mixing and chopping. Yet it also allows for reasonable simplifying assumptions due to the inherent organization of a kitchen around a human-centric workspace, the consistency of kitchen tools and tasks, and the ordered nature of recipes. Human chefs make many of these simplifications themselves: arranging measured ingredients in front of them on the preparation table (*mise en place*); keeping kitchen tools ready at hand; and buying matching sets of pots, pans, and spatulas.

The capabilities of the RoboChef system would transform our relationship with robots. It would enable versatile robotic household assistant to help with the tasks of daily life, an invaluable asset for a rapidly aging population. Manufacturers can apply the task planning and manipulation capabilities of such a system to implement versatile production facilities staffed with robots programmed with human-friendly "recipes" for product manufacture, rather than the virtually single-use robotic manipulators currently used.

1.2 Problem Statement

We present BakeBot, a first step towards realizing the RoboChef. BakeBot is initialized with a set of ingredients laid out on the table and a set of natural language instructions describing how to use those ingredients to cook a dish such as cookies, salad, or meatloaf. For example, ingredients might include flour, sugar, butter, and eggs arrayed in labeled bowls on the table, and instructions might include statements like "Add sugar and beat to a cream." The robot must parse the text of recipe and infer an action sequence in the external world corresponding to the instructions. It then executes the inferred action sequence on

Figure 1-1: The PR2 in the protective garment. The garmet mitigates the risk of spills and splashes during the cooking process.

the PR2 robotic manipulation platform, performing the motion and task planning necessary to follow the recipe to create the appropriate dish. For example, for the instructions above, the robot might empty the bowl of sugar into the mixing bowl, then stir the ingredients in the bowl. If the robot needs assistance executing an instruction, it asks its human partner for help, then continues with the recipe. Figure 1-2 shows human interaction with BakeBot during recipe following.

We assume:

- BakeBot has a spatula affixed to its manipulator, enabling it to mix,
- the pre-measured ingredients are presented to BakeBot on the preparation table,
- the mixing bowl and cookie sheet are presented to BakeBot on the preparation table,
- and the toaster oven is preheated to the temperature appropriate for the recipe.

The inputs to the system are:

- a plain text description of the recipe,
- and a labeled model of the environment including the relative positions of the ingredients to one another and the location of tools such as the oven.

The output from the system is:

- the finished baked good in the cookie sheet within the opened oven.

1.3 Related Work

In this thesis we demonstrate an end-to-end robot cooking system capable of implementing any baking recipe that requires pouring, mixing, and oven operations on raw ingredients provided to the system premeasured at arbitrary points in the workspace. Our system is able to follow recipes downloaded from the Internet; we demonstrate it by following two different recipes in the real world and by further evaluating its performance on a larger test set in simulation. It is built upon the ROS PR2 system and integrates many previously independent pieces of software. The combined system of existing software systems and our novel algorithms and planning and control techniques demonstrates singular end-to-end performance.

1.3.1 Planning and Manipulation

Kemp et al. [16] discussed the challenges faced by robot manipulators in human environments. BakeBot implements similar solutions to those proposed, relying heavily on perception, making common-sense simplifications of the manipulation tasks, and relying on its human partner for assistance with tasks it cannot perform itself. BakeBot utilizes the low-level manipulation and perception system described in [23] and implemented in the ROS `pick_and_place` manipulation pipeline, which was designed for the mobile manipulation of household objects with the PR2 platform [10]. BakeBot combines the individual components of the tabletop manipulation system to execute a complicated household task.

Work has been done by Perez et al. to improve the PR2's manipulation performance using RRT* [22], and by Cohen et al. using a heuristic based search over a motion primitive set [11]. Our approach is similar to the work done by Cohen, but with a lower-resolution primitive set. Implementing a planner similar to Perez's has the potential to eliminate some of the planning failures experienced during BakeBot testing.

Our approach did not utilize reinforcement learning. All of our motion primitives were defined a priori. Reinforcement learning techniques have been demonstrated on the PR2 [21] and could generate motion primitives that can be incorporated into the BakeBot planning system.

Gravot et al. [12] presented a hybrid planning approach to humanoid robotic cooking in simulation. BakeBot implements a symbolic planner [13] to plan sequences of motion primitives to execute a baking instruction and a separate geometric planner to execute the motion primitives. The hierarchical nature of the baking subtasks, each accomplished by motion primitives described in terms of their preconditions and effects, facilitates their future integration with hybrid symbolic and geometric planner, such as a hierarchical task network [20, 26] or the hierarchical planning in the now (HPN) framework [15]. Work is currently being done to port the HPN framework to the PR2 system.

Beetz et al. [6] have demonstrated processing instructions from the Internet to make pancakes, dispensing pancake batter from a premixed container and flipping the pancakes on a skillet using a robot team with a PR2 for basic manipulation and a pair of Barrett arms for complex manipulation tasks. Our work differs in our approach to recipe construction and in our emphasis on low-level manipulation using plans of motion primitives on the PR2 platform. Becker et al. have demonstrated handle recognition and the opening of cabinets and other articulated containers using specialized kinematic controllers [5]. Our approach utilizes compliant control techniques to open articulated containers.

We are not aware of other comparable end-to-end robotic systems for recipe execution. In this thesis we present a unified end-to-end system for execution simple plain-text recipes collected from the Internet. The BakeBot system can interpret simple baking recipes into robot instructions and can execute these arbitrary instructions, with the help of its human partner, to follow the recipe.

1.3.2 Compliant Control

Force and compliant control techniques are not new [4, 14, 18]. BakeBot's `ee_cart_imped` controller builds upon this body of work, implementing it within the PR2's realtime control loop. The controller's trajectory interpolation feature and packaging within the ROS framework enable the application of force/compliant control policies to the humanoid manipulation challenges the PR2 was designed to address. While sample code was provided in the ROS tutorials for implementing a compliant controller within the realtime control loop, we are not aware of any released ROS packages that provide force/compliance control at points or over trajectories. The BakeBot compliant controller has been reused to write, draw and erase, open cabinets, clean a table, and sweep with the PR2.

1.3.3 Natural Language Processing

Many previous authors have described robotic systems for following natural language instructions. Previous work focused on understanding natural language route directions [9, 17, 19] and mobile-manipulation commands [25]. Similar to existing approaches, our system learns a model that maps between the text of the instructions and a reward function over actions in the predefined state-action space, specifically focusing on the domain of following recipes. Recipes present a unique challenge to these approaches because of the richer space of actions inherent to the cooking domain.

1.4 Contributions

The technical contributions of this thesis are:

1. an end-to-end system for robot baking,
2. an algorithm for locating ingredients on a broad tabletop,
3. a general compliant endpoint trajectory controller for the PR2 robotic manipulation platform,

4. a compliant motion algorithm capable of mixing ingredients of different texture into a uniform dough inside a bowl,
5. a language processing system that can interpret natural language recipes into *baking instructions*,
6. a planning system that creates plans of motion primitives at runtime for execution on the PR2 robotic manipulation platform,
7. a hierarchical task logging system,
8. a planner that can handle any sequence of *baking instructions* and ingredients to execute pouring, mixing, and oven operations,
9. and a set of best-practices for the implementation of complex robotic manipulation tasks.

This thesis presents:

1. the environment, hardware, and software setup for the BakeBot system,
2. the key solution components created to enable BakeBot,
3. an overview of the bottom-up system development,
4. experiments performed on the BakeBot system,
5. and a discussion of overall system performance.

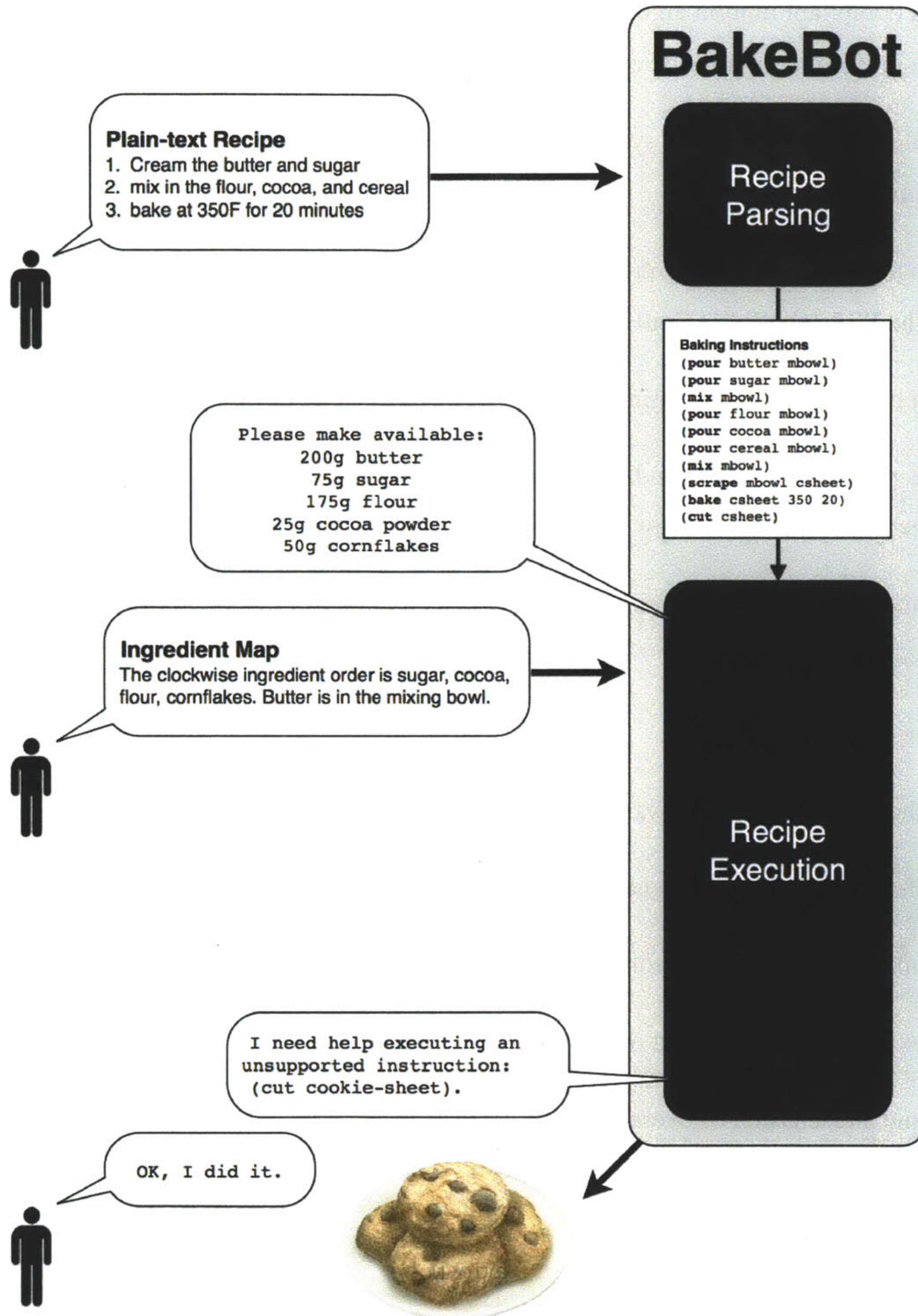


Figure 1-2: The human interaction with the BakeBot system for recipe execution. Provided with the plain-text recipe, BakeBot infers *baking instructions*. BakeBot asks its human partner to supply the required measured ingredients, the human partner arranges the ingredients on the worktable and tells BakeBot their relative positions. If an unsupported *baking instruction* is reached, BakeBot asks its human partner for help executing the instruction. The end result of the system is the baked cookies.

Chapter 2

System Setup

2.1 Hardware

BakeBot uses the Willow Garage PR2 humanoid manipulation platform with some modifications that protected it from food particles and enabled it to mix a batter with a spatula.

2.1.1 PR2

The PR2 was developed as a common open platform for robotics software research and development and was made available to MIT CSAIL through the PR2 Beta Program. The platform utilizes an omnidirectional wheeled base which enables it to access any area compliant with the Americans with Disabilities Act. The PR2 has two 7-degree-of-freedom torque controlled arms, providing an active workspace similar to that of an average adult human. Each arm terminates with a single degree-of-freedom parallel jaw gripper for object manipulation.

The sensor suite on the robot includes full joint state information, two pairs of stereo cameras, a tilting planar laser range finder, and a high resolution camera. The system is controlled by a pair of onboard computers, one of which controls the actuators over a gigabit-LAN network in a 1kHz real-time loop.

2.1.2 Hardware Modifications

Protective Covering

The PR2 is not designed to be protected from liquids and loose solids in the environment. In order to work with food, a protective cover was designed to protect the platform from potential spills. Figure 1-1 shows the PR2 wearing its protective garment.

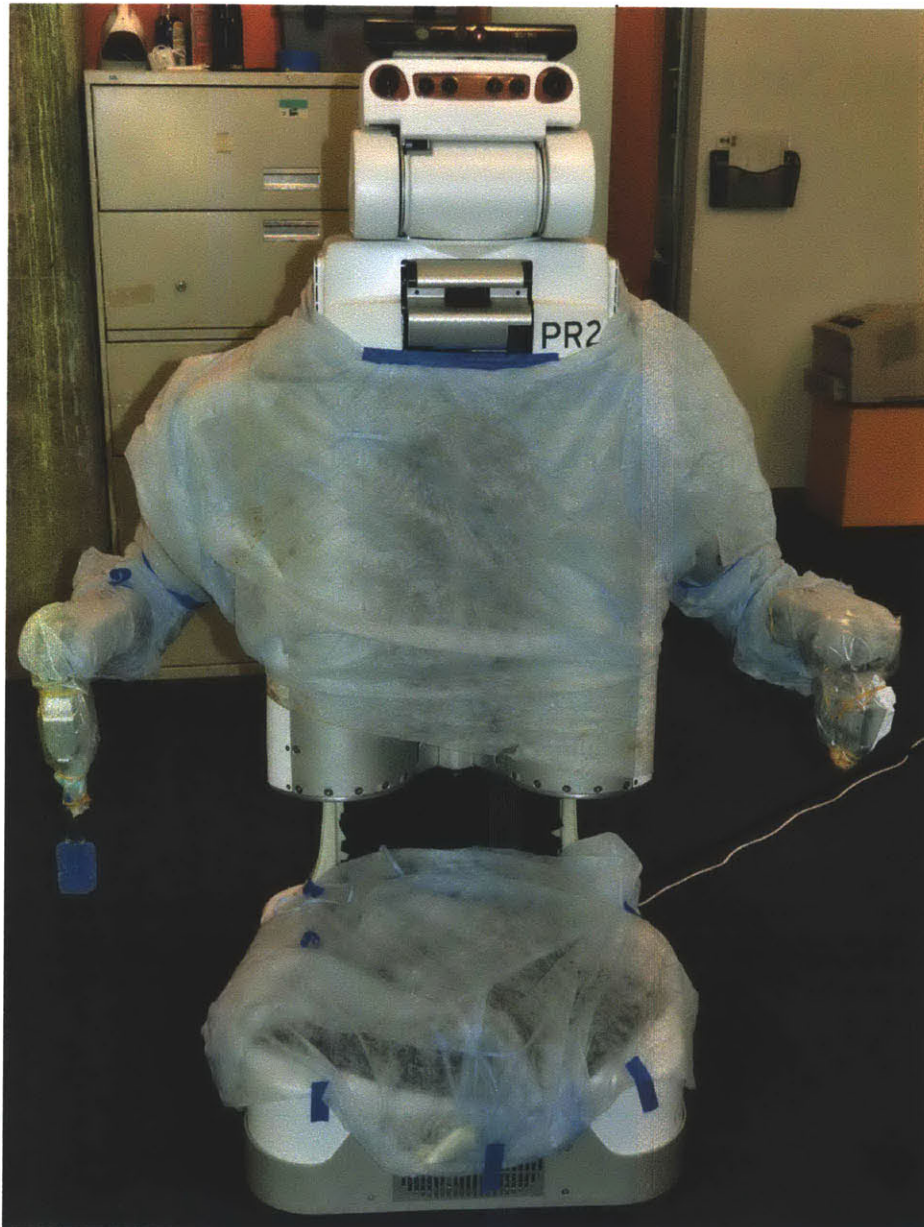


Figure 2-1: The PR2 in its protective covering with attached spatula standing in the *standard configuration* pose.

The protective covering was designed out of hospital surgical gowns, which are splash resistant and breathable (which was important to prevent controller boards and motors from overheating). The splash resistance was tested by running samples under the faucet and was found to be adequate for splash protection and spills that are not allowed to soak through the fabric (if a liquid spill should occur it was determined to remove the fabric from the robot to prevent soaking). Sensitive areas of the robot that faced the possibility of liquid spilling or soaking through the cloth, such as the manipulators and the top of the PR2 base, were covered with plastic sheeting to prevent soaked liquids from damaging the platform.

Care was taken to retain the PR2's range of motion when covered. To accomplish this, the gowns were tailored and segmented, with each arm segment only slightly overlapping neighboring segments, allowing for full rotation of the arm joints. The garments were affixed to the robot with rubber bands, which held the cloth tight to the arm segments, and blue painter's tape which enabled the cloth to be extended past the joints without constricting rotation and to be attached to the torso without leaving a sticky reside upon removal.

It took on average 30 minutes to dress the robot in the protective covering and 15 minutes to remove the covering. Section A.3.1 shows the process for covering the PR2.

Spatula Attachment

Grasp and manipulation planning to utilize a standard rubber spatula for mixing ingredients together was outside the scope of this thesis. It was decided to rigidly affix the spatula to the gripper to eliminate uncertainty about grasp planning and grasp strength during its use.

The spatula is attached by removing the finger pads from the right gripper (which on the model at CSAIL was not instrumented with sensor pads) and passing bolts through the mounting holes on one exposed finger, through the spatula inside the gripper, and out the mounting holes on the other finger. Figure 2-2 shows this mounting configuration and Section A.3.1 shows the attachment process in more detail. This method of attachment was chosen because it constrains the spatula, requires minimal setup, does not necessitate any permanent change to the PR2, and could be modified to enable the spatula to drop out of the gripper when no longer needed (though this functionality was never used).

Figure 2-2: The spatula attachment to the right end effector.

2.2 Environment Setup

BakeBot cooks in a kitchen environment consisting of two work surfaces, one for preparation and another to support a standard toaster oven. Figure 2-3 shows the kitchen layout. Both tables are immobile during the test, and constant distances are maintained between the robot's starting location, the preparation surface, and the oven. This eliminates the need for localization and navigation, neither of which were the focus of this thesis. Such capabilities have been demonstrated on the PR2 and we envision future generations of the BakeBot system to utilize them.

To execute a recipe, ingredients, the mixing bowl, and a cookie sheet are placed on the preparation table. The ingredient and mixing bowls are from the same set [2],they are made of melamine and are brightly colored to enable ingredient identification based on bowl color. Each bowl is of a novel diameter, enabling an alternative identification technique (which proved unnecessary). The "cookie sheet" is a standard 9" nonstick metal pie pan. It is small enough to be easily manipulated by the rim with a single gripper.

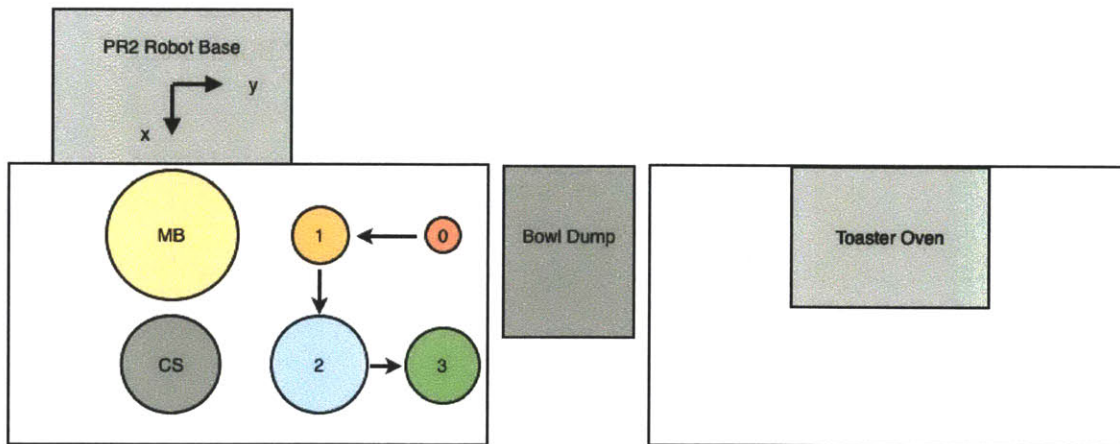


Figure 2-3: The environment setup for the baking task.
The environment setup for the baking task. The absolute positions of the bowls on the table surface is not enforced as long as a minimum distance between the objects is maintained. The ingredient bowls, numbered 0-3 are to the right of the mixing bowl (MB) and cookie sheet (CS).

2.3 Software Substrate

BakeBot as a software system was designed to run within the PR2's ROS environment and on top of a considerable software substrate responsible for updating system state, executing joint trajectories, inverse kinematic planning, object perception and manipulation.

The lowest level of software manages the sensor and actuator streams. These ROS topics are connected to nodes that integrate robot state information and control the manipulators. ROS nodes managing interpolated inverse kinematic trajectories are the next level of the software. The inverse kinematic planner used in the ROS system for BakeBot was the sample-based Open Motion Planning Library (OMPL) planner. The next level of software includes the grasp planners, Cartesian controllers, object manipulation planners, and laser point-cloud clustering nodes. Above those are nodes using the point-clusters that perform: object database connection management, tabletop segmentation, object recognition, and collision map processing. The highest level are integrated pick and place scripts that combine many of the lower features into a set of function calls that can be used to perform tabletop manipulation actions. These scripts comprise the bulk of the `pick_and_place_manager` which was modified for use by the BakeBot system.

Chapter 3

Key Solution Components

3.1 Object Recognition

BakeBot utilizes object recognition to locate and identify the ingredients, the mixing bowl, and the cookie sheet on the preparation table. The primary challenges overcome by the BakeBot object recognition were:

- detecting objects across a broad tabletop beyond what can be seen by a single sensor reading,
- and identifying objects and mapping them to the expected ingredient and tool set.

3.1.1 Object Detection

To detect objects across the broad tabletop a combination of local detections were taken and merged to create a single representation of objects in the workspace.

Local Object Detection

We utilized the object detection capabilities of the ROS tabletop manipulation pipeline to detect the bowls and cookie sheet on the table surface. The object detection system combines information gathered from the stereo cameras and the tilt-scan laser to find point clouds of interest above a detected table surface. The detection system can match point

clouds against a database of known objects to estimate a 3D model of the object. This database proved unreliable for the cooking task, as ingredients in the bowls interfered with the 3D model fitting, resulting in poor quality bowl grasps. Thus all object detection was based on the objects' raw point clouds above the table surface and their corresponding bounding boxes. This negatively affected the radial accuracy of the detections (bowl radius estimation had an average radius error of ± 1cm) but resulted in consistent detections.

3.1.2 Broad Tabletop Object Detection

We designed Algorithm 1 to detect and grasp objects across the entire table surface. This algorithm combines the results of many object detections across the table surface using a distance heuristic to filter duplicates. We implemented the algorithm in a `Broad Tabletop Object Manager` that combines objects from multiple detections and provides software handles that enable grasp planning for any object in the combined detection space. Figure 3-1 shows bounding boxes for eight detected bowls on the table. The `Broad Tabletop Object Manager` rests on top of the ROS manipulation layer and provides a centralized source of object information. It can save objects in the workspace to a file and load them from the file at runtime. This facilitated simulation and eliminated unnecessary broad tabletop detections during system debugging.

3.1.3 Object Identification

Given a set of object point clouds in the workspace, the task is then to determine which point cloud corresponds to which item expected on the tabletop: ingredients in bowls, the mixing bowl, and the cookie sheet.

HSB Thresholding

We integrated the Distributed Robotics Lab's existing HSB thresholding framework [24] to differentiate objects based on color. We did not find this framework useful in differentiating between the ingredients themselves, i.e. between flour and sugar. It was effective, however, in identifying the bowls based on their color. We thresholded based on Hue and Saturation.

Data: Objects on the table surface T, detected objects D, minimum object separation r_{min}, object list K
Result: K^*, the list of detected objects on T
Divide the surface T into a grid G;
for *every gridpoint g on G* **do**
    tabletop detection centered at g;
    for *every detected object d* **do**
        for *every object k in K* **do**
            calculate the distance r between d and k;
            if *r is greater than* r_{min} **then**
                add d to K;
            else
                calculate the volumes v_d and v_k;
                if v_d *is less than* v_k **then**
                    remove k from K and *add d* to K;
                end
            end
        end
    end
end
for *every object k in K* **do**
    tabletop detection centered at k;
    add object closest to midpoint of k to K^*
end

Algorithm 1: BROAD TABLETOP DETECTION

We found the Brightness value to be too dependent on lighting in the room and unreliable for segmentation. The blue bowl could not be differentiated from the blue carpet in the room. This was the only bowl that could not be identified, thus the point cloud that was not identified was always the blue bowl (identifying it). Figure 3-2 shows the segmentation steps of the HSB framework.

While HSB thresholding was effective in differentiating between the bowls (and thus the ingredients contained therein) during simple experiments, its over-sensitivity to variations in lighting in the testing room (which had several large windows) prevented it from being fully integrated into the BakeBot system. Implementation of auto-calibration could possibly improve performance, though doing so was outside the scope of this project.

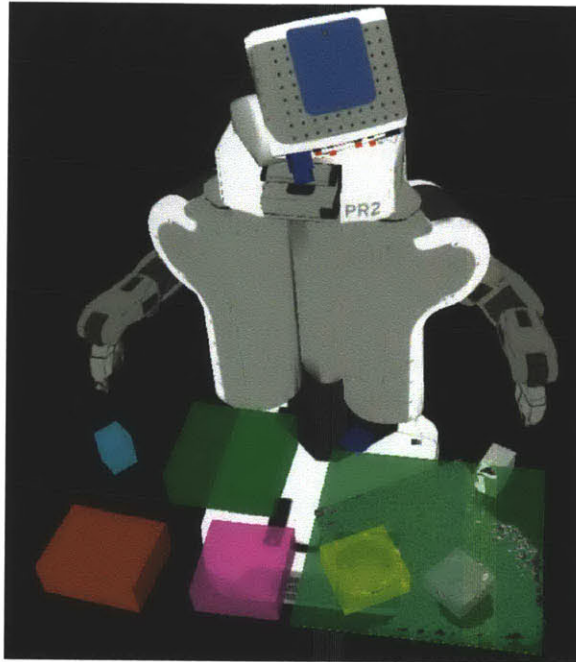


Figure 3-1: A screenshot of the PR2 visualizer showing the bounding boxes for eight objects on the work surface. The textured rectangle on the right of the table is the point cloud resulting from the most recent stereo detection and is representative of the detection space of a single scan.

Table 3.1: A mapping between the bowl order for two recipes (when objects on the table, excluding the mixing bowl and cookie sheet, are ordered counter-clockwise starting at the bowl with the smallest X and Y position values in the robot base frame) and ingredient name.

Ingredient Order	**Chocolate Afghans**	**Quick 'N Easy Sugar Cookies**
0	Sugar	Sugar
1	Cocoa	Salt
2	Flour	Flour
3	Rice Krispies	Baking Powder
Mixing Bowl	Butter	Eggs, Vegetable Oil, and Vanilla

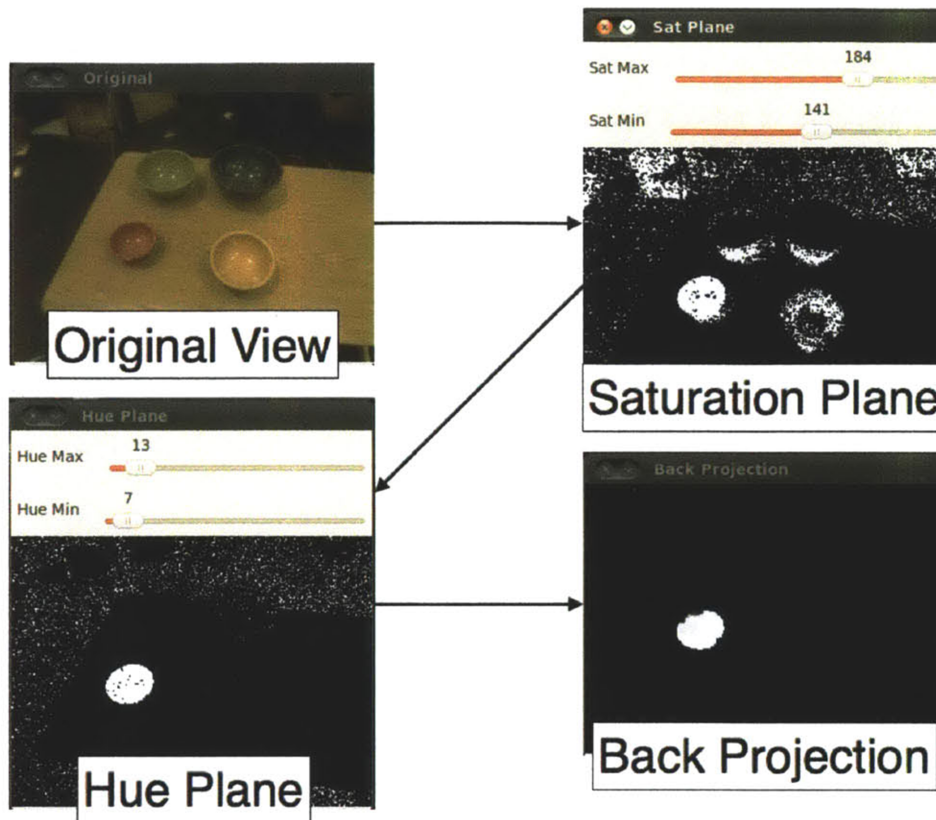


Figure 3-2: The segmentation steps for the HSB framework.

Relative Position Lookup

Ingredients, the mixing bowl, and the cookie sheet were ultimately identified from the point cloud set using their relative positions to one another. For example, the system was configured to take advantage of the fact that the two right most objects on the table were the mixing bowl (closer to the robot) and the cookie sheet (further from the robot). The other detected objects on the table were sampled in a counter-clockwise fashion, iterating through them starting with the object with the smallest X and Y position values in the robot base frame (the object to the near right). Figure 2-3 shows the order of ingredient collection. A map between the object order in the counter-clockwise sample and its ingredient name was stored in memory. This required whomever setup the ingredients *mise en place* on the table to input their relative positions to each other into the system. Table 3.1 shows the mapping between objects on the table and the ingredients for two recipes, Chocolate Afghans and

Quick 'N Easy Sugar Cookies. Some ingredients were stored in the mixing bowl at the start of the recipe to minimize unnecessary pouring. Ingredients that posed the greatest spill risk (such as beaten eggs or melted butter) were placed in the mixing bowl.

3.2 Compliant End Effector Control

In the process of cooking BakeBot must work with many different objects and surfaces in the environment. For tasks such as mixing and scraping, modeling the batter and its interactions with the spatula and the inner surface of the mixing bowl is an infeasible task. It was desirable to specify these actions generally in terms of forces applied to the batter rather than strictly in terms of the movement of the spatula in the Cartesian workspace. Instead of attempting to estimate a spatula path through the workspace that traced the inner circumference of the bowl, we desired a control policy that would push the spatula against the inner circumference as it worked its way around the bowl, regardless of the exact bowl size and position.

We designed a hybrid force/compliance controller for the PR2 manipulators to accomplish this. This controller, `ee_cart_imped` (for End Effector Cartesian Impedance Control), enabled one to specify Cartesian forces and impedances for each end effector. It enables one to generate and follow trajectories of forces and impedances through the Cartesian workspace.

3.2.1 Point Compliance

`ee_cart_imped` is a Jacobian-transpose endpoint force controller. Given a vector of desired forces and wrenches in the workspace

$$\vec{F} = \begin{pmatrix} F_x \\ F_y \\ F_z \\ \tau_x \\ \tau_y \\ \tau_z \end{pmatrix}, \tag{3.1}$$

the vector of necessary joint torques, $\vec{\tau}$, to accomplish $\vec{F}$ is

$$\vec{\tau} = \mathbb{J}^T \vec{F} \tag{3.2}$$

where $\mathbb{J}^T$ is the transpose of the instantaneous joint Jacobian matrix [4].

To accomplish compliant control of the end effector around a point $\vec{x}$ in the Cartesian workspace we can define a stiffness matrix $\mathbb{K}_p$ and find the restoring forces in response to small displacements in $\vec{x}$:

$$F = -\mathbb{K}_p \delta \vec{x}. \tag{3.3}$$

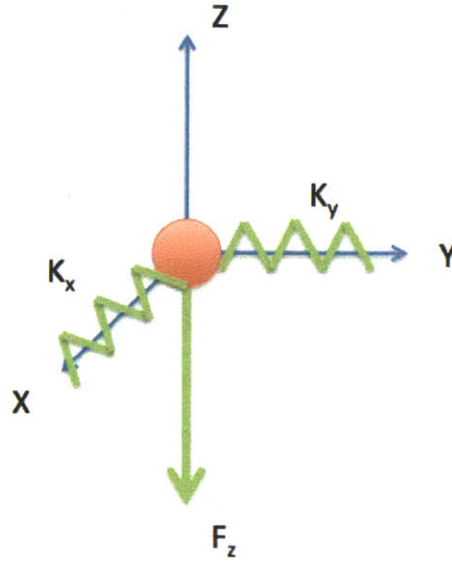


Figure 3-3: A point in the workspace with independent stiffnesses in x and y and a desired force along the negative z axis. Wrenches and rotational stiffnesses are not shown.

It is often desirable to specify compliance in specific degrees of freedom and forces in other degrees of freedom. For example, Figure 3-3 shows a point in the workspace with independent stiffnesses in x and y and a desired force along the negative z axis. This could represent a point in a trajectory that pulls a spatula around a bowl while maintaining a constant force against the bottom of the bowl. Specifying both a desired compliance and a desired force at a point would overconstrain the system so we define compliance selection matrix $\mathbb{A}$ and force selection matrix $\mathbb{B}$. $\mathbb{A}$ and $\mathbb{B}$ are both diagonal with entries of either 0 or 1 along the diagonal such that

$$\mathbb{A} + \mathbb{B} = \mathbb{I}. \tag{3.4}$$

It is then possible to construct a joint torque matrix $\vec{\tau}$ that achieves the desired forces and

stiffnesses around $\vec{x}$

$$\vec{\tau} = \mathbb{A}\mathbb{J}^T\vec{F} - \mathbb{B}\mathbb{K}_p\delta\vec{x}. \tag{3.5}$$

3.2.2 Trajectory Interpolation

We can specify trajectories of desired stiffnesses and forces in order to accomplish complex motions with the force-compliant controller. Each point in the trajectory specifies a force or stiffness value for each of the six degrees of freedom and a time from the start of the trajectory at which the point should be reached. Desired forces or stiffnesses for each degree of freedom are linearly interpolated (in the workspace) between consecutive points in the trajectory. Interpolation is not performed when a degree of freedom switches from force-control to stiffness-control between two consecutive points, rather the force value is repeated until the second point is reached, then the stiffness value is used.

3.2.3 ROS Implementation

We implemented the `ee_cart_imped` controller inside of the PR2's realtime control loop. Once the controller is enabled, one accesses the controller through a ROS Action interface. The controller receives trajectories of `StiffPoints` which specify the time and the desired stiffness around, or force at, the point. The controller executes the trajectories as discussed and returns the status of the trajectory back to the client.

The controller is currently open-loop as there is no method available to measure the applied force at the PR2 end effector. Measured joint currents (thus torques) are noisy and do not reflect losses due to joint friction. Anomalous behavior occurs when the desired force to be applied by the controller is not enough to overcome the static friction in the arm joints. Generally speaking, the controller provides accurate force directions (assuming joint friction is overcome) but less accurate force magnitudes. A force plate at the PR2 wrist, in discussion with Willow Garage, would enable us to close the loop on the controller.

We have released the controller to the ROS community.

3.2.4 Discussion

Force-compliant control with a mobile manipulator does not require any inverse kinematic planning and is robust to variations in the environment and uncertainty in sensing. These control techniques proved to be extremely useful on the PR2 platform. The BakeBot system uses the `ee_cart_imped` controller to mix, clean the spatula, scrape, and open the oven. Other student projects at MIT have used the controller to clean a table, write and erase with a pencil, open cabinets, and sweep.

The workspace for the controller is limited, as all joint-limits should be avoided. Closing the loop around the controller would widen the workspace by allowing other joints to compensate for joints at their limits and would enable fine control over the magnitude of applied force.

3.3 Mixing and Scraping

We want BakeBot to interact with the cookie batter in as human a way as possible. This presented the greatest manipulation challenge but rewarded us with a high degree of versatility. We utilized the `ee_cart_imped` force-compliant controller to implement trajectories to mix the batter together and to scrape it from the mixing bowl.

3.3.1 Mixing Trajectories

Several different kinds of mixing trajectories were implemented in order to achieve adequate mixing performance. These trajectories were parametrized at runtime based on the detected diameter and position of the mixing bowl in the workspace. All mixing trajectories utilized the force-compliant controller discussed in Section . Due to limitations in the kinematic model of the arm, the controller stabilized the center of the gripper, not the endpoint of the spoon. The trajectories appear shifted relative to the bowl because of this offset. As the spoon was usually kept horizontal or vertical (with a high rotational stiffness) this linear offset is of little consequence.

Plunging the Spoon

The spoon must be lowered into the mixing bowl before mixing can commence. To accomplish this, the spoon is held over the bowl, then a force trajectory is executed. The trajectory keeps the spoon in a constant orientation (pointing downward into the bowl) while a downward force is applied, pushing the spoon into the center of the bowl. The downward force is slowly increased, ensuring the spoon reaches the bottom of the bowl (through whatever batter may be in the way) but minimizing the chances that it will violently hit the bowl from above.

Circular Mixing

Circular mixing scrapes the spatula around the inner circumference of the mixing bowl, combining the ingredients together by moving them around the perimeter of the bowl.

Initial experimentation with circular mixing focused on following a trajectory of force vectors tangent to the bowl at each point in the trajectory. This produced adequate mixing but was not very reliable. If the spoon were to jump out of the bowl for any reason (if the ingredients were to pile up, for example) the force vector desired at the endpoint would cause the arm to swing around wildly. This failure mode could not be elegantly recovered from and would result in an aborted mix.

Instead of a moving force vector, the circular mixing routine instead follows a circular position/stiffness trajectory that moves the end effector around a circumference larger than the mixing bowl's and a bit over the mixing bowl. The controller stabilizes the end effector to this trajectory, resulting in a movement in which the spatula is pushed against the wall of the bowl, with a force proportional to the difference between the radius of the bowl and the radius of the desired trajectory, in a circle around the bowl's circumference. Figure 3-4(a) shows the path of the center of the end effector during circular mixing. This trajectory is robust to the spatula popping out of the bowl. When the spatula slips out of the bowl it continues around the trajectory (being larger than the radius of the bowl) without wildly swinging. This can be easily detected and subsequent actions can put the spatula back into the bowl.

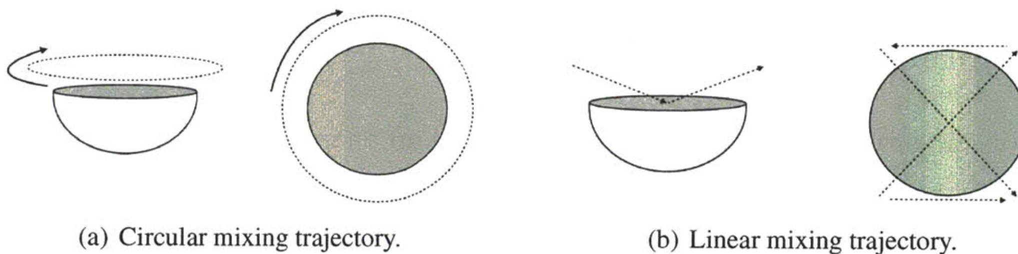
The stiffnesses around the positions on the trajectory were chosen to keep the spatula against the sides of the bowl during the mixing process. The controller bandwidth, backlash in the arm, and the density of points on the generated trajectory caused the end effector to jitter during the mixing action. While we initially sought to eliminate this jitter, we found that it was useful in keeping the spatula from getting stuck against the side of the bowl and kept it as it was.

Circular mixing tends to force the batter against the manipulator holding the mixing bowl, resulting in a pile of batter near the hand. Periodically switching the direction of circular mixing helps reduce the size of the batter building but ultimately results in smaller amounts pressed against both sides of the manipulator. This pileup is dealt with by switching the sides of the mixing bowl held by the manipulator. By releasing the grasp on the mixing bowl, switching the grasp to the other side of the mixing bowl, and translating and rotating the mixing bowl so that the manipulator is back on the left side of the bowl, the

buildup can be moved to an area of the inner circumference that allows it to be more easily broken up by the circular mixing trajectory.

Linear Mixing

Linear mixing scrapes the spatula through the center of the mixing bowl to incorporate ingredients that were not reached by the circular mixing trajectory. It is accomplished by generating a series of position/stiffness waypoints alternately inside and outside the bowl. These waypoints are strung together and interpolated, creating a trajectory that moves the spatula back and forth through the center of the bowl along opposite diagonals. Figure 3-4(b) shows the path of the center of the end effector during linear mixing. This position/stiffness trajectory moves the batter through the center of the bowl and evens out irregularities produced by circular mixing.



(a) Circular mixing trajectory.

(b) Linear mixing trajectory.

Figure 3-4: Top and side views of the compliant trajectories followed by the center of the end effector during circular 3-4(a) and linear 3-4(b) mixing routines.

Rim Plunging

Rim plunging pushes the spatula down from above around the inner circumference of the mixing bowl. This pushes down batter that has built up against the sides of the bowl and helps to remove buildup near the gripper that holds the mixing bowl. Rim plunging utilizes the same motion as the regular spatula plunge but moves it to the four cardinal positions around the bowl circumference (twelve o'clock, three o'clock, six o'clock, and nine o'clock). The spatula is rotated before plunging to ensure that it is tangent to the side of the bowl, maximizing the contact area with stuck batter.

This motion was necessary to dislodge stuck batter before scraping the contents of the

mixing bowl onto the cookie sheet. The inverse kinematic planning for spatula movement at each of the cardinal positions was complicated by the cluttered workspace and the presence of the second manipulator holding the bowl. Many planning attempts had to be made before successful trajectories could be executed to move the spatula into position prior to plunging it downward. This made the action very slow, ultimately it was not performed during all mixing operations, only during the final mix before scraping.

Whisk Mixing

We attempted whisk mixing, in which the wrist with the spatula spins as fast as possible to mimic a hand mixer. Unfortunately we were not able to get the wrist to spin fast enough to effectively mix anything with this technique. We tried joint velocity control and torque control at the wrist joint but were unable to achieve the desired behavior.

Cleaning the Spatula

Batter in the mixing bowl had a tendency to build up on each face of the spatula. This would make mixing less effective, as the batter on the spatula would seldom incorporate itself back into the main mixture. Batter that was stuck on the spatula would drop onto the floor when the spatula was retreated to the side of the robot after mixing, making a mess on the floor.

Each side of the spatula was wiped against the rim of the bowl to remove build up batter. This was done in a similar fashion to what a human would do. The spatula was:

1. held parallel to the table so the head of the spatula is over the mixing bowl,
2. lowered with a constant force, contacting head of the spatula with the rim of the bowl,
3. compliantly pulled across the lip of the bowl to dislodge batter,
4. lifted off of the lip and rotated around its long axis to repeat the process on the other side of the spatula.

3.3.2 Mixing Experimentation

Mixing required considerable trial and error to generate a set of trajectories that were effective and robust. Initial mixing attempts were performed with dry beans in order to test the controller with no risk to the robot (and no need to dress the robot in protective garments).

Tuning Trajectories

Once overall mixing techniques were proofed with beans, real ingredients were used to tune the mixing trajectories. The trajectories were tuned with the ingredients for the "Chocolate Afghan" biscuit recipe [1] and were later tested with the ingredients for other similar recipes. Ingredient proportions were varied wildly in order to get a set of robust mixing parameters.

A custom graphical user interface was developed to enable quick testing of mixing trajectories. Figure 3-5 shows the interface with default values preloaded. This saved a considerable amount of time by eliminating the need for the code to be reloaded between every test.

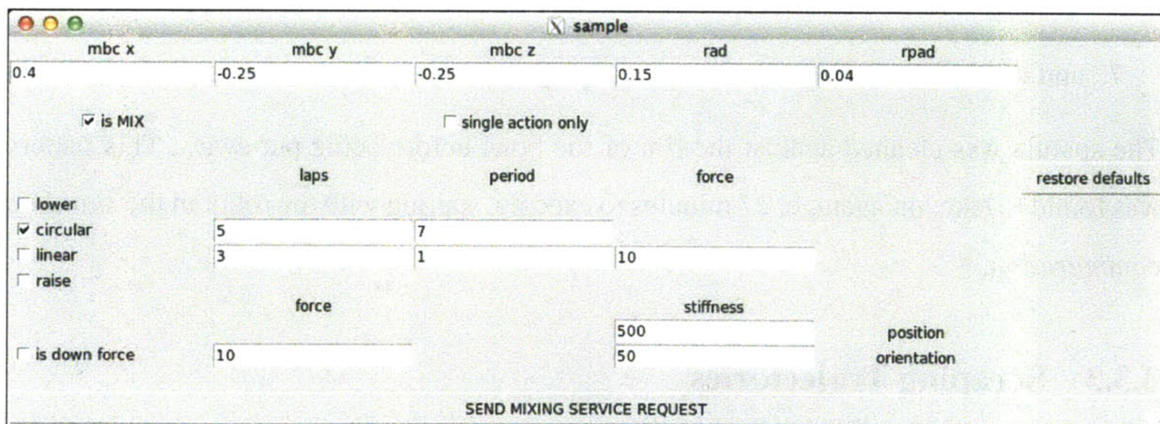


Figure 3-5: A screenshot of the mixing testing interface that was used to refine the mixing trajectories.

Determining Mixing Completeness

We attempted to use joint torques during mixing as a measure of mixing completeness, reasoning that the batter would change consistency in proportion to how well it was mixed

together. Figure 3-6 shows a record of the joint torques during one lap of circular mixing. Ultimately no heuristic was found to estimate mixing completeness from joint torques.

Visual methods were considered to determine mixing completeness, but were quickly determined to be outside of the scope of this project.

We decided to over-mix the batter (which is often acceptable, the notable exception being pancakes) in order to guarantee that it is mixed. A combination of mixing trajectories was chosen that ensured that contents of the mixing bowl are well combined. The mixing pattern found to be very effective was:

1. 5 laps of circular mixing
2. 3 iterations of linear mixing
3. 5 laps of circular mixing
4. switching the side of the mixing bowl held by the gripper
5. 5 laps of circular mixing
6. 3 laps of linear mixing
7. and if this was the last mix before scraping was to be performed, rim plunging.

The spatula was cleaned against the rim of the bowl before being put away. This pattern was found to take, on average, 27 minutes to execute, starting with the robot in the *standard configuration*.

3.3.3 Scraping Trajectories

The goal of scraping is to dislodge any stuck batter remaining in the mixing bowl after pouring onto the cookie sheet. Scooping out individual cookies was determined to be outside the scope of this thesis, thus one giant cookie is created in the cookie sheet.

Scraping the contents of the mixing bowl into the cookie sheet utilized similar motions as regular mixing. Generally speaking, these motions were modified by switching their Y and Z axes.

Scraping was complicated by several factors:

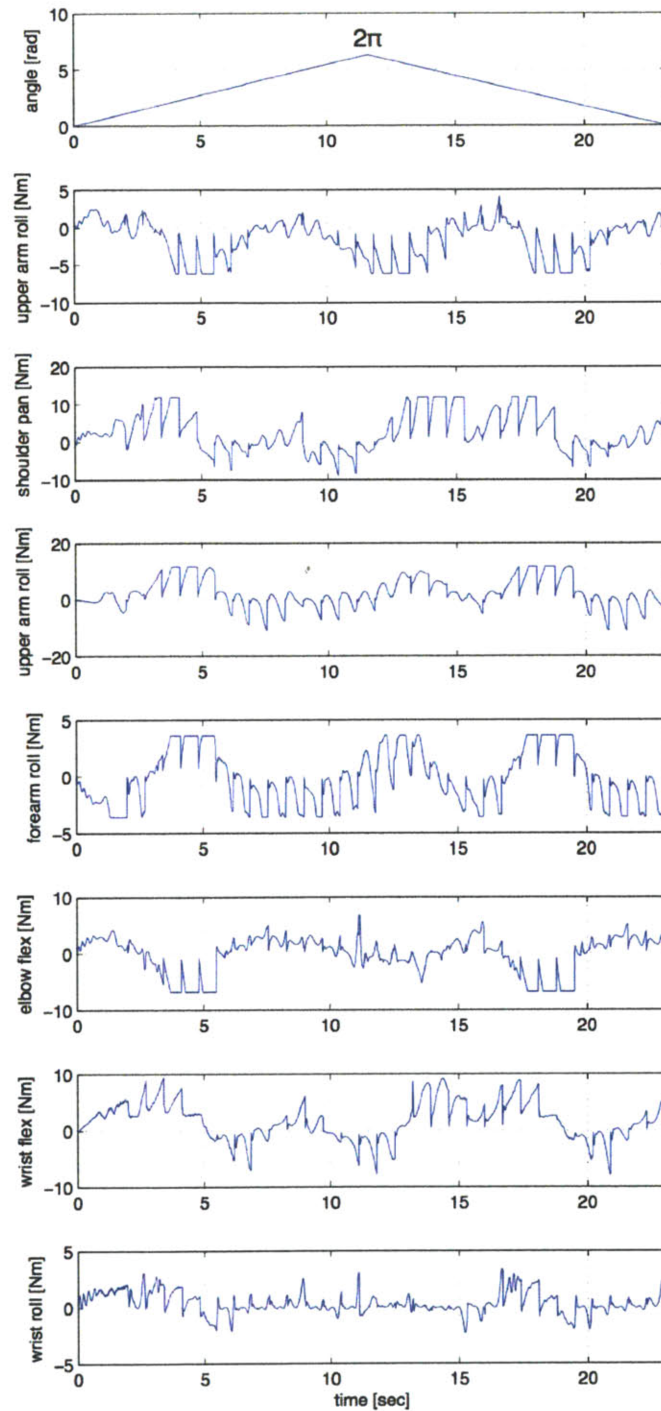


Figure 3-6: Joint torques during execution of the circular mixing trajectory in a mixing bowl full of cookie batter. The spatula angle is plotted in the top plot, the trajectory moves the spatula around the bowl once, then reverses its direction and ends with the spatula in the starting position.

- The bowl is entirely supported by the opposite manipulator. The manipulator is not completely rigid, thus the bowl moves whenever it comes in contact with the spatula. Additionally, the manipulator cannot effectively resist bowl twisting moments, thus any asymmetry during scraping has the potential to dramatically twist the bowl in the manipulator.

- The batter tended to stick to the inside of the bowl. Rim plunging during the final mix mitigated this, but higher forces than used for mixing were required during scraping to dislodge stuck batter.

- Several manipulation steps preceded the actual scraping maneuver: detecting and grabbing the mixing bowl, detecting the cookie sheet, moving the mixing bowl over the cookie sheet, and pouring the mixing bowl (tilting it into position to be scraped). A considerable amount of work had to be done to setup the scraping code so that different scraping techniques could be tested without having to redo all of the pre-requisite manipulation.

Two scraping trajectories were used, analogous to plunging the spatula and circular mixing.

Plunging the Spatula

This motion is immediately analogous to its mixing equivalent. The pre-plunge position, rather than being based on the detected position of the mixing bowl, is computed from the known position of the mixing bowl in the other manipulator. The spatula is held at about a 45° angle above the horizontal, in order to aim at (and hopefully dislodge) batter stuck near the manipulator holding the bowl.

Circular Scraping

This motion moves the spatula from the position near the top of the upturned mixing bowl (where plunging leaves it) around the circumference of the bowl to the bottom of the mixing bowl. At the bottom of the mixing bowl the spatula is pulled backwards and out of the

bowl, forcing batter onto the cookie sheet. The spatula has a tendency to overshoot and press downwards into the cookie sheet, this is advantageous as it produces a flat cookie. After scraping one side of the mixing bowl, the spatula is replunged and the process is repeated on the other side of the mixing bowl.

3.3.4 Scraping Experimentation

Like mixing, scraping required trial and error to generate effective and robust trajectories. As mentioned previously, the pre-manipulation required to setup the scraping action slowed development considerably and much time was spent setting up the codebase to facilitate testing of new scraping trajectories.

The mixing testing interface was modified with an option to test scraping trajectories. A debugging mode was implemented in the software framework. This mode logged the pose of the manipulators immediately before scraping commenced, allowing the system to restore itself to this point again and again while testing new scraping trajectories.

Ultimately the refinement of the scraping was two-part: improvements in mixing (such as rim-plunging and rotating the mixing bowl) helped to prevent batter from sticking to the sides of the bowl, and techniques like plunging the spatula during scraping helped to dislodge batter from problem areas.

3.4 Recipe Parsing

The robot is initialized with the text of a recipe along with a set of ingredients arrayed in bowls on a table. Its goal is to infer an action sequence in the environment that corresponds to following the text of a recipe. The robot proceeds by segmenting the recipe into sentences based on punctuation. Then for each sentence, it infers an action sequence in the environment corresponding to the words in the sentence. After executing the action sequence successfully, the robot will have produced the appropriate dish, for example cookies, brownies, or meatloaf.

3.4.1 Baking Instruction Set

We defined a set of *baking instructions* that enable BakeBot to execute a variety of simple baking recipes:

- **pour(ingredient i)**: adds ingredient i to the mixing bowl
- **mix()**: mixes the mixing bowl
- **scrape()**: scrapes the batter out of the mixing bowl onto the cookie sheet
- **bake(time t)**: puts the cookie sheet in the oven, opening the oven t minutes later
- **preheat(temperature T)**: preheats the oven to T degrees Fahrenheit

3.4.2 State/Action Space for the Kitchen

We formally define a state-action space for the kitchen domain using primitive actions corresponding to the *baking instructions*. Within this state space, many specific action trajectories can be followed, yielding a variety of different dishes. In order to follow a specific recipe, we define a reward function based on the text of a recipe. The robot uses forward search to find the sequence of states and actions that maximizes rewards and executes it in the external world. Because the text of the recipe generates the reward function, this optimization corresponds to finding an action sequence that follows the recipe.

We define a state S_k as the collection of unused ingredients S_k^i in the workspace, the mixing bowl S_k^b and its contents, the cookie sheet S_k^s and its contents, and a toaster oven S_k^o and its temperature and contents. Given any state S_k, we define $actions(S_k)$ to be the set of available actions in that state:

- For each unpoured ingredient S_k^i, $pour(S_k^i, S_k^b) \in actions(S_k)$.
- If $nonempty(S_k^b)$ then $mix(S_k^b) \in actions(S_k)$.
- If $nonempty(S_k^b)$ then $scrape(S_k^b, S_k^s) \in actions(S_k)$.
- If $empty(S_k^o)$ then $preheat(S_k^o) \in actions(S_k)$.
- If $empty(S_k^o) \wedge nonempty(S_k^s)$ then $bake(S_k^s) \in actions(S_k)$.

After executing an action such as *pour* in state S_k, the next state S_{k+1} contains one less unused ingredient, and the mixing bowl S_{k+1}^b contains the corresponding poured ingredient. Although the currently available actions do not always execute robustly on the real robot, they support a surprising array of recipes, ranging from brownies to meatloaf to salads. Over time we plan to increase the robustness of the existing actions and add additional ones such as chopping and whisking in order to increase the range of recipes that the robot supports.

Actions are executed on the physical robot using the ROS system as discussed in the remainder of this thesis.

3.4.3 Reward Function

In order to follow a recipe, the system takes as input the natural language text of a recipe and induces a reward function over the state/action space, then searches for the sequence of states and actions that maximizes the reward. The reward function is learned from a labeled dataset of recipes paired with the correct action sequence. Formally, a recipe consists of a list of sentences $d_1 \dots d_N$. For each sentence, the system infers a sequence of states $S_1 \dots S_K$

that maximizes the reward function R:

$$\underset{S_1 \ldots S_K}{\operatorname{argmax}} \sum_j R(d_j, S_m \ldots S_n) \tag{3.6}$$

We use a greedy algorithm to incrementally find a state sequence for each sentence d_i; in the future we plan to implement an HMM or CRF model to jointly optimize the action sequence across multiple sentences as in [17]. We define the reward function as a probability distribution parameterized using a log-linear model [7]. The model is trained from a corpus of recipes annotated with the correct action. The feature functions are bag-of-words features crossed with the presence of specific actions and arguments in the state sequence. For example, a feature would be "Mix" $\in d_j \wedge pour \in S_1 \ldots S_k$. The system takes positive examples from the annotations. To form negative examples, it finds action trajectories which result in an incorrect final state. The system then finds model parameters which maximize the likelihood of this training set using a gradient descent algorithm.

3.4.4 Experimentation

Our experimentation was two-fold: we passed recipes through our language processing system to create robot instruction sets and we executed the instruction sets on the physical robot system. The robot's workspace is initialized with a set of objects containing the ingredients and implements necessary to follow the recipe; the system does not parse the ingredients list.

We collected a dataset of 50 recipes from the internet, describing how to make simple dishes such as brownies, meat loaf, and peach cobbler. For each recipe, we formally specified the specific ingredients and implements necessary to follow the recipe. This initialization is given to the robot. Next, for each instruction in the recipe text, we annotated the sequence of primitive actions the robot should take in order to follow that instruction. We used 35 recipes from this corpus to train the model, and 15 recipes to test it. A sample recipe from the dataset appears in Fig. 3-7, together with the automatically inferred action sequence.

We performed a quantitative evaluation in simulation to assess the recipe processing

Recipe Text	Inferred Action Sequence
Afghan Biscuits 200g (7 oz) butter 75g (3 oz) sugar 175g (6 oz) flour 25g (1 oz) cocoa powder 50g cornflakes (or crushed weetbix)	
Soften butter.	$pour(butter, bowl); mix(bowl)$
Add sugar and beat to a cream.	$pour(sugar, bowl); mix(bowl)$
Add flour and cocoa.	$pour(flour, bowl); mix(bowl)$
Add cornflakes last.	$pour(cornflakes, bowl)$
Put spoonfuls on a greased oven tray.	$preheat(350)$
Bake about 15 minutes at 180°C (350°F).	$mix(bowl); scrape(bowl, pan); bake(pan, 20)$

Figure 3-7: Text from a recipe in our dataset, paired with the inferred action sequence for the robot.

system's ability at finding a plan to understand a variety of recipes. For this evaluation we assessed the system's performance at inferring an action sequence for specific sentences in the recipes. The 15 recipes in the test set contained a total of 92 individual sentences. Of these sentences, the system correctly inferred exactly correct plans for 49% of them. Many of the errors consist of omitting an ingredient, as in "Mix with hands - form small balls and flatted on ungreased pan." which referred to all ingredients in the recipe; for thsi command the system infered only $pour(sugar, bowl); mix(bowl)$ but excluded other ingredients such as butter and flour.

3.4.5 Discussion

The natural language component of the system shows promising performance at understanding individual instructions; it inferred exactly correct actions for many instructions, and many of the mistakes were due to missing one or two ingredients from aggregate phrases such as "Sift together the dry ingredients." However, the robot's physical and perceptual capabilities limited the scope of the recipe processing. For example, the system ignores a warning such as "Do not brown; do not over bake" because detecting the color of overcooked cookies through an oven door is an unsolved perceptual problem.

3.5 Dynamic State Machine Framework

Transitioning from a static finite state machine representation, in which all of the transitions between states are predefined, to a plan-based representation, in which the robot is given a goal state, assesses its environment, and plans a path of states to achieve its goal, required the design of a dynamic state machine framework.

Traditional finite state machines have predefined transitions between states. Figure 3-8 shows a standard finite state machine with three states: A, B, and C. Execution starts in state A and transitions based on the success or failure of each state's execution, ultimately either reaching the goal or a state representing failure.

We desired a planning-based system, in which the robot is given a goal and a set of states, assesses its environment, and plans a sequence of states to achieve the goal. Figure 3-11 shows such a system, a dynamic state machine. Analogous to the state machine shown in Figure 3-8, execution starts in state A. As shown, state A executes successfully, transitioning to state B. State B, however, fails to execute, triggering the system to generate a new plan. The planner assesses the environment, estimates the current state, and generates a new path to the goal. This new plan is then executed in similar fashion.

3.5.1 SMACH

The BakeBot static state machine implementation was coded using the SMACH ROS package. The SMACH (for 'state machine') ROS package is a task-level architecture that makes it easy to create hierarchical finite state machines using Python [3]. SMACH provides representations for individual states and for state machines. The package handles data transfer between the states, state transitions, and program flow. Data transfer is handled by an abstract `userdata` storage class that encapsulates the data, allowing (purposefully) limited access and mutability of data to the states. SMACH also provides a visualization interface that makes it easy to see state connections and visualize program flow.

All of the BakeBot motion primitives were represented as states or nested finite state machines within the SMACH framework. Our goal was to reuse as much of this code as possible when migrating to a planning-based dynamic state machine.

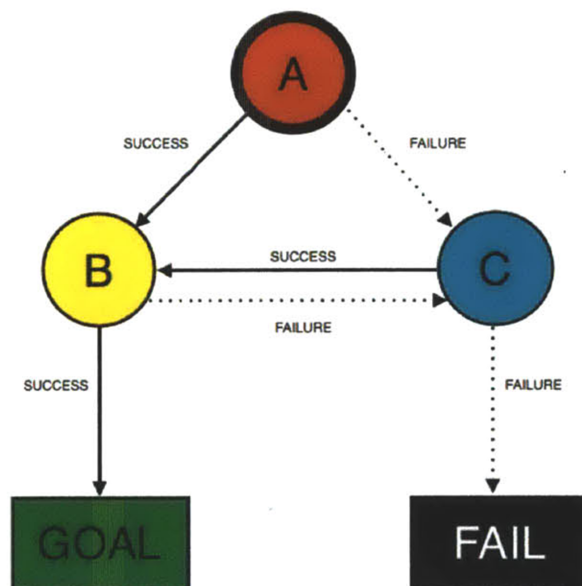


Figure 3-8: A standard finite state machine. Execution starts in state A and transitions based on the success or failure of each state execution.

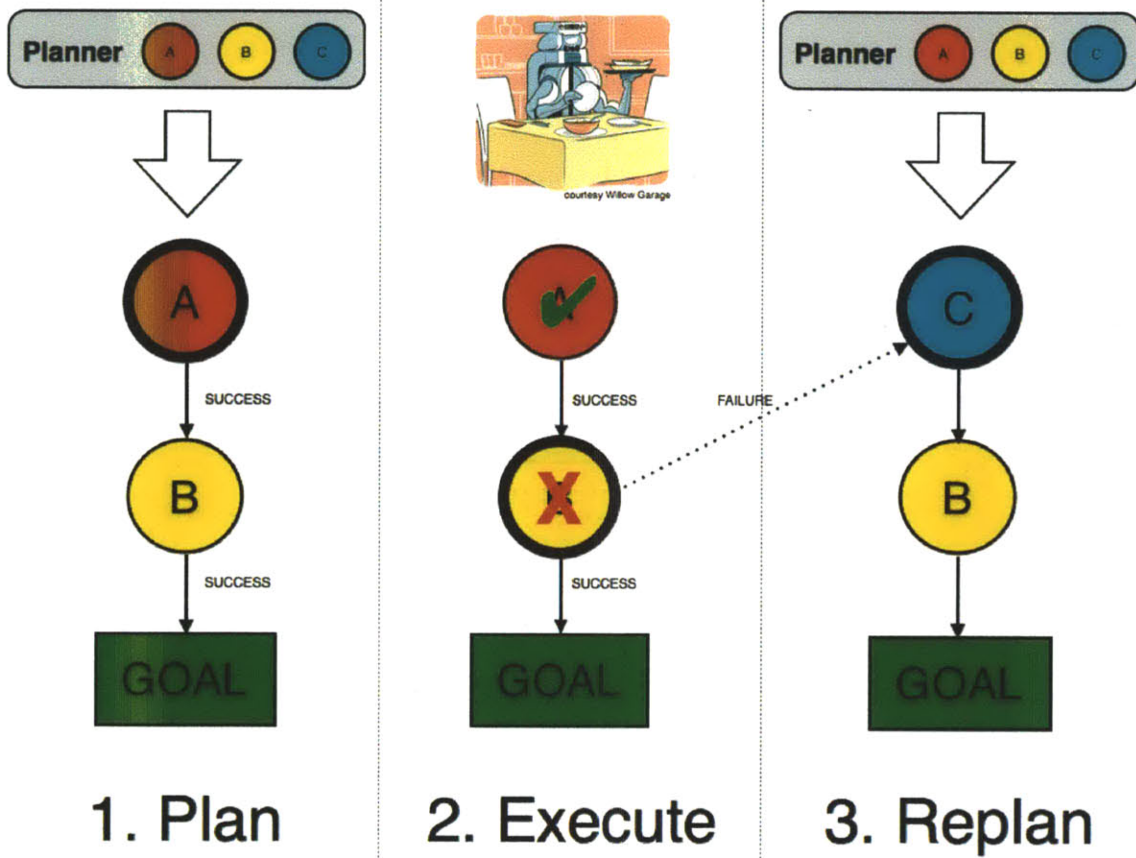


Figure 3-9: A dynamic state machine analogous to the static state machine in Figure 3-8. The planner, containing all available states, plans a path to the goal from the current estimated state. The plan is executed. If execution of a state results in success, the next state in the plan is executed. If execution of a state results in failure, a new plan is generated to reach the goal from the current estimated state.

3.5.2 Fast-Forward

Fast-Forward (FF) is a popular domain independent planning system developed by Jörg Hoffmann [13]. It can handle classical STRIPS planning tasks specified in the Planning Domain Definition Language (PDDL). We specified the baking task as a set of STRIPS planning tasks in PDDL and defined preconditions and effects for each BakeBot motion primitive. This enabled us to use FF to plan sequences of motion primitives to accomplish the baking task.

3.5.3 System Design

We designed `smachforward` (a word sandwich of SMACH and Fast-Forward), a system that uses `ff` to plan sequences of SMACH states to accomplish a goal. `smachforward` creates a dynamic state machine that can be integrated into a larger dynamic or static state machine, enabling hierarchical task execution within the existing SMACH framework, shown in Figure 3-10. The static finite state machine executes as it would normally. When the `smachforward` state (shown in grey) is reached, it initiates the `smachforward` system to plan and execute a sequence of motion primitives (SMACH states) to accomplish the goal. When execution of the motion primitives fails, the state either replans (based on limits on replanning set by the user) or fails, returning control to the static state machine. If execution of the motion primitive sequence succeeds, control is returned back to the static state machine.

Figure 3-11 shows the `smachforward` system architecture, which is contained within the `smachforward` state shown in grey in Figure 3-10. The State Estimator, PDDL Assembler, Fast-Forward Planner, Plan Assembler, and Executive comprise the `smachforward` system.

System Inputs

The `smachforward` system takes as input:

PDDL Goal a description of the goal state represented in PDDL

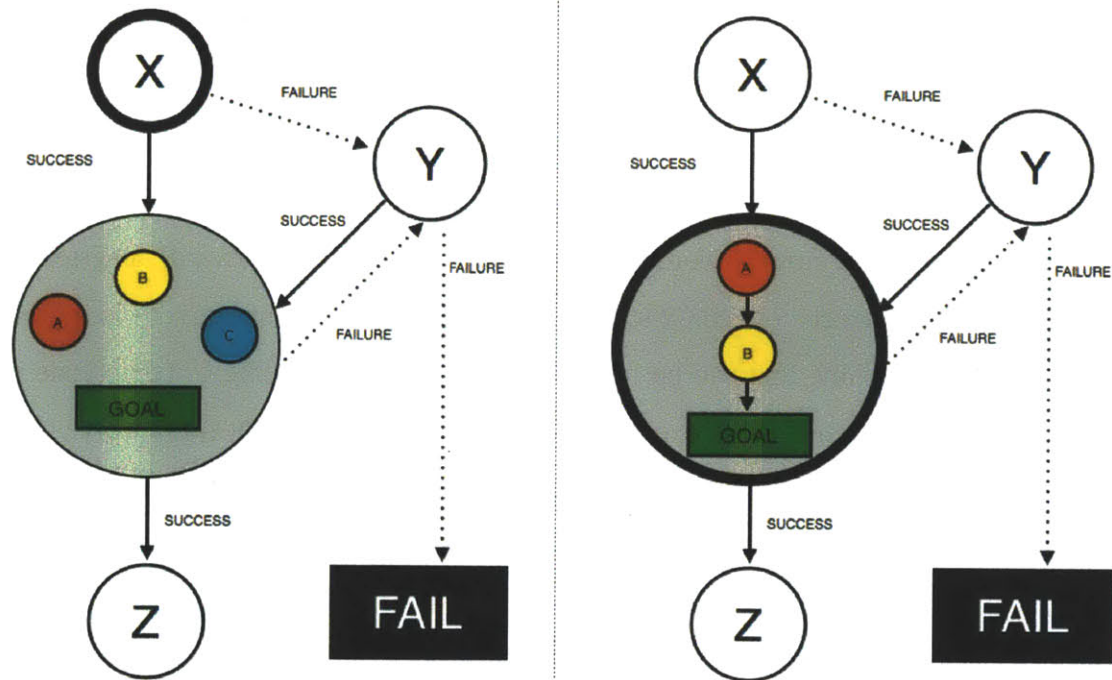


Figure 3-10: A static finite state machine with one state, represented in grey, holding a nested dynamic state machine. The state machine executes normally (shown on the left). When the nested `smachforward` state machine, in grey, is reached during execution (shown on the right), `smachforward` plans a sequence of motion primitives to accomplish the state's goal and then executes the sequence. Reaching the goal returns to the static state machine. Failure results in either a replan or return of control to the static state machine, depending on configuration.

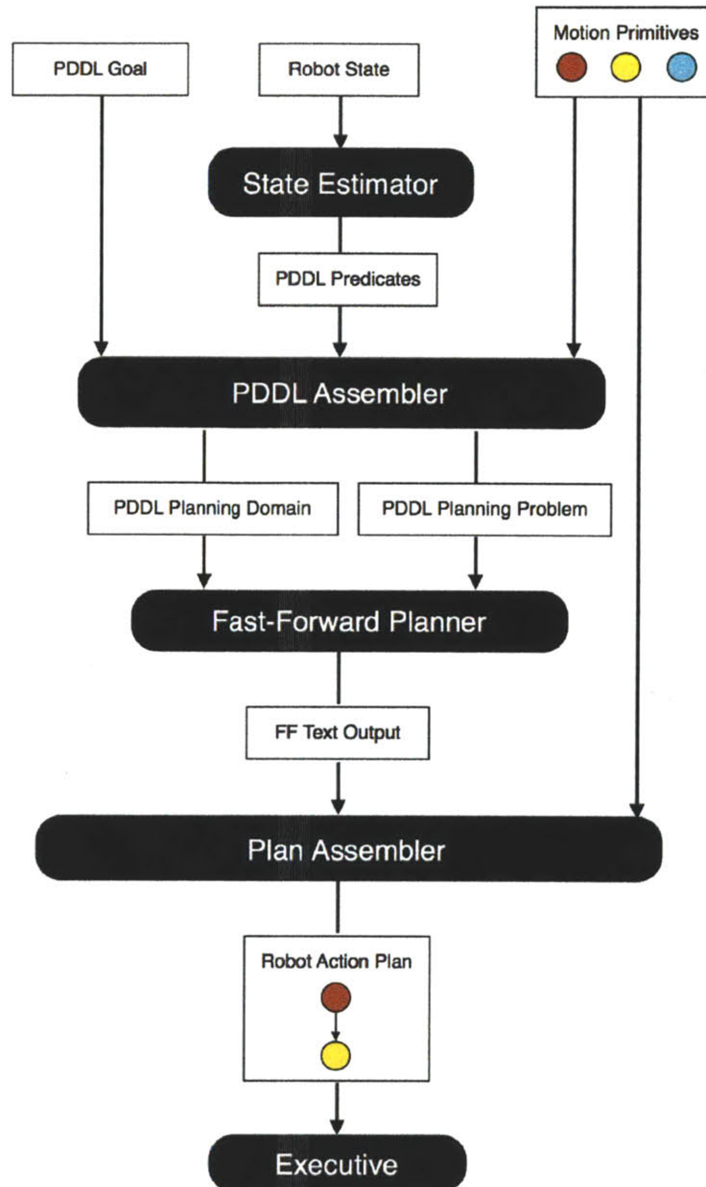


Figure 3-11: The architecture of the `smachforward` system. This system is nested within a single state in a larger static finite state machine, such as is shown in Figure 3-10.

Robot State access to the necessary `SMACH` `userdata` and ROS network connections to satisfy the State Estimator implementation

Motion Primitives a set of `SMACH` states implementing the `fstate` interface. In addition to being an executable `SMACH` state, each state specifies pre- and post-conditions in PDDL and is limited to a boolean result (success or failure).

State Estimator

The State Estimator takes the `SMACH` `userdata` and ROS network as input and uses it to determine the initial conditions of all of the PDDL predicates used by the motion primitive set. This is an abstract class that is implemented by each `smachforward` dynamic state machine. The State Estimator generates a list of PDDL predicates and their boolean values.

PDDL Assembler

The PDDL Assembler takes the PDDL Goal, the initial condition PDDL predicates from the State Estimator, and the set of motion primitives and generates two files:

PDDL Planning Domain a definition of each predicate and a description of each motion primitive, with pre- and post- conditions described in terms of the defined predicates.

PDDL Planning Problem an initial condition boolean value for relevant predicates and a goal state described in terms of the defined predicates in the PDDL Planning Domain.

Fast-Forward Planner

`smachforward` provides a Python wrapper around a command-line call to `ff`. It passes `ff` the PDDL Planning Domain and PDDL Planning Problem and passes the output to the Plan Assembler.

Plan Assembler

The Plan Assembler parses the Fast-Forward planner output into a Robot Action Plan. If Fast-Forward could not find a plan to reach the goal the Plan Assembler generates an

empty plan, triggering the `smachforward` state to return failure. If a plan is found, the Plan Assembler uses the sequence of states in the plan to assemble a corresponding sequence of motion primitives. It links the motion primitives together into a linear sequential SMACH finite state machine and passes it to the Executive for execution.

Executive

The Executive runs the Robot Action Plan. If the plan results in success, the Executive terminates and the `smachforward` state returns success. If plan execution fails, the Executive handles replanning, analogous to Figure 3-11. A user-specified number of replanning attempts are allowed. When this number is exceeded the `smachforward` state returns failure.

To replan, the Executive passes control back to the State Estimator, which estimates the new system state and the process is repeated.

3.5.4 Discussion

`smachforward` enables us to create dynamic state machines nested within static state machines. This hybrid capability allowed us to create a basic robot control framework that uses planning to respond to failures in task execution. Encapsulating the planning in individual `smachforward` states within a larger state machine eased made it easier to transition from a fully static state machine to the hybrid system: not everything needed to be ported at once. The `smachforward` state machine uses motion primitives that are extensions of regular SMACH states, meaning that only the addition of PDDL markup to existing states is necessary to port the states to the dynamic planning system.

Specifying a dynamic finite state machine required less work than handling all of the failure cases of a large static finite state machine. The brunt of the implementation effort is placed in the State Estimator, which must be implemented for each application. Careful specification of pre- and post- conditions is important, having an over-specific predicate set unnecessarily complicates task execution.

When implementing the BakeBot system using `smachforward` it was useful to prototype the entire system in PDDL before deciding on the appropriate markup for the existing

SMACH states. A generator that converted the PDDL into valid SMACH state definitions in Python was desired and will be implemented in future relases. Ultimately the greatest difficulty in porting the BakeBot system to smachforward was in redesigning state behaviors when they were revisited and handling the userdata to ensure that states had the information they needed regardless of the order they were executed in. These issues arose from all of the logic embedded in the states that was necessary for the enormous static state machine to function, a fresh writing of the states would have preempted many of these problems. Section 4.4 discusses implementing BakeBot as a dynamic finite state machine.

3.6 Logging

ROS offers the ability to log all messages between nodes running on the robot, enabling one to reconstruct the robot state of testing runs for data analysis and simulation. While this capability was useful when testing and debugging, we needed a logging system that was suited to the hierarchical nature of our BakeBot implementations that could provide information about task execution rather than about robot state. To address this need, we designed a custom logger for BakeBot, the `htlogger` (for Hierarchical Task Logger). The `htlogger` logs the starting and ending of hierarchical tasks, producing text logs that record task success/failure status and duration.

3.6.1 Framework

`htlogger` runs as an individual ROS node on the PR2 network. It receives messages over the ROS network, using these to generate its hierarchical log. A client class was implemented to syntactically streamline message construction and sending, though properly formatted messages can be received from any node or class on the robot.

3.6.2 Activity Representation

Logging was performed using the idea of an *activity stack*. Activities are created by the logger whenever it receives a "start logging" command. New activities are added to the activity stack and are represented as the child activities of the other activities in the stack. When a "stop logging" command is received activities are popped off of the stack until the activity corresponding to the command is reached.

Figure 3-12 shows typical logging output. The hierarchical nature of the robot task execution is apparent in the output. Each new logging start command is represented by a new line indented from the previous start command. Each logging stop command terminates all previous commands between it and its associated start command. This behavior is demonstrated in the figure by the final stop command, most likely caused by abrupt termination of the robot control program. The "STOP — logging (117)" command terminates all activities in the stack up to the matching "START — logging (117)" command is reached (117

is the unique activity number).

3.6.3 Discussion

The log output, in simple structured text, provides an intuitive representation of the hierarchical robot task. Useful statistics, such as task and subtask durations and success rates can be calculated by processing the logs. `htlogger` was very useful during BakeBot development and is being packaged for release to the ROS community.

```
START | logging (117) @ 1311188726 | | 2011-07-20-bakebotlogging.log
      START | fsm (118) @ 1311188735 | | FSM stateclass RobotInitialzeState (bakebot fsm)
      STOP | fsm (118) @ 1311188750 (duration 15) | success
      START | fsm (119) @ 1311188750 | | FSM stateclass UserInitializeState (bakebot fsm)
      STOP | fsm (119) @ 1311188757 (duration 7) | success |
      START | fsm (120) @ 1311188764 | | FSM stateclass DealMaster firstpass (bakebot fsm)
            START | fsm (121) @ 1311188764 | | FSM stateclass dealing 1 ingredient
                  START | milestone (122) @ 1311188766 | | grab and postgrab lift and rotate
                        START | fsm (123) @ 1311188766 | | FSM stateclass Initialize state (deal bowl fsm)
                        STOP | fsm (123) @ 1311188766 (duration 0) | success |
                        START | fsm (124) @ 1311188794 | | FSM stateclass RefineMixingBowlPosition (mixing fsm)
                              START | refine and get graspable (125) @ 1311188794 | |
                              STOP | refine and get graspable (125) @ 1311188806 (duration 12) | success |
                        STOP | fsm (124) @ 1311188806 (duration 12) | success |
                        START | fsm (126) @ 1311188808 | | FSM stateclass grab (deal bowl fsm)
                              START | refine and get graspable (127) @ 1311188808 | |
                              STOP | refine and get graspable (127) @ 1311188818 (duration 10) | success |
                              START | grasp (128) @ 1311188819 | |
                              STOP | grasp (128) @ 1311188856 (duration 37) | success |
                        STOP | fsm (126) @ 1311188860 (duration 52) | success |
                        START | fsm (129) @ 1311188861 | | FSM stateclass lift (deal bowl fsm)
                              START | move arm (130) @ 1311188861 | | moving to pose
STOP | logging (117) @ 1311188867 (duration 141) | failure |
```

Figure 3-12: Typical output from the `htlogger`. This excerpt shows robot initialization, followed by grabbing the first ingredient and attempting to pour it into the mixing bowl. It can be seen that the testing run ended 141 seconds after it started, during the finite state machine state "lift" while calling a "move arm" action. The hierarchical nature of the task execution is reflected by the indentation in log output.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

System Design Methodology

BakeBot was developed from the bottom up. This enabled us to add functionality incrementally while maximizing code reuse. Incremental development facilitated testing and debugging: only when a component was implemented and thoroughly tested was new functionality added on top of it.

The bottom-up design methodology was:

1. create clients to abstract away hardware and ROS layers,
2. discretize the baking task into subtasks and design sequences of motion primitives to execute the subtasks,
3. design and implement a static finite state machine to robustly execute the sequences of motion primitives,
4. and design and implement a dynamic state machine that plans and executes sequences of motion primitives at runtime.

Our design goal was to develop a functioning end-to-end system with the minimum of development effort; we sought to integrate existing systems, rather than creating new ones from scratch, whenever possible. Some of the solutions presented are not optimal. What is important however, is that they work in the real world on a real robot, and more importantly, that all of the solutions presented *work together in an end-to-end system.*

4.1 Hardware/ROS Abstraction Layer

A set of client classes were designed to abstract away the robot hardware and other ROS services. These classes were useful for several reasons. They abstracted away many of the unnecessary or unchanging fields in ROS messages, eliminating programming overhead when requesting services from the ROS system. An additional benefit of abstracting away the ROS service calls to the client classes was that these were the only classes that had to be changed when updates in ROS changed the message formats (new versions of ROS are pushed out twice annually). The client classes provided a consistent and relatively stable API, facilitating software development.

The client classes were implemented as singleton classes and utilized the "static factory method" design principle. This eliminated expensive reinitialization of connections into the ROS system. This also allowed consistent internal state to be shared across classes that utilized the clients, an important feature when using the SMACH state machine framework, which represents each state as its own class.

4.1.1 Arm Client

The Arm Client abstracts away calls to the inverse kinematic planning and cartesian trajectory following services. It was modeled on the `pick_and_place_manager` class in the ROS `pick_and_place_demos` package and utilizes a modified version of the `controller_manager` class from the ROS `pr2_gripper_reactive_approach` package.

The Arm Client provides simple method handles that moves the end effector to a position in the workspace. Options are provided to interpolate a trajectory between the current pose and the desired pose in cartesian space (minimizing extraneous end effector movement) and to move with orientation constrains on the end effector. It checks the results of the inverse kimeatic planning and execution, if the end effector ends up within a tolerance of the desired position at the end of a specified time interval success is reported.

The Arm Client had several built-in robustness mechanisms: it can be configured to replan upon failure, to recursively plan closer to a desired position, or to selectively scale back the end effector orientation and interpolation constraints in the event of repeated plan-

ning failures.

4.1.2 Base Client

The Base Client translates and rotates the PR2 base. It receives odometry messages from the ROS framework and sends velocity messages to guide the robot base to a desired position. It uses a PID controller to perform closed loop velocity control of the base. The accuracy of the odometry was negatively affected by wheel slippage but the error over the course of cooking was acceptable and did not negatively affect system perforamnce. Countermeasures to the odometry drift include localizing based on constant features in the environment (the oven, the table, etc.), relying on external odometry data (i.e. from a motion capture system), or utilizing SLAM techniques. They can be implemented in the future if necessary.

4.1.3 Mix Client

The Mix Client abstracts away the generation of the force/compliant mixing trajectories. It provides a single entry point for all mixing operations: plunging the spatula into the bowl, circular and linear mixing, rim plunging, and cleaning the spatula. It does not perform and manipulation of the mixing bowl (i.e. grabbing the bowl or switching the sides of the bowl grasp), it only sends the compliant mixing trajectories.

At the time that the mixing code was written, use of the `ee_cart_imped` controller was limited to programs written in C++. The Mix Client acts as the Python intermediary between the main BakeBot software base and a low-level C++ code that generates and executes the trajectories.

4.1.4 Torso Client

The Torso Client acts a single entry point for movement of the PR2 torso degree of freedom (up-down). It provides a minimal API to manipulate the torso position.

4.1.5 TF Client

The TF Client provided a single point to access frame transformations. It eliminated the need to reinitialize TF for every class that used information from the service, the reinitialization was buggy and expensive. The client also provided a simple API that hid unnecessary frames to eliminate programming error when access frame transforms.

4.1.6 Controller Manager

The Controller Manager switched between `ee_cart_imped` controller and the `cartesian_trajectory` controllers for each arm. It was necessary to switch between these controllers when going from compliant control to cartesian endpoint control.

4.2 Task Discretization

The baking task was broken into independent high level subtasks: finding the ingredients on the table, collecting the ingredients into the mixing bowl, mixing everything together, scraping the batter onto the cookie sheet, and putting the cookie sheet into the oven. Figure 4-1 shows this breakdown pictorally. The high level subtasks are naturally independent and roughly correspond to the chronological actions a human would use to bake a simple cookie recipe. Table 4.1 provides an overview of the high level subtasks and their requirements and effects.

Table 4.1: The high level subtasks of the baking task with their respective requirements and effects.

High Level Subtask	**Requires**	**Effect**
finding the ingredients on the table	None	The positions of the ingredients, mixing bowl, and cookie sheet become known.
collect an ingredient	The ingredient and mixing bowl position are known, the mixing bowl is on the table.	The contents of the ingredient bowl are poured into the mixing bowl and the ingredient bowl is disposed of.
mixing the ingredients in the mixing bowl	The mixing bowl is on the table and contains ingredients.	The contents of the mixing bowl are mixed.
scrape batter onto cookie sheet	The mixing bowl contains ingredients.	The contents of the mixing bowl are transferred to the cookie sheet and the mixing bowl is disposed of.
put the cookie sheet into the oven	The cookie sheet contains ingredients.	The cookie sheet is removed from the table and placed into the oven.

High level subtasks like ingredient collection and mixing can be done repeatedly in several different orders. For example, one could collect ingredient A, collect ingredient B, mix, collect ingredient C, then mix. Alternately, one could collect ingredient C, collect ingredient A, collect ingredient B, then mix.

High level subtasks such as scraping the ingredients onto the cookie sheet or baking the cookie sheet in the oven can only happen at specific points in the baking process. For

example, it does not make sense to put an empty cookie sheet into the oven, nor does it make sense to pour the nonexistant contents of the mixing bowl onto the cookie sheet. Scraping the contents out of the mixing bowl onto the cookie sheet results in the disposal of the mixing bowl, thus scraping cannot be done until all of the ingredients have been added.

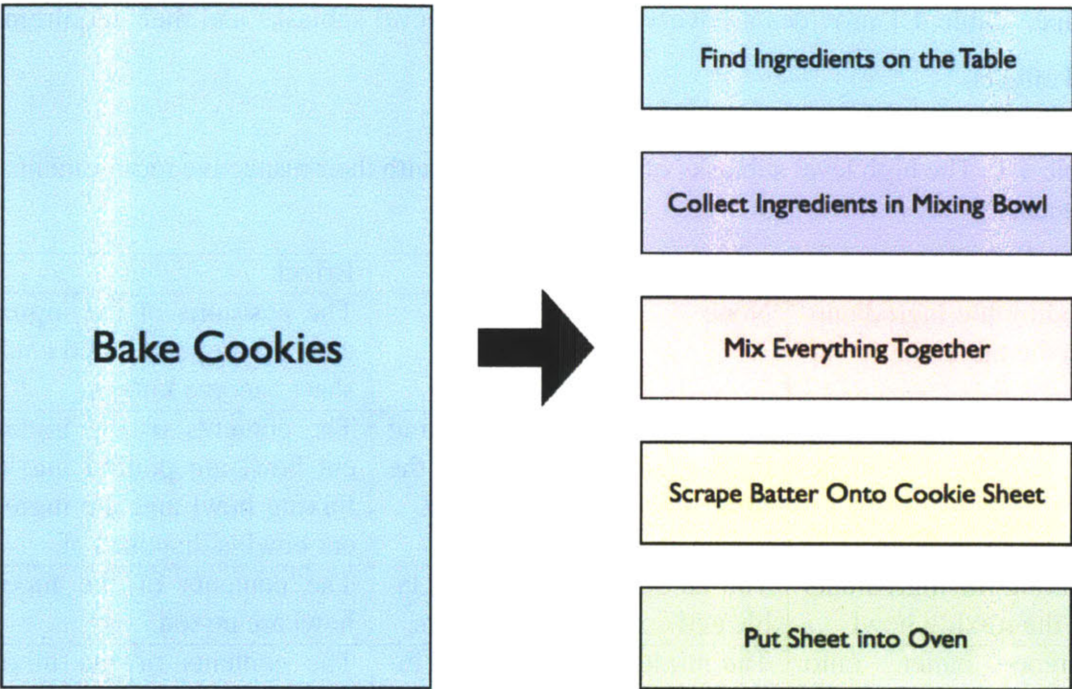


Figure 4-1: The baking task was broken into high level subtasks.

Each high level subtask was further broken down into a sequence of actions. Figure 4-2 shows this breakdown pictorally.

4.2.1 Finding the Ingredients

Locating and identifying ingredients was discussed in Section 3.1. The `Broad Tabletop Object Manager` was used for detection and persistent storage of ingredient information. It contained an internal representation of the detected objects that enabled grasp actions to

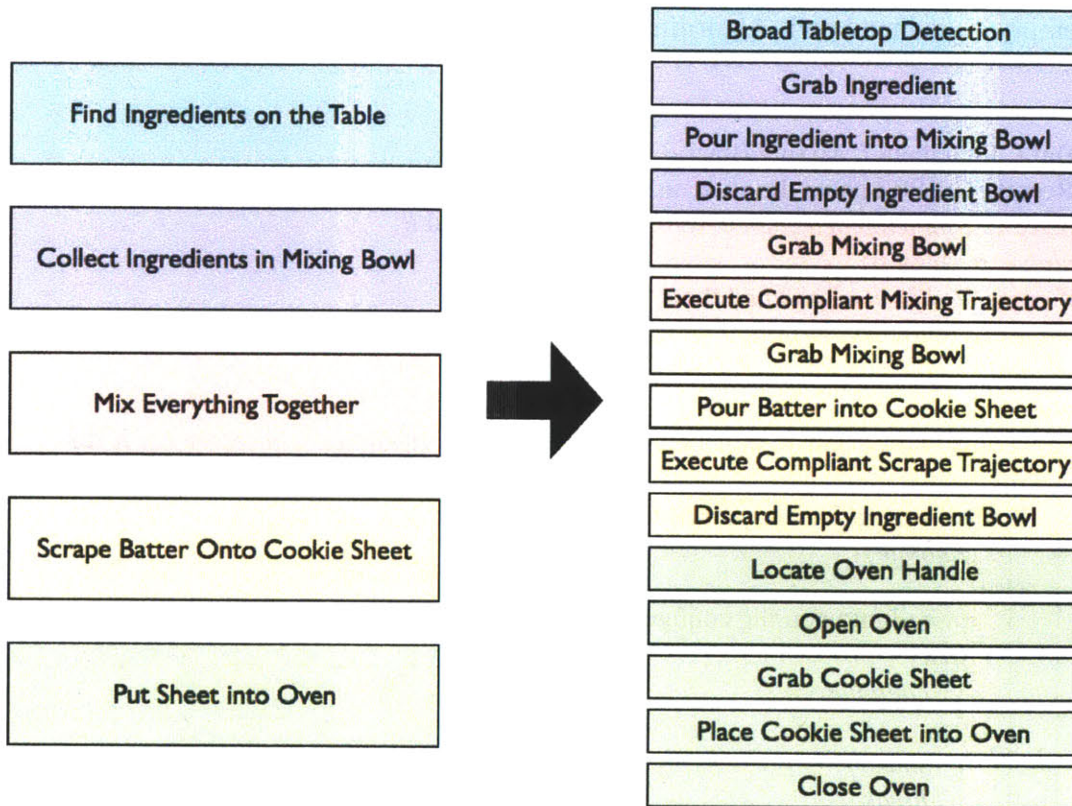


Figure 4-2: Each high level subtask was broken into lower level action sequences.

be performed on those objects.

4.2.2 Collecting an Ingredient

Adding an ingredient to the mixing bowl was broken down into three high-level actions: grabbing the ingredient bowl, pouring the ingredient into the mixing bowl, and discarding the empty ingredient bowl. Algorithm 2 outlines the procedure for collecting an ingredient.

```
Data: Ingredient I, Mixing Bowl B
Result: The PR2 pours the contents of the bowl I into the mixing bowl B
perform a tabletop detection to refine positions of I and B;
plan a grasp on B;
if grasp could not be planned then
    move robot base to bring I within nominal grasp region;
    plan a grasp on B;
end
plan collision-free path to move I above B for known orientation k in known list K do
    rotate to k;
    if rotation could not be planned then
        continue;
    else
        rotate I into pouring configuration for orientation k if rotation could not be planned
        then
            continue;
        else
            shake I;
            break;
        end
    end
end
move I to place position P;
if move could not be planned then
    translate robot base closer to P;
    move I to place position P;
end
place I;
translate robot base to starting position;
```

Algorithm 2: ADD INGREDIENT

Grabbing Ingredient Bowl

Grabbing the ingredient bowl was performed using a modified version of the `pick_and_place_manager` from the ROS `pick_and_place_demos` package. Given a stored representation of a detected object (from the `Broad Tabletop Object Manager` for example), the detection is refined by rescanning the object with the laser scanner and stereo camera pair, then a grasp is computed by the ROS `pr2_gripper_reactive_approach` package and executed. Refining the detection was important: it eliminated uncertainty caused by moving the robot base or environmental changes. Before executing the grasp, the position of the mixing bowl was also refined, ensuring that the pour movements occurring later would also be accurate. Successful grasps result in the bowl being firmly held by the gripper about 10 cm over the table.

Planning movement of ingredients near the surface of the table was difficult and error-prone. The uncertainties in detection (leaving bounding boxes slightly larger than the actual objects) and the configuration space of the robot made manipulating objects near the table surface in a reliably collision-free way difficult. As the focus of this thesis is on the development of an end-to-end system rather than on manipulation planning, we sought to simplify the manipulation task to make it possible to execute with the standard OMPL planning system in ROS.

To simplify manipulation of the ingredient bowls they were first lifted over all of the other bowls on the table. The height they were lifted to was determined experimentally and was the height of the detected table surface plus 50 cm. Any object lifted to this height above the table would not collide with any other objects on the table when translated laterally across the workspace.

Pouring the Ingredient

Pouring the ingredient into the mixing bowl requires several movements, each with a specific set of orientation constraints. Planning these movements in the cluttered table environment was difficult and often unsuccessful. It was useful to break down the pour into several stages: an initial movement over the mixing bowl, a rotation of the bowl to a known

orientation, and a pour maneuver from that orientation. This breakdown was useful for several reasons: it allowed the movement to be planned in stages, which was both more intuitive and more likely to result in planning success; and it offered several points from which to replan, if a specific orientation of the bowl did not result in any successful pouring plans the bowl could be rotated to a new orientation and planning could be reattempted.

Determining Bowl Pose in Hand Before movements to the pre-pour position could be planned, it was necessary to determine the grasp pose, primarily which "finger" was inside of the bowl. This was determined by computing the position of the grabbed object's (bowl's) frame in relative to the manipulator's frame, something that could be computed by TF. One finger on the gripper has a prominent barcode, this was used as a intuitive visual reference point, all grasps on the bowl were classified as *barcode_in* or *barcode_out*.

Moving the Bowl Over Mixing Bowl Once lifted over the other objects on the table, the bowl is moved laterally to over the center of the mixing bowl. The movement is done while maintaining constant orientation of the end effector to ensure that the contents of the bowl are not spilled.

Discretization of Bowl Poses To make the pours easy to specify and intuitive to program, the range of grasps around the circumference of the bowl were discretized into four grasp positions, corresponding to the cardinal directions on the bowl circumference. These directions were referred to as (and corresponding to the robot's view of the bowl): *three o'clock*, *six o'clock*, *nine o'clock*, and *twelve o'clock*. Since the gripper can hold the bowl either *barcode_in* or *barcode_out*, a total of eight bowl grasps exist. Figure 4-3 shows the grasp poses on the mixing bowl.

Pour Poses The bowl is rotated to one of the cardinal bowl pose discretizations (based on whether the barcode is on the inside or the outside of the bowl) before pouring. A series of pour poses and translational offsets are mapped to each bowl pose. The pour maneuver moves through these pour poses and offsets, each one steeper than the last. The primary reason to break the pour into stages was to accommodate for planner failure, in the event

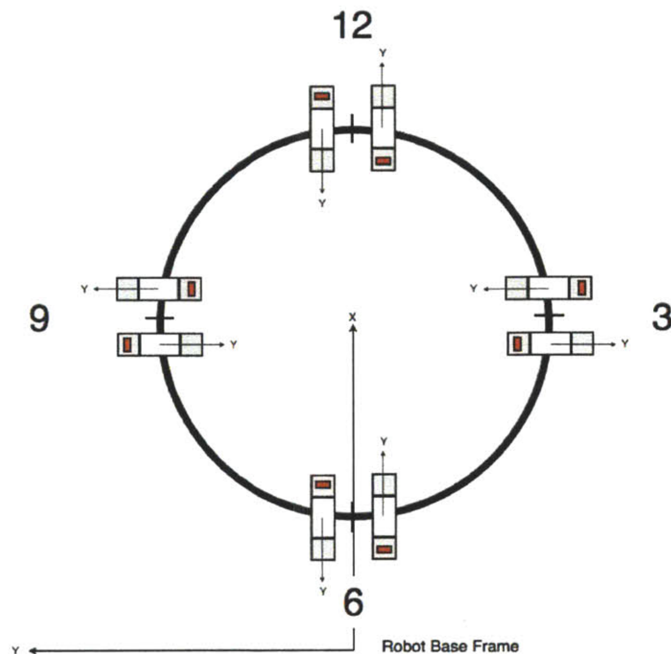


Figure 4-3: The eight left end-effector position discretizations around the mixing bowl circumference. The red square indicates the barcode on the end effector (opposite the positive Y-axis in the end effector's frame).

that final steep pour could not be planned to at least one of the intermediate pour waypoints would allow for some of the ingredient to be dumped into the mixing bowl. Table 4.2 lists the grasp and pour orientations for each of the bowl grasp discretizations. The steps of the pour maneuver are shown in Figure 4-4

Discarding Empty Bowl

Once the ingredients were poured into the mixing bowl the empty bowl is dropped into a bin beside the table. Initial efforts were made to place the bowl back onto the table, but the version of the ROS manipulation pipeline (Diamondback release) in use with BakeBot experienced difficulties placing point-cloud based (rather than database-matched) detected objects onto a table surface. These difficulties and a desire to keep the table area clean and free of unnecessary objects led us to choose to dump the empty bowls into a bin.

The empty bowl is dumped into the bin in stages. First the bowl is rotated back to its original pre-pour orientation (to prevent spilling of any accidentally-unpoured ingredients).

Table 4.2: For each cardinal discretization around the bowl circumference (shown in Figure 4-3), the left end effector: grasp orientation; pour offset direction from mixing bowl center; and orientations for the pour, steep pour, and super steep pours.

Pose	**Grasp** (q_x, q_y, q_z, q_w)				**Pour Offset** (x, y, z)			**Pour** (q_x, q_y, q_z, q_w)				**Steep Pour** (q_x, q_y, q_z, q_w)				**Super Steep Pour** (q_x, q_y, q_z, q_w)			
6I	-0.5	0.5	0.5	0.5	0	0	0	0.2	0.7	-0.7	0.2	0	-0.7	0.7	0.3	-0.2	0.7	-0.7	0.2
6O	-0.5	-0.5	0.5	-0.5	1	0	0	-0.1	0.7	0.7	0.2	0.3	0.6	0.7	-0.1	0.2	0.7	0.7	-0.3
9I	0	0.7	0	0.7	1	0	0	-0.7	-0.2	-0.1	0.7	-0.2	-0.4	0.3	0.9	-0.6	-0.3	-0.3	0.7
9O	0.7	0	-0.7	0	-1	0	0	0.7	0.1	-0.1	0.7	0.8	0.3	0.2	0.3	0.6	-0.4	0.4	0.6
12I	-0.5	-0.5	0.5	-0.5	-1	0	0	-0.4	0.6	-0.4	0.6	-0.7	0.3	-0.3	0.6	0.6	-0.4	0.6	-0.4
12O	-0.5	0.5	0.5	0.5	0	-1	0	0.6	-0.5	-0.5	0.4	0.5	-0.6	-0.4	0.6	0.3	-0.6	-0.3	0.7
3I	0.7	0	-0.7	0	0	-1	0	-0.6	-0.4	0.6	0.4	-0.4	-0.6	0.3	0.7	-0.3	-0.7	0.2	0.7
3O	0	0.7	0	0.7	0	1	0	0.4	0.6	0.4	0.6	0.5	0.5	0.5	0.5	0.6	0.3	0.7	0.4

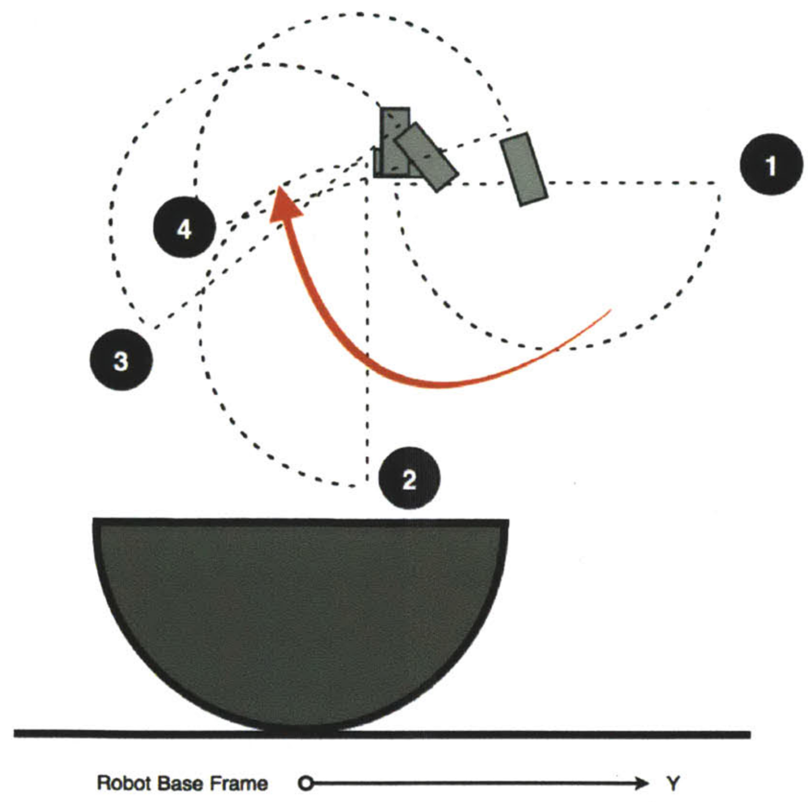


Figure 4-4: The four steps of the pour maneuver, corresponding to the pour orientations in Table 4.2: (1) the pre-pour position offset from the center of the mixing bowl (Pour Offset), (2) the Pour pose, (3) the Steep Pour pose, and (4) the Super Steep Pour pose.

Data: Mixing bowl B
Result: The PR2 mixes the contents of the B together.
Perform a *tabletop detection* to refine position of B;
move base to bring B into nominal mixing region;
grasp left side of B with the left end effector;
move spatula over center of mixing bowl in collision-free way;
move spatula down into mixing bowl;
execute compliant mixing trajectories;
move spatula to original pose in collision-free way ;
release mixing bowl;
return base and end effectors to their original poses;
Perform a *tabletop detection* to refine position of B;

Algorithm 3: MIX

Next the bowl is translated while maintaining constant orientation to a position over the bin. If the bin is outside of the workspace the robot is translated closer to the bin and planning of the bowl movement is reattempted. Once over the bin, the bowl is lowered in stages before the manipulator is opened, releasing the bowl.

4.2.3 Mixing the Mixing Bowl

Mixing is the most manipulation intensive component of the BakeBot system. It combines joint position control, planning inverse kinematic paths for Cartesian trajectories, and force-compliant control of the end-effector. Algorithm 3 describes the mixing process.

Balancing Workspaces

The PR2 constrains the mixing bowl by executing a grasp on the left of the mixing bowl and maintaining it during the mixing motion. The position of the mixing bowl during mixing was determined experimentally using the technique described in Section 3.3.2. This position maximized the effectiveness of the compliant mixing motion and also placed the mixing bowl in a position where it could be grabbed and held by the non-mixing manipulator. Unfortunately this mixing bowl position is not conducive to pouring ingredients into the mixing bowl. This required the robot to be moved (relative to the mixing bowl) before mixing and returned after mixing has been completed. The robot first redetects the mixing bowl position, then translates to a calculated offset to put the mixing bowl in the desired

position, then redetects the mixing bowl position to ensure that it is as desired. After mixing, the opposite procedure is performed. The mixing bowl itself could not be moved as this would interfere with the other bowls on the work surface.

Mixing Bowl Grasp

The grasp pose was chosen to hold the mixing bowl in place while keeping the grasping arm out of the way of the mixing arm. In order to have a consistent grasp and mixing bowl location for every mixing attempt, the standard stochastic grasp planner could not be used. Likewise, the proximity of the mixing bowl to the robot in the cluttered workspace and the salient grasp features within the mixing bowl made grasp planning unreliable. Instead, a specialized grasping routine was used, following this procedure:

1. Calculate the position, radius, and height of the mixing bowl based on a fresh object detection.
2. Determine the *nine o'clock* position of the mixing bowl circumference
3. Plan and execute a collision free path placing the end effector several inches over this position with the axis of gripper opening perpendicular to the tangent of the circumference (either *barcode_in* or *barcode_out*).
4. Open the end effector and lower it so the finger pads are just below the height of the mixing bowl.
5. Close the end effector.

Spatula Movement

Once the mixing bowl is held in place the spatula is moved into position inside of the mixing bowl. The inverse kinematic planner could not plan collision free movements of the spatula-end-effector from its initial position beside the robot to inside the mixing bowl. The movement was instead planned in several stages: first a pre-recorded joint trajectory was followed to bring the spatula to a safe position slightly over the table and to the side

of the robot; next a Cartesian trajectory was planned based on the detected position of the mixing bowl, bringing the spatula vertical immediately over the center of the mixing bowl; next a force trajectory was calculated to plunge the spatula downwards through the batter and into contact with the bottom of the mixing bowl.

Mixing

Mixing was performed by executing a series of mixing trajectories, which are outlined in Section 3.3.1.

Undoing mixing setup

The spatula movement and bowl grasp steps were reversed to bring the robot back into its pre-mix configuration: the spatula was lifted out of the bowl, then returned to its safe above-and-beside the table position, then retreated using a joint trajectory; the mixing bowl was released by opening the gripper, lifting it, and returning it to its beside-the-robot starting position, and the robot base was servoed back to its starting position.

4.2.4 Scraping Onto the Cookie Sheet

Pouring the final mixture from the mixing bowl into the cookie sheet utilized two action sequences: the pouring action, parametrized for the mixing bowl and the cookie sheet (rather than for ingredient bowls and the mixing bowl, respectively), and the mixing action, parametrized for a horizontal scraping motion on the upturned mixing bowl. Balancing the workspaces of the compliant controller on the right arm (with the mixing spatula) and the Cartesian controller on the left arm (holding the upturned bowl over the cookie sheet) proved difficult and required considerable trial and error to find a solution that worked consistently.

Grabbing the Mixing Bowl

The mixing bowl was grasped using the same grasp planner used to grab the ingredient bowls, discussed in Section 4.2.2. Likewise, the mixing bowl was also lifted above the

other objects on the table (in this case only the cookie sheet). The orientation of the gripper relative to the bowl was determined using the same technique previously discussed. The mixing bowl pour pose was limited by the requirement that the poured bowl be facing the right of the robot so the contents could be scraped out by the spatula. Therefore the bowl was rotated so that it was grasped at the *nine o'clock* position.

Pouring Batter Into Cookie Sheet

The mixing bowl was moved to the side after being grasped so that a clear detection could be made of the cookie sheet, which is normally partially obstructed by the mixing bowl. Once the detection is made, the robot servos to bring the cookie sheet into a nominal position for the scraping maneuver, similar to the base servoing performed for mixing. This nominal position was determined experimentally and was found to balance the workspace requirements of the Cartesian controller used to pour the mixing bowl and the compliant controller used to scrape the contents out of the mixing bowl.

The mixing bowl was then moved over the cookie sheet and moved through the series of pour poses. The pour maneuver was executed identically to that of the ingredients, utilizing the same motion primitives.

Executing Compliant Scrape Trajectory

With the mixing bowl held inverted over the cookie sheet, the spatula is brought from its resting position into a horizontal position inside of the mixing bowl. This follows an identical routine as was used during the mixing action, the only difference being the final orientation and direction of the spatula. The amount of time that it takes to perform the pour motion and bring the spoon into position helps the contents of the bowl to slowly slide down from the upper inner walls of the bowl. Oftentimes many of the mixing bowl contents fall out of the bowl into the cookie sheet by the time the actual scraping is performed, leaving only the batter that is stuck to the inside surface of the bowl. The scraping motion follows the trajectory described in Section .

Discarding the Empty Mixing Bowl

Discarding the empty mixing bowl utilized the same routine and motion primitives as during ingredient collection discussed in Section 4.2.2.

4.2.5 Putting Cookie Sheet Into Oven

Putting the cookie sheet into the oven was done in several stages because the modified PR2, which had the spatula bolted to one gripper, could not open the oven with the cookie sheet "in hand". BakeBot first drives to the oven, locates the handle and pulls it to open the oven, then returns to the table to grab the cookie sheet, then drives to the oven and places the cookie sheet into the oven, and finally closes the oven.

Locating the oven handle and opening the oven was work done with Jenny Barry and Annie Holladay and is further described by [8].

Locating the Oven Handle

The oven handle is located by detecting the front plane of the oven and segmenting all cloud points in front of the plane into grasp features. The table with the oven has a tablecloth on it, extending to the floor. We use this flat surface of the table cloth, in plane with the oven door, to perform our segmentation. Figure 4-5 shows the segmentation plane. The grasp features are filtered by height, size and orientation, to eliminate the oven buttons and spurious laser returns from the oven window. If no feature remains after filtering for the oven handle a new detection is performed.

Opening the Oven

Given the oven handle grasp feature, an inverse kinematic path is planned to place the left end effector just in front of the oven with the manipulator open. The manipulator is then advanced toward the oven until the fingertip sensors detect that the handle has been reached. At this point the manipulator is closed and the next phase of the opening process can occur.

The oven is opened by following a force compliant trajectory in the XZ plan in the robot's base frame. The trajectory specifies an impedance along the -X axis, from the initial position of the grasp backwards to a point approximately where a horizontal open door would reach. A downward force is specified along the -Z axis. The result of this trajectory is the oven door being pulled down and backwards (but only backwards until fully open). Figure 4-5 illustrates this maneuver.

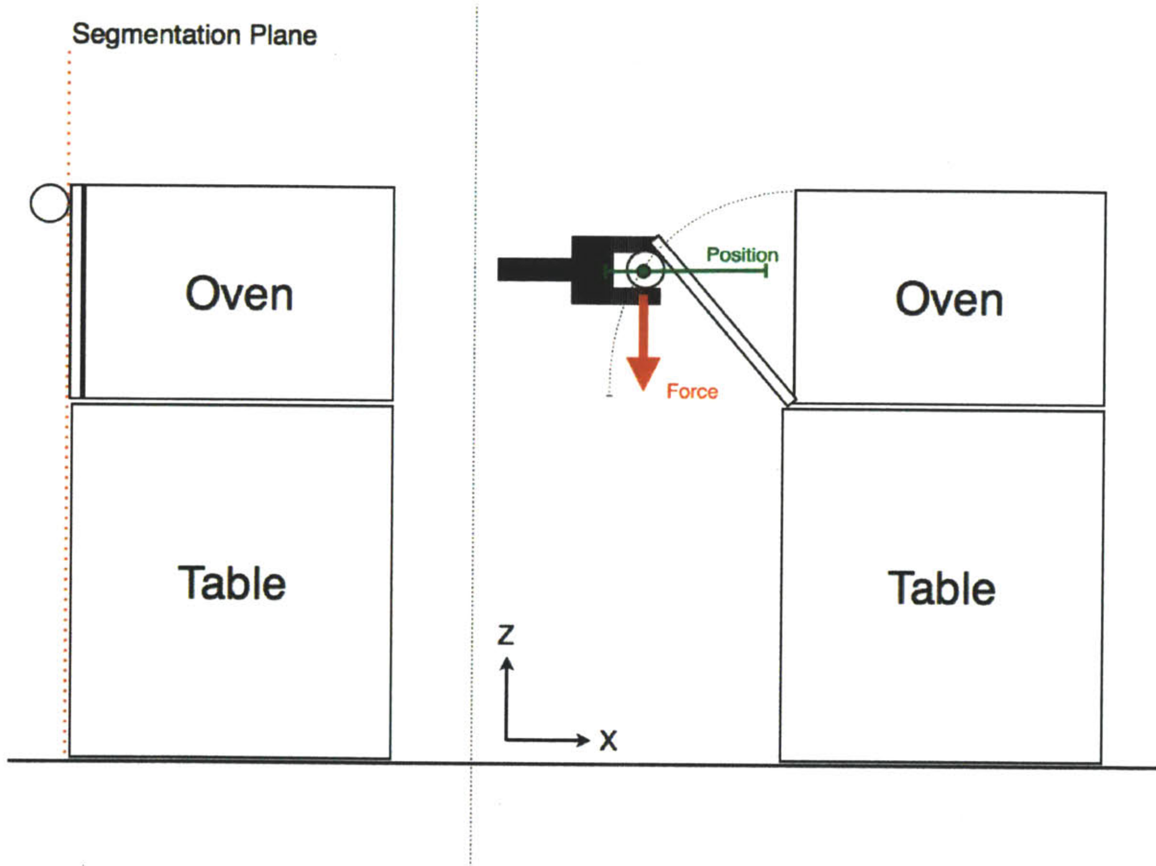


Figure 4-5: The segmentation plane (shown on the left) is coincident with the front of the oven and the hanging tablecloth in front of the table. To open the oven (shown on the right), a force-compliant trajectory is executed with stiffnesses in the X direction and a downward force in the Z direction.

Grabbing the Cookie Sheet

BakeBot drives back to the preparation table after opening the oven. It performs a broad tabletop detection to locate the cookie sheet, which is the only remaining object on the table. A broad tabletop detection is performed in order to guarantee that the cookie sheet is

found regardless of odometry error during translation from the oven and possible variation in the cookie sheet position resulting from the scraping maneuver.

The low height of the cookie sheet and the glob of batter inside of it made it difficult to plan grasps for with the standard grasp planning pipeline. Grasps could either not be found or would attempt to grab salient grasp features of the cookie sheet rather than the rim of the sheet. Successful grasps on the rim would frequently be aborted because the thickness of the rim's lip would confuse the grasp checking algorithm.

Instead of utilizing the grasping pipeline, the same grasping technique that was used to constrain the mixing bowl during mixing, outlined in Section 4.2.3, is used to grasp the cookie sheet. In order to be ready to move the cookie sheet forward into the oven, a grasp is only planned to the *six o'clock* position of the cookie sheet. This guaranteed a successful and consistent grasp based only on the detected position of the cookie sheet.

Before lifting the cookie sheet from the table surface, the orientation of the end effector relative to the cookie sheet (while it is still on the table) is recorded. This is used as the reference orientation for a horizontal cookie sheet, ensuring robustness to variations in the angel of the cookie sheet caused by unmodeled interactions between the lip of the pan and the gripper pads.

Placing the Cookie Sheet Into Oven

BakeBot drives back to the oven and moves itself to an estimated position that places the cookie-sheet (in-hand) directly in front of the open oven. This action is currently performed open-loop with acceptable rates of success. Methods to close the loop include localizing off of a fiducial (either added to the environment or calculated from the oven or table surface), external odometry information from a motion capture system, or the utilization of SLAM techniques.

The cookie sheet is inserted into the oven by planning and executing a path through a series of precomputed Cartesian waypoints while maintaining the horizontal pose of the cookie sheet. Once the cookie sheet is in the oven the gripper is opened and the arm is returned to its neutral position beside the robot.

Closing the Oven

The oven is closed by following a pre-computed joint trajectory that first moves the left manipulator under the oven door, then upwards through the door, causing the door to swing upwards and close. The oven door's internal spring pulls the door closed as long as the robot is able to push it upwards at least half-way.

Leaving the Cookie Sheet Within Oven

The PR2 manipulator is not designed to handle hot objects coming out of an oven. To prevent damage to the platform, we do not remove the hot cookie sheet from the oven. The baking task terminates with the cookie sheet resting within the opened oven.

4.3 Static Finite State Machine

BakeBot was initially implemented as a static hierarchical finite state machine. The high-level state machine coordinates recipe execution, while each of the nested state machines execute *baking instructions* following the task discretization steps outlined in Section 4.2. Implementing the action sequences to accomplish the baking subtasks enabled us to add error handling and recovery, increasing overall system robustness dramatically. Figure 4-6 shows the high-level state machine, states shaded in grey contain nested state machines. There are 169 individual states at the lowest level of the state machine when all of the nested state machines have been expanded.

BakeBot execution starts in `ROBOT INIT` which zeros the odometry and moves the robot into the *standard configuration*. In the *standard configuration* the torso is extended to maximum height and both arms are at the sides of the robot. Figure 2-1 shows the robot in the standard configuration. All action sequences initialize with the robot in the standard configuration and end with it in the standard configuration. This helps with testing and execution by enabling the action sequences to be performed in nearly any order since they all start and leave the robot in the same configuration.

The `USER INIT` state is where BakeBot interacts with the operator to determine whether to scan for objects or load a saved detection from a file and whether to execute a recipe action sequence or jump directly to a sub-state-machine (such as pouring an ingredient or mixing). This state facilitated debugging by enabling the system to start where it left off, rather than starting at the beginning again. During end-to-end system tests a flag was set in the code that skipped this state so the test could be run without any user interaction.

`SCAN TABLETOP` and `LOAD FROM FILE` perform object detection or load object info from a previous detection, respectively. Both have the same result and pass the detected objects to the `FILTER INGREDIENTS` state which associates the detected objects with the mixing bowl, cookie sheet, and ingredient bowls (as discussed in Section 3.1.3).

The `BRANCH` state acts on the program flow decisions made in during the `USER INIT` and either skips to a specific subtask or moves to the `DEAL MASTER` which coordinates recipe following. Pouring an ingredient into the mixing bowl was referred to as "dealing an

ingredient" since the robot resembled a card dealer handling the bowls on the table. This also helped to separate the specific "pour" primitive from the higher level subtask.

The DEAL MASTER is what performs the actual recipe following. A recipe, represented as a list of *baking instructions* (see Section 3.4.1), is followed by delegating each instruction to its corresponding state machine (shown in grey in Figure 4-6). If the respective state machine succeeds, control is returned to the deal master which then delegates the next instruction in the recipe. The ordering requirements of the *baking instructions* is reflected by the state machine. For example, the nested state machine SCRAPE AND POUR goes right to OVENING rather than returning to the DEAL MASTER. Failure of a subtask state machine cannot be recovered from at the high level and results in overall failure of the BakeBot task.

Each sub-state machine, for pouring an ingredient into the mixing bowl, mixing, scraping the ingredient onto the cookie sheet, and putting the cookie sheet into the oven, follows the general activity sequence outlined in Section 4.2 but has added logic and error-recovery states to add robustness.

4.3.1 Collecting an Ingredient

Ingredient collection (aka "dealing a bowl") follows general the action sequence in the task discretization in Section 4.2 with added states and logic to deal with planning failures during bowl movement and pouring. Figure 4-7 shows the finite state machine to collect an ingredient. The state machine:

1. grabs a bowl (state GRAB),
2. rotates the bowl to one of the bowl pose discretizations (state PRE TRANSIT ROTATE),
3. moves the bowl to a pre-pour position relative to the mixing bowl (state TRANSIT BOWL) and dependent on the bowl pose discretization,
4. pours the bowl into the mixing bowl (states POUR, STEEP POUR, SUPER STEEP POUR),
5. attempts to place the bowl back onto the table (states PLACE BOWL, FORCE PLACE BOWL),

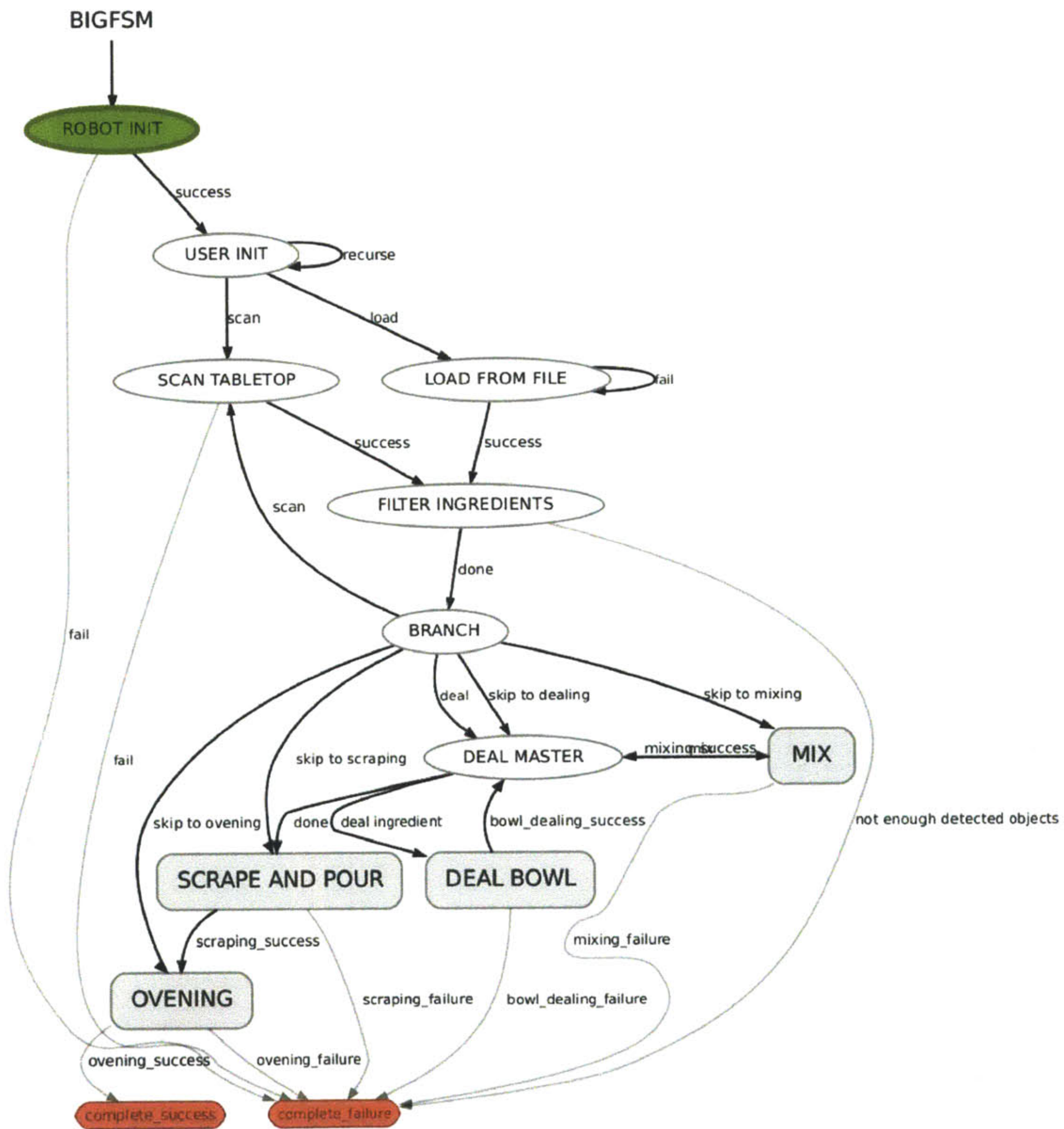


Figure 4-6: The high level BakeBot static state machine. Shaded states represent sub-state machines. DEAL MASTER controls the recipe following and delegation to sub-state machines for subtask execution.

Figure 4-7: The state machine that pours an ingredient into the mixing bowl.

6. should that fail (it always did) dumps the bowl into a bin beside the table (state `DUMP BOWL`),
7. and returns the robot to the *standard configuration*.

Adding Robustness to Grasping

Grasp planning in the cluttered table environment was difficult: bowls were close together, close to the robot torso, and far away from the robot at the edge of the manipulation workspace. Rather than focus on redesigning the grasping pipeline on the PR2, we added robustness to the grasping action via error-recovery states in our finite state machine. If an attempt to grasp an ingredient bowl failed, we:

- moved closer to the bowl if it is far from the robot,
- or moved further from the bowl if it is very close to the robot

and tried again. These actions are encoded in the states `MOVE ROBOT LEFT` and `MOVE ROBOT RIGHT` states, respectively (since the ingredients are on the left side of the table relative to the robot this naming convention was convenient at the time of implementation, the PR2 actually moves left, right, forwards, and backwards). Hard limits were placed on the overall range of motion of the robot base during grasp repositioning in order to prevent the robot from driving forward into the tabletop, laterally into the supporting table legs, or backwards into oblivion.

Adding Robustness to Bowl Movement

Manipulating the ingredient bowl, from the point after the grasp was complete through the pour maneuver, comprised the most difficult inverse kinematic planning tasks of the baking process. The cluttered tabletop, the proximity of the manipulation tasks to the robot's torso, and the range of motion required to perform the tasks complicated the planning.

Bowl movement was broken into stages: lifting the bowl, rotating it to a known orientation, moving it across the table, and finally pouring. If inverse kinematic planning failed, we would attempt to replan to either a different goal or from a different bowl orientation.

When lifting the bowl, if planning failed to find an inverse kinematic path that lifted the bowl vertically from its current orientation, we would replan to points around the desired point. If we still failed to find an inverse kinematic path, we relaxed the orientation constraints on the lifting movement and attempted to plan again. If no trajectories could be found to lift the bowl after relaxing the orientation constraints, the state returns failure but still transitions to the `PRE TRANSIT ROTATE` state.

If planning the inverse kinematics to move the bowl across the table to the pre-pour pose fails, the bowl is rotated to a new grasp pose discretization and planning is reattempted. Preference is given to certain grasp pose discretizations as they increase the probability of success when planning the pour maneuver (they ultimately require less complicated arm movement to accomplish the pour). The ranking of grasp pose discretizations by decreasing preference is: `3 o'clock, 6 o'clock, 12 o'clock` and `9 o'clock`. If planning the rotation fails the next pose is attempted. If rotation to, or planning from, all poses fails twice (two iterations through the pose discretizations), the state machine returns failure.

Breaking the bowl manipulation into stages and enabling replanning between stages enabled us to reliably accomplish a complicated manipulation task in a cluttered environment with a generic inverse kinematic planner.

4.3.2 Mixing the Mixing Bowl

Mixing the mixing bowl follows the general action sequence in the task discretization in Section 4.2. Figure 4-8 shows the finite state machine to mix the mixing bowl. The state machine:

1. servos to the mixing bowl (nested FSM `MOVE ROBOT`),
2. grabs the mixing bowl (nested FSM `GRAB MB`),
3. puts the spatula into the mixing bowl (state `PLUNGE SPOON`),
4. mixes the mixing bowl (state `ASSESS MIX` delegates to states `CIRCLE MIX, LINEAR MIX, WHISK MIX` and the `RIM PLUNGE` and `CLEAN SPOON` nested state machines),
5. and returns the robot to the *standard configuration*.

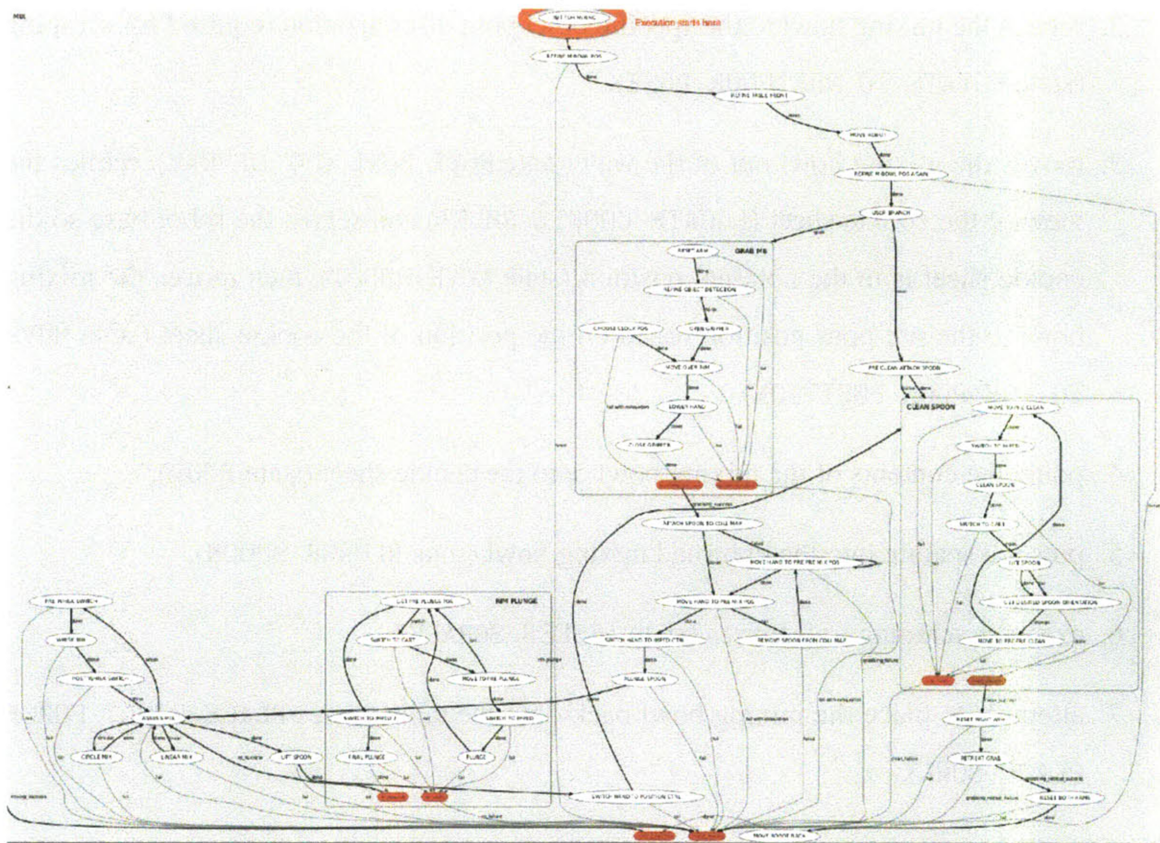


Figure 4-8: The state machine that mixes the mixing bowl.

4.3.3 Scraping Onto the Cookie Sheet

Scraping the batter out of the mixing bowl and onto the cookie sheet follows the general action sequence in the task discretization in Section 4.2. The scraping state machine combines states and features from the ingredient collection and mixing finite state machines. Figure 4-9 shows the finite state machine for scraping. The state machine:

1. grabs and lifts the mixing bowl (state `GRAB MIXING BOWL`),
2. rotates the mixing bowl to the specific bowl pose discretization required for scraping (state `ROTATE TO PRE POUR POSE`),
3. moves the mixing bowl out of the way (state `MOVE BOWL OUT OF WAY`), refines the view of the cookie sheet (`LOCATE COOKIE SHEET`) and servos the robot base so the cookie sheet is in the nominal position (state `MOVE ROBOT`), then moves the mixing bowl to the pre-pour position based on the position of the cookie sheet (state `MOVE TO PRE POUR POSITION`),
4. pours the contents of the mixing bowl onto the cookie sheet (state `POUR`),
5. puts the spatula into the upturned mixing bowl (state `PLUNGE SPOON`),
6. executes scraping trajectories (state `ASSESS SCRAPE`),
7. attempts to place the mixing bowl back onto the table (states `PLACE BOWL, FORCE PLACE BOWL`),
8. and should that fail (it always did) dumps the mixing bowl into a bin beside the table (state `DUMP BOWL`).
9. and returns the robot to the *standard configuration.*

4.3.4 Putting Cookie Sheet Into Oven

Putting the cookie sheet into the oven follows the general action sequence in the task discretization in Section 4.2. Figure 4-10 shows the finite state machine for putting the cookie sheet into the oven. The state machine:

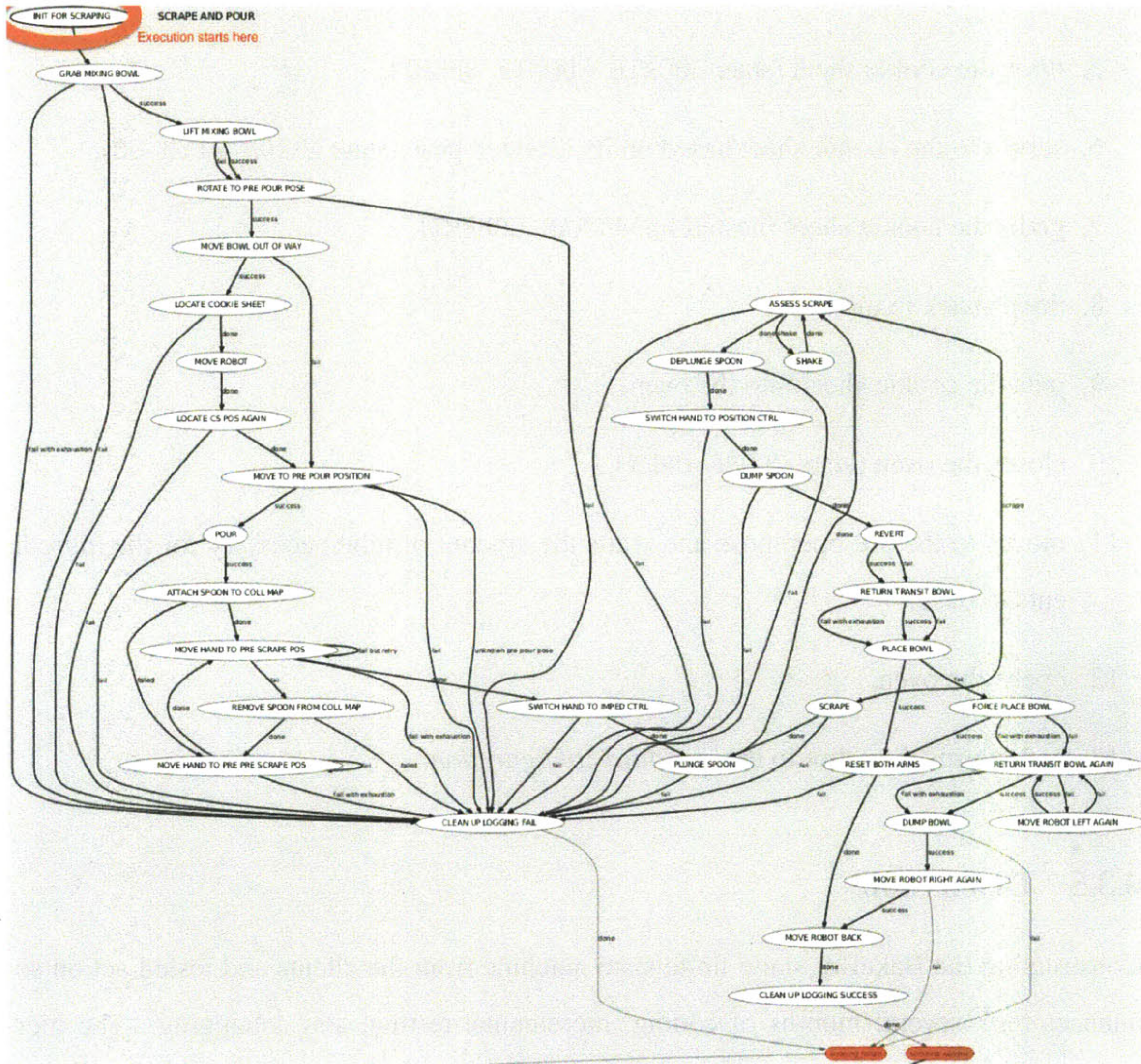


Figure 4-9: The state machine that scrapes the batter out of the mixing bowl and into the cookie sheet.

1. drives the robot to the oven area (state `DRIVE TO OVEN`),
2. servos to the oven based on detected pose of oven handle (state `ALIGN WITH OVEN`),
3. opens the oven (state `OPEN OVEN`),
4. drives the robot back to the table (state `DRIVE TO TABLE`),
5. finds the cookie sheet (state `LOCATE COOKIE SHEET`),
6. servos to the cookie sheet based on its detected pose (state `ALIGN WITH CS`),
7. grabs the cookie sheet (nested FSM `GRAB CSHEET`),
8. drives back to the oven,
9. puts the cookie sheet into the oven,
10. closes the oven (state `CLOSE OVEN`),
11. moves to the pre-open pose and waits the amount of time necessary for the ingredients to bake,
12. opens the oven,
13. and returns the robot to the *standard configuration.*

4.3.5 Discussion

Constructing the BakeBot static finite state machine from the clients and tested action sequences took several months of coding, incremental testing, and debugging. The hierarchical nature of the state machine made understanding and debugging its components manageable, as one did not ever have to deal with the entire state machine simultaneously. Testing the end-to-end baking process took several months: ultimately the presented static finite state machine was tested end-to-end 27 times, with 16 of these tests successfully resulting in cookies.

We learned important lessons about the design and implementation of large state machines for robot control, enumerated below.

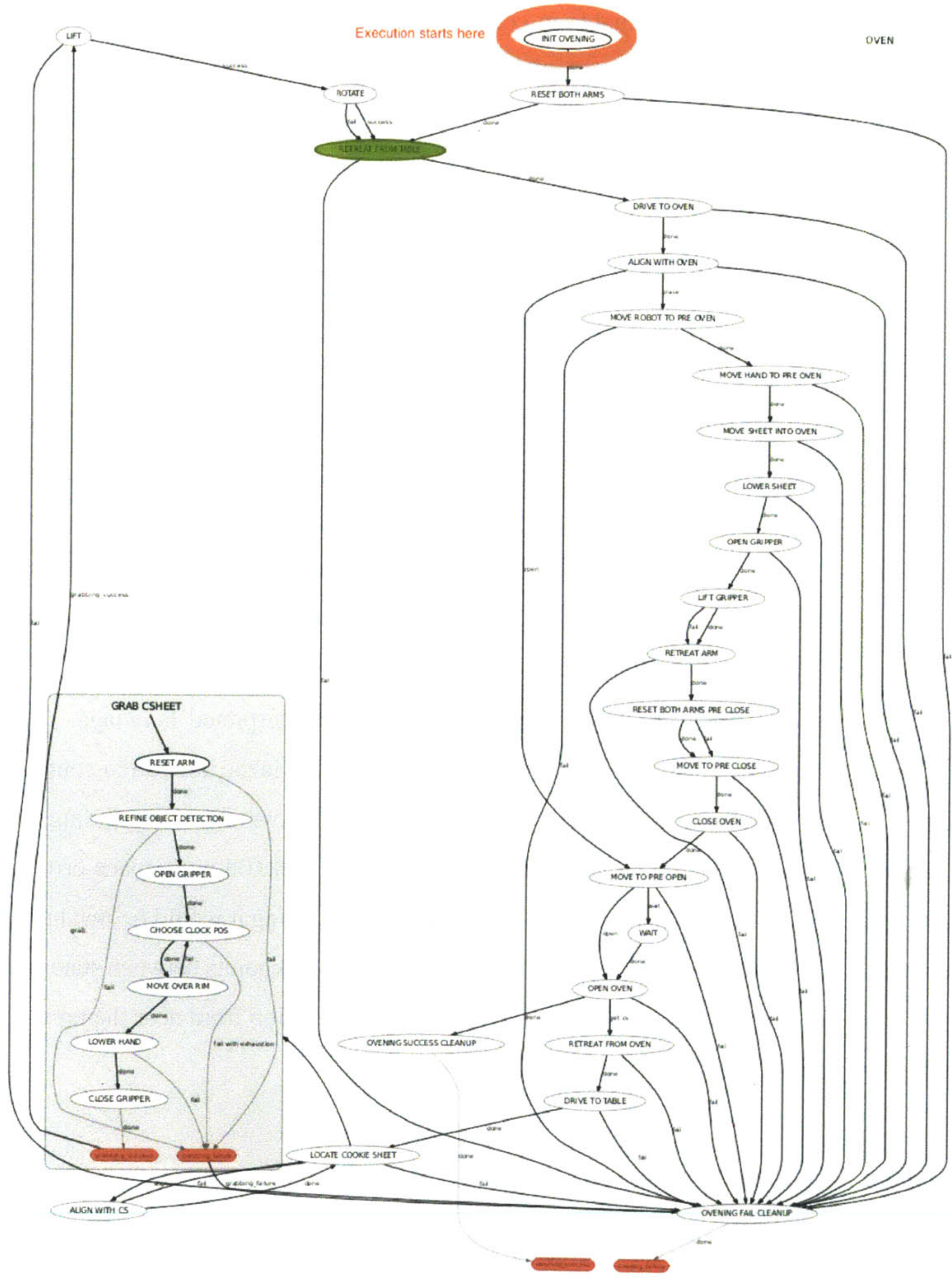


Figure 4-10: The state machine that puts the cookie sheet into the oven.

Pros and Cons of SMACH

The SMACH framework was invaluable in constructing the state machine. It enabled us to focus on implementing the states, rather than worrying about how everything fit together. It encapsulated each state, making it easy for us to shift the finite state machine around and to quickly prototype new frameworks for task execution and error recover. The state machine visualizer helped to determine and debug state transitions. We did, however, experience several growing pains with SMACH as we added layers and complexity to our overall state machine.

SMACH represents each state in the state machine as a unique instantiated object. This means that during the runtime construction of the state machine each state initializes its own subcomponents, creating many unnecessary connections to the ROS network. We addressed this by implementing all of our low-level clients as singleton classes: as a result, each had to only be instantiated once, with a pointer to the runtime object shared across all state classes that used it. This sped up state machine instantiation and facilitated persistence and static variable sharing between states.

SMACH requires states to be implemented in Python, an interpreted language. This, combined with the fact that all inter-state data sharing happened via the `userdata` container object, led to frequent instances where the system would run most of the way through the baking task (often taking an hour or more) only to fail with a SMACH namespace error. A way to "compile" or otherwise error check the code before running it would be invaluable. Ultimately we addressed this shortcoming by implementing breakpoints between states that made it easy to "jump around" the FSM to resume recipe execution from near the previous failure point, minimizing wasted time while testing.

Benefits of Consistent Breakpoints

We decided at the start of BakeBot development that action sequences, when possible, would start and end in the *standard configuration*. This made it possible to run baking subtasks in virtually arbitrary order and made it easy to ensure a consistent setup before starting a specific action sequence. This feature was invaluable when restarting the system

and continuing from where it left off. We implemented several flow-control states that enabled the user to skip parts of the action sequence to facilitate debugging of specific system components.

Servoing vs Mapping

Rather than maintaining a map of the environment, we performed our detections immediately before they were needed. For example, we re-detected the ingredient bowl and the mixing bowl immediately before starting the grasp procedure to pour the ingredient into the mixing bowl. This enabled us to eliminate error in their positions that accumulated since their previous detections.

We also servoed to objects once we were close enough to find them. For example, when driving the robot to the oven to open it, we servo to the oven based on its detected handle position rather than relying on the odometry to get us into the exact right position. When returning to the table, once the robot was close it would do a broad tabletop detection to find the cookie sheet, then servo based on that detection to get to a position from which it could reliably grasp the cookie sheet.

Benefits of Bottom-Up Development

Abstracting away the low level code into client classes made implementing the `SMACH` state machine much easier. Since the bulk of the sensor processing and actuation work was done by the client classes, the state machine classes were usually pretty minimal container classes. In addition to facilitating rapid development of the state machine, this facilitated debugging and refinement: fixes only needed to be made once in the client classes to correct an error across the entire state machine. When modifications had to be made to account for changes in the ROS substate, only the client classes had to be fixed, leaving the state machine unaffected.

State Reuse and Parameterization

Figure 4-2 shows the breakdown of the high-level baking subtasks into lower level action sequences. We were able to identify commonalities between the subtasks and create versatile motion primitives that accomplished the subtasks based on their runtime parameterizations. Figure 4-11 shows the motion primitive set mapped onto the lower level actions sequences. In practice, we implemented this by creating generic `SMACH` states that were parametrized at runtime by parameters stored in the `userdata` container.

The `GRAB` state attempts to grasp whatever is stored in the `userdata.object_of_desire` variable. This enables the same `GRAB` state to be placed in several different finite state machines (for collecting an ingredient or scraping, for example) without having to be modified. The pre-pour movements and the `POUR` state were parameterized based on the latest detected pose of the mixing bowl and the pose of the ingredient bowl in the gripper, stored in the `userdata.current_known_pose_name` variable. Discarding an empty container acted on the original pose of the object in hand (`userdata.grasp_pose`) and on the position of the backup dump location. In the case of the *Execute Compliant x Trajectory* primitive, we found it most useful to accomplish this parameterization in the mixing client rather than in the `SMACH` states.

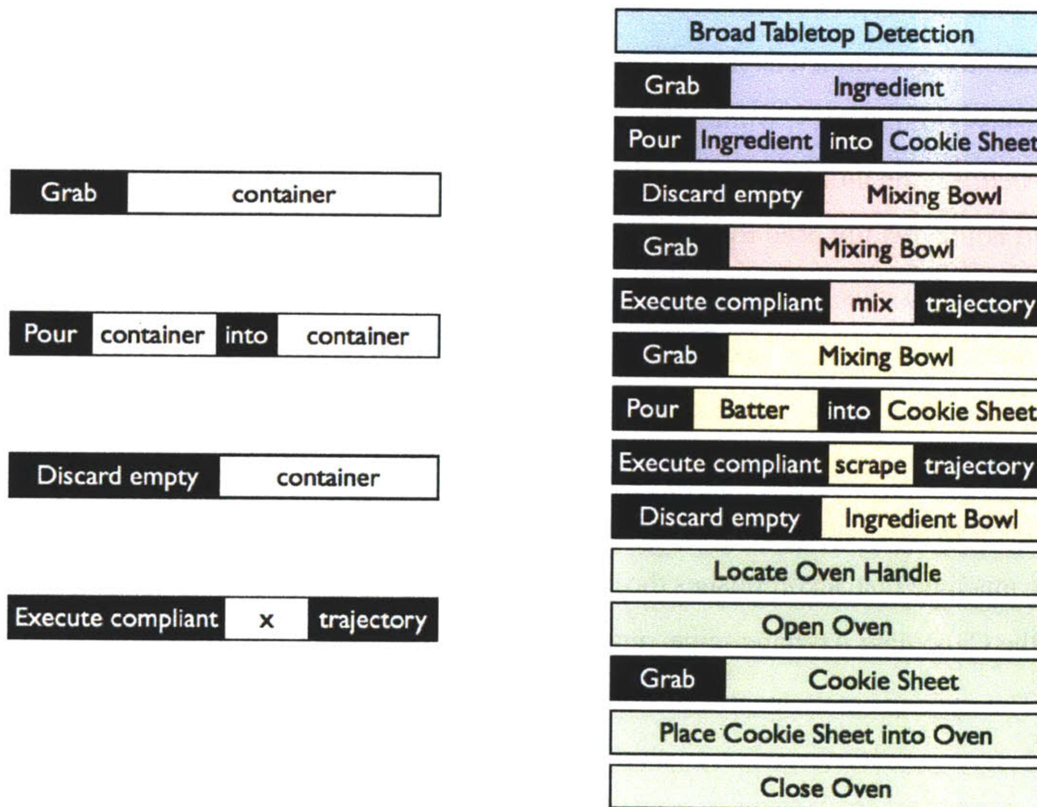


Figure 4-11: The motion primitive set (left) mapped onto the lower level action sequences (right).

4.4 Dynamic State Machine

The static finite state machine served as a proof-of-concept of the BakeBot system: it demonstrated that the individual subcomponents could be combined into a functional end-to-end baking system. This implementation had several shortcomings, however. It was cumbersome to handle all of the possible failure modes: sometimes the system would abort task execution because no state transition had been programmed to address a specific failure, even though a casual observer would find an intuitive work-around that would allow the task to be completed. The static finite state machine also had limited applicability to other tasks and recipes, while the `DEAL MASTER` state could accept different sets of *baking instructions*, the motion primitives remained trapped within their finite state machines and could not be readily used to accomplish other tasks.

To address this, we implemented a dynamic state machine using the `smachforward` framework discussed in Section 3.5. Our implementation had separate dynamic state machines for each of the subtasks and assembled them together based on the set of *baking instructions* the interpreted from the recipe. Though it was outside the scope of this thesis, future work can combine the subtasks and their associated primitives into a single dynamic state machine that accomplishes the entire baking task. Figure 4-13 shows the dynamic fsm for the Chocolate Afghan recipe, analogous to the static fsm shown in 4-6.

The dynamic state machine implementation, henceforth BakeBot, is outlined in Figure 4-12. It:

1. processes natural language baking recipes into a set of *baking instructions* using the Recipe Parsing system described in Section 3.4,
2. compiles the *baking instructions* into a hierarchical dynamic state machine, as represented by Figure 4-13,
3. executes the hierarchical dynamic state machine and returns the result.

Each dynamic sub-state machine for subtask execution was prototyped in PDDL and then implemented using `smachforward`. The action sequences for each subtask follow

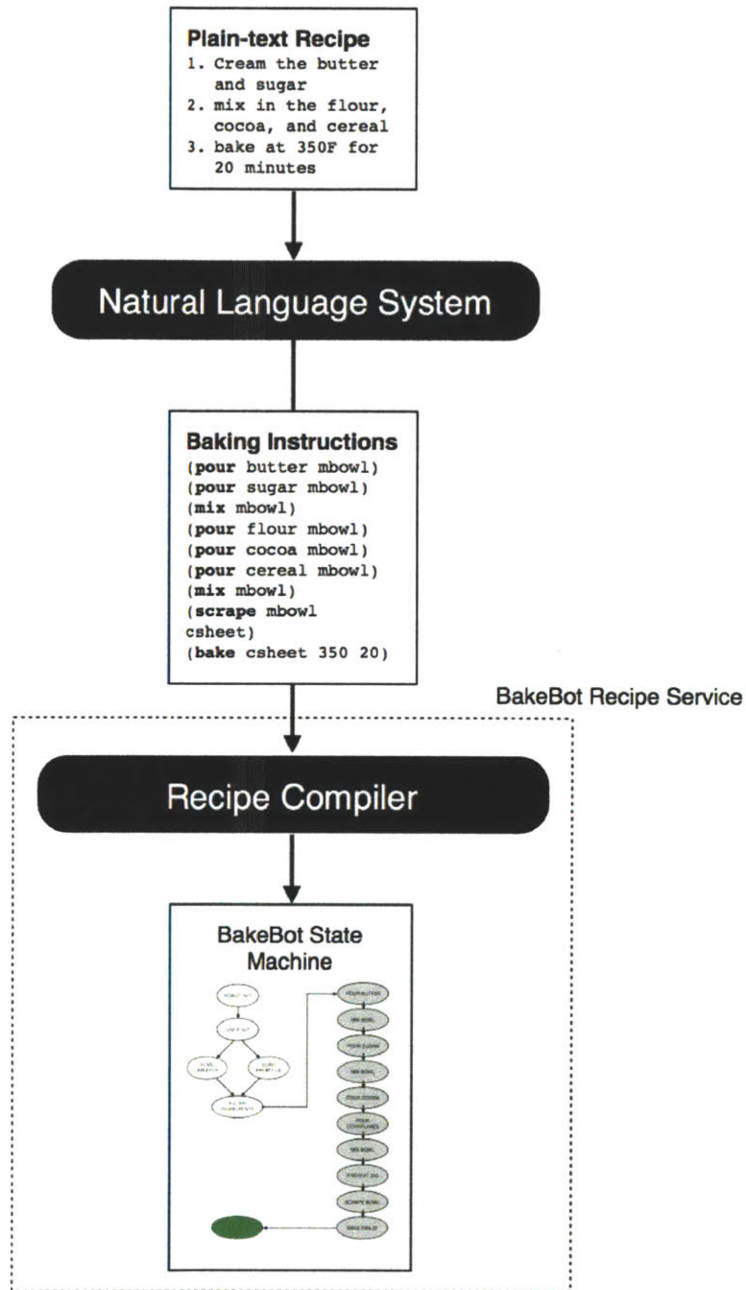


Figure 4-12: The end-to-end BakeBot system. The natural language recipe is processed into *baking instructions* which are compiled into a hierarchical dynamic state machine (as represented by Figure 4-13) and executed.

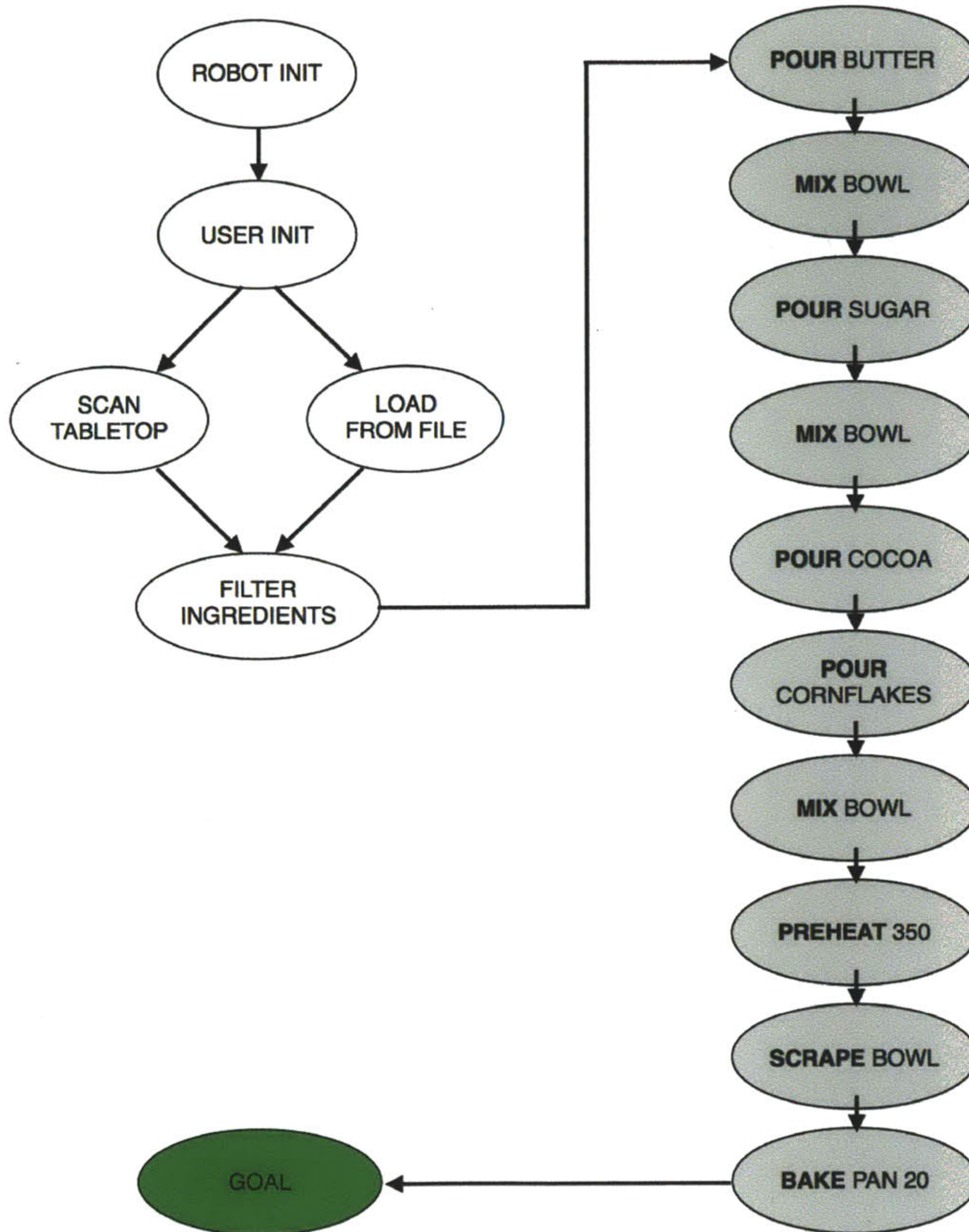


Figure 4-13: The high level BakeBot state machine for Chocolate Afghan cookies. Shaded states represent dynamic sub-state machines. This state machine is assembled at runtime to execute a set of *baking instructions*. Each shaded dynamic sub-state machine assembles sequences of motion primitives to accomplish its subtask at runtime (as discussed in Section 3.5).

the general activity sequence outlined in Sections 4.2 and 4.3. The implementation of the subtasks as dynamic state machines is discussed below.

4.4.1 Recipe Compiler

The Recipe Compiler constructs the BakeBot State Machine, a static finite state machine with each subtask implemented as a nested dynamic state machine, to execute the set of *baking instructions* received from the natural language system. The Recipe Compiler is simple, it performs string matching on each line of the baking instructions to parse the instruction and arguments. Supported instructions were: *pour, mix, scrape, preheat*, and *bake*. It uses factory methods to create dynamic state machines to accomplish the instructions and adds these state machines to the subtask sequence. An example subtask sequence is seen in grey in Figure 4-13.

The Recipe Compiler ignores comments (lines starting with "#") in the *baking instructions*. It uses ingredient declaration lines to instruct BakeBot's human partner which ingredients are needed to execute the recipe. Using the ingredient map provided by the human, the Recipe Compiler eliminates pour actions for ingredients that are already stored in the mixing bowl (such as the butter for the Chocolate Afghan recipe). It also eliminates the mix actions that were associated with those pour actions, preventing two mixes from being performed sequentially unnecessarily. Preheat instructions are pulled out of the robot instruction set and are included in the preparation instruction set presented to the human partner.

Handling Unknown Instructions

When the Recipe Compiler receives an unsupported *baking instruction* it constructs a *user interaction state* in the action sequence for recipe execution. When encountered during recipe execution (runtime), the *user interaction state* audibly (via the PR2 voice synthesizer) and visibly (via terminal output) asks the user for assistance. The user is presented with the unsupported *baking instruction* and is asked to perform the task. The user hits ENTER when the task is complete and the *user interaction state* terminates with success,

continuing to the next activity in the action sequence.

4.4.2 Collecting an Ingredient

Breaking the ingredient collection task into actions that could be represented in terms of PDDL boolean preconditions and effects required the definition of the following predicates:

`(INGRED ?X)` : True if `?X` is an ingredient bowl.

`(GRIPPER ?X)` : True if `?X` is a robot manipulator.

`(carry ?X ?Y)` : True if `?Y` is a robot manipulator and `?X` is an ingredient bowl.

`(free ?X)` : True if `?X` is an empty robot manipulator.

`(poured-out ?X)` : True if `?X` is an ingredient bowl whose contents have been poured into the mixing bowl.

`(in-transit ?X)` : True if `?X` is an ingredient bowl in the manipulator over the height of the other bowls on the table. In this region the bowl can be moved laterally across the entire table without fear of collision with objects on the table surface.

`(on-table ?X)` : True if `?X` is an ingredient bowl resting on the table surface.

`(over-mb ?X)` : True if `?X` is an ingredient bowl in the manipulator over the mixing bowl.

`(inverted ?X)` : True if `?X` is an ingredient bowl inverted in the middle of the pour maneuver.

`(cardinal ?X)` : True if `?X` is an ingredient bowl held in the manipulator at one of the cardinal bowl pose discretizations.

`(randomp ?X)` : True if `?X` is an ingredient bowl held in an arbitrary orientation (not a cardinal bowl discretization).

`(done ?X)` : True if `?X` is an ingredient bowl which has been dumped into the bowl storage off of the table (after being poured out).

`(reset ?X)` : True if `?X` is a robot manipulator that is resting empty beside the robot in the *standard configuration*.

The `deal_bowl_predicate_estimator` was implemented to evaluate these predicates within the `smachforward` framework.

The SMACH states in the static finite state machine were supplemented with PDDL precondition and effect statements, in terms of the predicates just defined, satisfying the `fstate` requirements and enabling them to be planned with in the `smachforward` framework. The following states, analogous to the states in the static finite state machine, were defined:

grab : grasps an ingredient bowl.

lift : lifts an ingredient bowl into the `in-transit` zone.

rotate-to-known : rotates an ingredient bowl to a cardinal bowl discretization.

transit : moves an ingredient bowl over the mixing bowl

pour : pours the contents of the ingredient bowl into the mixing bowl.

revert : returns the poured ingredient bowl to its original horizontal pose (undoes the pour).

dump : drops the ingredient bowl into the bin beside the table.

reset-arm : returns the manipulator to the *standard configuration*.

We described the completion of the ingredient collection task in terms of the defined predicates:

```
(:goal (and (INGRED ingred)
            (GRIPPER gripper)
            (poured-out ingred)
            (reset gripper)))
```

4.4.3 Mixing the Mixing Bowl

Breaking the mixing task into actions that could be represented in terms of PDDL boolean preconditions and effects required the definition of the following predicates:

`(GRIPPER ?X)` : True if `?X` is a robot manipulator.

`(SPATULA-GRIPPER ?X)` : True if `?X` is the robot manipulator with the spatula bolted to it.

`(MIXING-BOWL ?X)` : True if `?X` is the mixing bowl.

`(carry ?X ?Y)` : True if `?Y` is a robot manipulator and `?X` is an ingredient bowl.

`(free ?X)` : True if `?X` is an empty robot manipulator.

`(over-mb ?X)` : True if `?X` is a manipulator over the mixing bowl.

`(reset-safe ?X)` : True if `?X` is a manipulator that is not inside the mixing bowl or holding the mixing bowl. The manipulator can be reset to the *standard configuration* without disrupting anything.

`(in-mb ?X)` : True if `?X` is a manipulator inside the mixing bowl.

`(mix-zone ?X)` : True if `?X` is the mixing bowl and it is in the ideal mixing region relative to the robot (as discussed in Section 3.3.2).

`(mixed ?X)` : True if `?X` is a mixing bowl that has been mixed.

`(reset ?X)` : True if `?X` is a robot manipulator that is resting beside the robot in the *standard configuration*.

The `mixing_predicate_estimator` was implemented to evaluate these predicates within the `smachforward` framework.

The `SMACH` states in the static finite state machine were supplemented with PDDL precondition and effect statements, in terms of the predicates just defined, satisfying the

`fstate` requirements and enabling them to be planned with in the `smachforward` framework. The following states, analogous to the states in the static finite state machine, were defined:

servo-to-mb : servos the robot base to bring the mixing bowl into the ideal mixing position.

servo-from-mb : servos the robot base to return it to the starting position.

grab-mb : grasps the mixing bowl.

transit-spoon : moves the spatula from the starting position to above the mixing bowl.

plunge-spoon : plunges the spatula down into the mixing bowl.

mix : mixes the mixing bowl.

deplunge-spoon : removes the spatula from the mixing bowl.

retreat-spoon : returns the spatula to its position beside the robot.

retreat-grab : releases the grasp of the mixing bowl and returns the manipulator to its position beside the robot.

reset : returns the manipulator to the *standard configuration*.

We described the completion of the mixing task in terms of the defined predicates:

```
(:goal (and (MIXING-BOWL mb)
            (GRIPPER larm)
            (GRIPPER rarm)
            (SPATULA-GRIPPER rarm)
            (mixed mb)
            (reset larm)
            (reset rarm)))
```

4.4.4 Scraping the Cookie Sheet

Breaking the scraping task into actions that could be represented in terms of PDDL boolean preconditions and effects required the definition of the following predicates:

`(GRIPPER ?X)` : True if `?X` is a robot manipulator.

`(MIXING-BOWL ?X)` : True if `?X` is the mixing bowl.

`(COOKIE-SHEET ?X)` : True if `?X` is the mixing bowl.

`(SPATULA-GRIPPER ?X)` : True if `?X` is the robot manipulator with the spatula bolted to it.

`(carry ?X ?Y)` : True if `?Y` is a robot manipulator and `?X` is an ingredient bowl.

`(free ?X)` : True if `?X` is an empty robot manipulator.

`(over-cs ?X)` : True if `?X` is a mixing bowl and it is over the cookie sheet.

`(reset-safe ?X)` : True if `?X` is a manipulator that is not inside the mixing bowl or holding the mixing bowl. The manipulator can be reset to the *standard configuration* without disrupting anything.

`(on-table ?X)` : True if `?X` is a mixing bowl resting on the table surface.

`(in-transit ?X)` : True if `?X` is a manipulator in the manipulator over the height of the other bowls on the table. In this region the bowl can be moved laterally across the entire table without fear of collision with objects on the table surface.

`(done ?X)` : True if `?X` is the mixing bowl which has been dumped into the bowl storage off of the table (after being poured out).

`(in-mb ?X)` : True if `?X` is a manipulator inside the mixing bowl.

`(out-of-way ?X)` : True if `?X` is the mixing bowl and it is held to the side of the robot in the transit zone to enable an unobstructed view of the cookie sheet.

`(cs-zone ?X)` : True if `?X` is the cookie sheet and it is in the ideal scraping region relative to the robot (as discussed in Section 3.3.2).

`(scraped ?X)` : True if `?X` is a mixing bowl that has been scraped.

`(randomp ?X)` : True if `?X` is a mixing bowl not held at one of the cardinal bowl pose discretizations.

`(cardinal ?X)` : True if `?X` is a mixing bowl held in the manipulator at one of the cardinal bowl pose discretizations.

`(inverted ?X)` : True if `?X` is a mixing bowl inverted in the middle of the pour maneuver.

`(poured-out ?X)` : True if `?X` is a mixing bowl whose contents have been poured onto the cookie sheet.

`(reset ?X)` : True if `?X` is a robot manipulator that is resting beside the robot in the *standard configuration.*

The `scraping_predicate_estimator` was implemented to evaluate these predicates within the `smachforward` framework.

The `SMACH` states in the static finite state machine were supplemented with PDDL precondition and effect statements, in terms of the predicates just defined, satisfying the `fstate` requirements and enabling them to be planned with in the `smachforward` framework. The following states, analogous to the states in the static finite state machine, were defined:

grab-mb : grasps the mixing bowl.

lift : lifts the mixing bowl into the `in-transit` zone.

rotate-to-known : rotates the mixing bowl to a cardinal bowl discretization.

servo-to-cs : servos the robot base to bring the cookie sheet into the ideal scraping position.

servo-from-cs : servos the robot base to return it to the starting position.

transit-aside : moves the mixing bowl out of the way (to the side) so that the cookie sheet can be detected.

transit : moves the mixing bowl over the cookie sheet.

pour : pours the contents of the mixing bowl into the cookie sheet.

transit-spoon : moves the spatula from the starting position to in front of the upturned mixing bowl.

plunge-spoon : plunges the spatula sideways into the upturned mixing bowl.

scrape : scrapes the batter out of the mixing bowl.

deplunge-spoon : removes the spatula from the mixing bowl.

retreat-spoon : returns the spatula to its position beside the robot.

revert : returns the poured mixing bowl to its original horizontal pose (undoes the pour).

dump : drops the mixing bowl into the bin beside the table.

reset-arm : returns the manipulator to the *standard configuration*.

We described the completion of the scraping task in terms of the defined predicates:

```
(:goal (and (MIXING-BOWL mb)
            (GRIPPER larm)
            (GRIPPER rarm)
            (SPATULA-GRIPPER rarm)
            (scraped mb)
            (reset larm)
            (reset rarm)))
```

4.4.5 Putting Cookie Sheet Into Oven

Breaking the mixing task into actions that could be represented in terms of PDDL boolean preconditions and effects required the definition of the following predicates:

`(COOKIE-SHEET ?X)` : True if `?X` is the cookie sheet.

`(ROBOT ?X)` : True if `?X` is a robot.

`(OVEN ?X)` : True if `?X` is an oven.

`(aligned ?X ?Y)` : True if `?X` is a robot, `?Y` an oven, and the robot is aligned with the oven.

`(at-oven ?X)` : True if `?X` is a robot that is in front of the oven.

`(at-table ?X)` : True if `?X` is a robot that is in front of the table.

`(open ?X)` : True if `?X` is an oven that is open.

`(in-oven ?X)` : True if `?X` is a cookie sheet inside of the oven.

`(carry ?X ?Y)` : True if `?Y` is a robot and `?X` is a cookie sheet.

`(cooked ?X)` : True if `?X` is a cookie sheet that is cooked.

The `ovening_predicate_estimator` was implemented to evaluate these predicates within the `smachforward` framework.

The `SMACH` states in the static finite state machine were supplemented with PDDL precondition and effect statements, in terms of the predicates just defined, satisfying the `fstate` requirements and enabling them to be planned with in the `smachforward` framework. The following states, analogous to the states in the static finite state machine, were defined:

drive-to-oven : servos the robot base to bring it in front of the oven.

align : aligns the robot base with the oven.

open-oven : opens the oven.

drive-to-table : servos the robot base to bring it in front of the table.

grab : grasps the cookie sheet.

put-in-oven : puts the cookie sheet into the oven.

close-oven : closes the oven.

wait : waits for enough time for the contents of the cookie sheet in the oven to become cooked.

We described the completion of putting the cookie sheet into the oven task in terms of the defined predicates:

```
(:goal (and (COOKIE-SHEET cs)
            (OVEN oven)
            (cooked cs)
            (open oven)))
```

4.4.6 Discussion

Defining the baking subtasks as dynamic state machines was straightforward since we defined the smachforward framework exactly for the transition from a regular static SMACH finite state machine to a dynamic state machine. The bulk of the transition task was spent in describing the entire baking task in PDDL. Once the tasks were defined in PDDL, the existing SMACH states were modified and combined to correspond to the analogous states listed above. The states, with their associated PDDL preconditions and effects, were assembled into smachforward state machine containers.

Predicate estimators for each of the subtasks were implemented to translate the robot state information from the ROS system into predicates that could be used for task execution. Writing the predicate estimation code took quite a bit of development time, but paled in comparison to the amount of time needed to weave together the error recovery state transitions in the static finite state machine.

One noticeable improvement of the predicate estimation versus static finite state machine approach was in the ability to restart the system from virtually any position. For example, if the system crashed while an ingredient bowl was held over the mixing bowl ready to be poured, upon restarting the predicate estimator would detect the starting configuration and would plan an activity sequence for ingredient collection starting at the current position (rather than attempting to restart with the ingredient on the table and the robot in the *standard configuration*).

However, some bugs arose from porting some states to the dynamic state machine as a result of this versatility. In the static finite state machine, the exact order of state execution was known. In the dynamic state machine, however, the order of state execution was not exactly defined. In cases when the system was restarted mid-subtask, several states may not even be executed. Issues arose when required variables were not in the `userdata` namespace because the states in which they were initialized were not executed. Developing a dynamic finite state machine from scratch, rather than porting static finite state machine code, would prevent this issue from arising.

Once the porting and integration bugs were ironed out, we found the dynamic state machine to be more manageable and robust that the static state machine. Implementing new states and reusing states for other tasks was easy and did not require reconnecting any other states in the state machine. The dynamic state machine's ability to recover-in-place from a program restart made us more robust to planning and subsystem failure: the whole system could be restarted and the robot could resume where it left off as if nothing had happened.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Experimental Results

BakeBot testing mirrored its development and followed a bottom-up approach. Each subsystem of BakeBot was tested more than 100 times. We then tested the assembled static finite state machine, incorporated what we learned into the dynamic state machine, and tested the dynamic state machine. The natural language system was tested in parallel with the dynamic state machine as its testing could be conducted off of the robot in simulation. The *baking instruction* abstraction layer between the natural language system and the baking executive facilitated the parallel testing.

The bulk of BakeBot testing was conducted on the static finite state machine but is applicable to the dynamic state machine implementation as well. Under ideal operation (no error handling), the static and dynamic state machines follow the same sequence of motion primitives to accomplish the baking task. The functional code of the two systems is nearly identical, the primary difference between the two systems is in the planning and error-recovery phases of execution. During planning, the dynamic state machines that handle subtask execution are assembled into a similar action sequence as the static state machines. When task execution fails, however, the error-handling mechanisms are very different. The static state machine attempts to address the failure with with error-handling states (such as `ROTATE TO KNOWN`), while the dynamic state machine replans a new path of motion primitives to accomplish the subtask.

Our testing of the dynamic state machines, incrementally performed by folding the subtasks into the static finite state machine, focused on the differences between the two

systems: error-handling and recovery. Once we were confident in the performance of the dynamic state machines, full end-to-end system tests were performed, integrating the dynamic state machines with the natural language processing system.

5.1 Testing the Static Finite State Machine

The end-to-end static finite state machine system, executing a set of baking instructions to bake Chocolate Afghan cookies, was tested 27 times. The inputs to the system were:

1. the set of *baking instructions* describing the Chocolate Afghan cookie recipe,
2. the premeasured ingredients arrayed in the ingredient bowls on the preparation table,
3. a preheated toaster oven,
4. the cookie sheet and mixing bowl on the preparation table,
5. and a mapping between the ingredients on the table and their relative bowl positions to one another.

The outputs of the system were:

1. baked chocolate Afghans, sitting in the open toaster oven.

In all, 16 of the 27 tests ran to completion, with an average runtime of 142 minutes from start to finish. The plurality of the runtime was spent mixing the ingredients. Each of the two mixing actions takes, on average, 27 minutes to execute. Each of the four ingredient collections takes 8 minutes to execute. Table 5.1 shows the average runtime for each of the subtasks.

The goal of these experiments was been to establish the robustness of the end-to-end BakeBot system and to collect performance data for each operation, including time and feedback from the robot's sensors, such as the torques in Figure 3-6. Figure 5-1 shows the end-to-end static finite state machine tests plotted horizontally across the task space. The tests are plotted chronologically, from top to bottom. Circles on the figure represent failures that required human intervention to correct. Minor failures, such as the fingers slipping off

Table 5.1: Average runtimes for each of the baking subtasks for Chocolate Afghans.

Subtask	Avg. Duration [sec]	# per batch	Avg. Total Time [sec]
Broad tabletop detection	118	1	118
Adding an ingredient	477	4	1908
Mixing	1610	2	3220
Scraping onto cookie sheet	978	1	978
Putting into oven	1105	1	1105

of the oven door halfway through the opening procedures, were corrected during runtime and the tests were allowed to continue. More serious failures that required the system to be fully restarted or a new piece of code to be written caused the termination of the test.

Clusters of failures occurred during the pour phase of the third and fourth ingredient collection. These are attributed to failures of the inverse kinematic planning system on the PR2. Several kinds of errors were experienced during these pour maneuvers:

- the planner would seize and require a restart of the ROS manipulation pipeline;
- after executing the planned inverse kinematic trajectory with the manipulator the controller that manages the trajectory fails to yield control, requiring a restart of the ROS manipulation pipeline;
- the collision/contact map becomes corrupted, causing the planner to believe the robot is in a state of collision from which it is unable to plan, requiring a restart of the ROS manipulation pipeline;
- or the planner cannot find an inverse kinematic path to the desired pose.

In nearly all cases the robot was able to continue with the baking task after being restarted. When restarted, the bowl is reset onto the tabletop and BakeBot repours the ingredients from that bowl into the mixing bowl. Being able to execute the same planning task after a system restart reinforces the hypothesis that these failures are attributed to the inverse kinematic planning system.

Other failures were less consistent than the planning problems. They were the result of the inability of the static finite state machine to deal with every possible error condition. For

example, the tabletop manipulation package occasionally chose to grasp the ball of batter in the center of the mixing bowl, rather than the rim of the bowl, because the batter presented a more salient grasp feature than the rim of the bowl. While this failure only occurred twice during testing, the large number of environmental interactions and sequential subtasks to complete the baking task make BakeBot particularly sensitive to even low probabilities of subsystem failure. The inability of the static finite state machine to deal with these kinds of failures, coupled with its rigidity and the lack of reusable motion primitives, motivated us to reimplement BakeBot as a dynamic finite state machine.

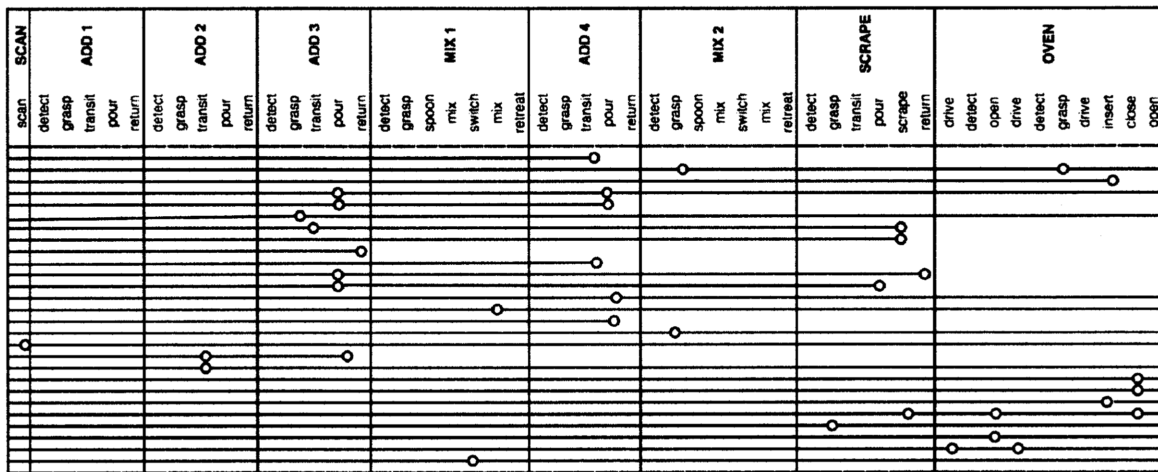


Figure 5-1: This chart shows the failure modes of the baking attempts using the static finite state machine to execute *baking instructions* to produce Chocolate Afghan cookies. Trials are plotted chronologically from top to bottom. A trial starts at the left with tabletop detection and ideally complete at the right with an oven opening. Circles represent failures that required human intervention. Trials that end in a circle represent a failure that could not be quickly recovered from, while other trials were able to recover with slight human interaction and continue.

5.2 Testing the Recipe Processing

Testing the recipe processing was primarily performed in simulation, as the system did not require the physical robot to run. Section 3.4.4 discusses the testing of the recipe processing in more detail. Once tested on its own, the recipe processing system was integrated into the dynamic state machine.

5.3 Testing the Dynamic State Machine

The dynamic state machine subtasks were tested incrementally by replacing the static finite state machines in the previous BakeBot framework with dynamic state machine implementations. Once we were confident in all of the new subtask executions, we combined the dynamic state with the natural language system, creating a new end-to-end system, shown in Figure 4-12. This end-to-end system was tested five times: three executions of the same Chocolate Afghan cookie recipe followed by the static finite state machine; and two executions of "Quick and Easy Sugar Cookies." The inputs to the system were:

1. a plain-text English recipe, selected by the user via the Recipe Browser user interface,
2. the premeasured ingredients arrayed in the ingredient bowls on the preparation table,
3. a preheated toaster oven,
4. the cookie sheet and mixing bowl on the preparation table,
5. and a mapping between the ingredients on the table and their relative bowl positions to one another.

The outputs of the system were:

1. baked cookies according to the selected recipe, sitting in the open toaster oven.

All five tests succeeded, resulting in cookies. Figure 5-2 shows photographs of BakeBot executing a recipe from start to finish.

The dynamic state machine, while susceptible to the same kind of planner failures as the static state machine, was able to recover more gracefully. A system restart would enable the system to pick up right where it left off, without having to redo the subtask.

Recovery from unexpected failures, like grasping the batter by mistake, was also improved. When a grasp was accidentally planned on the batter, the system would notice the failure and recover: grasping the batter would succeed, but lifting would fail because the manipulator would be in collision with the detected bowl on the table (which was not attached to the hand since the smaller batter grasp feature was attached), the failure of the

lifting would force a replan, which would note that the gripper was empty and re-attempt to grasp the bowl, merging back onto the correct motion primitive path.

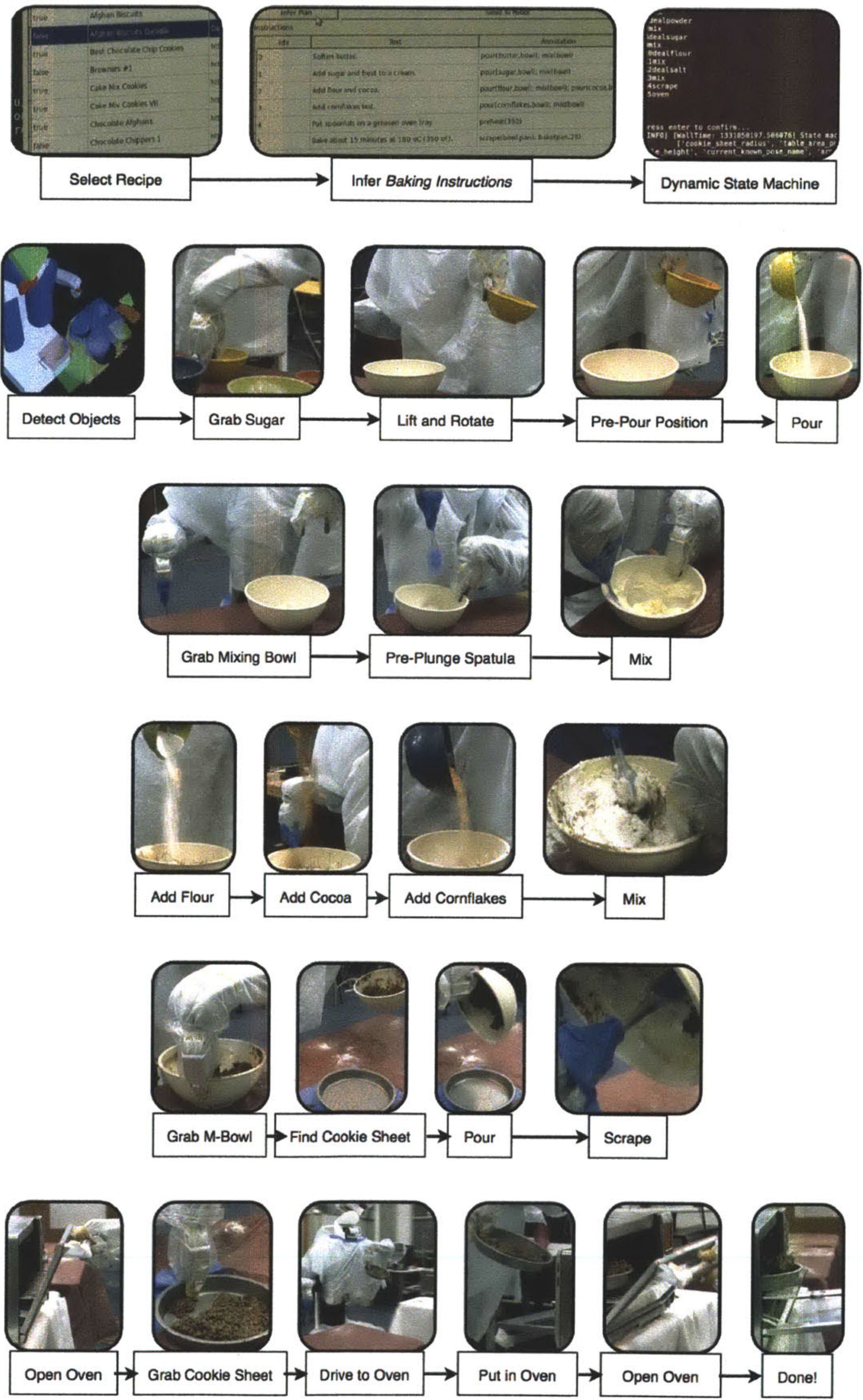


Figure 5-2: The steps of executing the Chocolate Afghan recipe with the BakeBot system.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Discussion

Our goal is to create a robot chef that is able to execute recipes in standard kitchens from arbitrary recipes downloaded from the Internet. BakeBot represents our early progress towards this goal, demonstrating that a task-based hierarchical state machine combined with basic planning and manipulation are sufficient to accomplish the complicated task of baking cookies. The more general application of the project is twofold: BakeBot demonstrates a use of compliant and force control to compensate for uncertainty in the environment and BakeBot uses a set of hierarchical subtasks composed of motion primitives that can be modified and re-ordered to achieve many different types of household goals.

6.1 Compliant Control Techniques

In the process of making cookies, BakeBot must work with many different objects and surfaces in the environment. Modeling each of these surfaces is an infeasible task, so for many motion primitives, such as mixing and scraping, we use compliant control algorithms that do not require models. For example, we use a force trajectory for the task of scraping the spatula against the side of the mixing bowl. Instead of trying to precisely model the rim of the bowl and scraping along it using a complicated position-controlled trajectory, we use a force trajectory that directs the spatula downwards and back towards the robot. The result is that the spatula is lowered to, and scraped along, the rim of the bowl without ever using precise knowledge about the position or shape of the rim. BakeBot uses this strategy

effectively in multiple motion primitives, increasing its versatility and robustness as these trajectories require very little a priori knowledge about the environment and adapt well to uncertainty.

BakeBot also uses hybrid position and force trajectories to bypass the need for complicated planning in constrained situations. For example, in opening the oven, BakeBot uses position control in one dimension and force control in the other. The result is a robust opening trajectory that is indifferent to the positioning of the oven relative to the robot and the robot's grasp of the oven handle. Unlike most algorithms for opening a door, this trajectory requires no planning time and no information about the oven except the width of its door.

6.2 Task Planning

The world of BakeBot is non-deterministic and there are many possible failure modes for each primitive. By designing a set of hierarchical subtasks and identifying these failure modes, we have a versatile set of subplans that could be used with many different task-level symbolic planners for many different tasks. For example, the oven-opening subtask can be re-parameterized for cupboard opening, while the ingredient-collecting task could just as easily be used to put bowls away. There are many such kitchen tasks that could be done with only re-parameterizations and re-orderings of the tasks used in BakeBot, and many more that would require the introduction of only a small number of extra tasks.

The advantages of task planning over static finite state machines were immediately apparent when BakeBot was reimplemented as a dynamic state machine. State estimation and replanning as an error-recovery technique offered more robustness than hard-coded state transitions because it enabled the system to respond to unexpected failure modes. Planning from a set of motion primitives enables us to and redefine tasks and apply the BakeBot primitive set to different manipulation problems. The greater encapsulation of primitives required by the dynamic state machine enabled us to reuse many of the primitives developed for use in ingredient collection and mixing for scraping the batter out of the mixing bowl.

The `smachforward` framework enabled us to combine static and dynamic state ma-

chines, focusing our development effort on planning the interesting parts of the baking problem. The hybrid state machine, assembled based on *baking instructions* from the Recipe Compiler, containing static components for setup and object detection and dynamic components for task execution was an effective engineering solution. Our testing found the solution to be robust and versatile.

6.3 Large Robotic Software Systems

BakeBot is a very large software system containing about 20,000 lines of code across the `bakebot`, `ee_cart_imped`, `smachforward`, and `htlogger` packages. This pales in comparison to the size of the ROS software running on the PR2, whose size we cannot begin to measure.

6.3.1 ROS

It would not have been possible to develop BakeBot without the large ROS software base provided with the PR2. To be completely honest, I started working on this project an `lcm` (Lightweight Communication and Marshalling, a UDP-based bare-bones message passing system) fanatic and a ROS skeptic. The complexity of the PR2 and the quality of many of the software components to control it have changed my mind. ROS abstracted away many tedious aspects of the problem; sensor and actuator integration, point cloud clustering, collision map, coordinate frame transformations, and inverse kinematics were all provided by various nodes in the system. This is a huge improvement for the field of robotics, letting researchers (ourselves included) focus on the interesting research problems rather than reinvent the wheel over and over just to get the system up and running.

There were, however, some drawbacks to having such a wide array of software components at one's fingertips. Figuring out which component to use for a specific task was sometimes difficult; with many options one can easily become overwhelmed or lost comparing wiki pages. Documentation of the system was very component specific, with few pages that aggregated the many approaches to solving a problem. The tutorials for how to use the PR2 were great, especially the low-level tutorials that were designed to get some-

one up and running quickly. Unfortunately, as ROS was updated (it is currently on a 6 month major-release cycle) the tutorials for more advanced behaviors, such as arm movement with Cartesian orientation constraints on the end effector did not accurately reflect ROS best-practice. Having a few examples of how to implement the manipulation system, like those in the tabletop manipulation and pick and place stacks, were invaluable. Much of BakeBot's low level manipulation code is adapted from these.

The encapsulation of many of the ROS subsystems into independent nodes made it easy to swap components in-and-out and made the overall system relatively robust to failures of a single component. Unfortunately, if a key component (such as the inverse kinematic planner) hung or crashed, there were few supervisory components that could detect the failure and allow the rest of the system to continue. This was the cause of many of the failures experienced during BakeBot end-to-end system testing.

Generally speaking, however, the distributed node approach was much better than having a single monolithic software process run the entire robot system: it let nodes be swapped or restarted without affecting the whole system, enabled the components to be developed in multiple programming languages, and offered relatively seamless running of nodes on various computers. That being said, BakeBot largely runs as a single node on top of the ROS system; this was much easier from a software engineering perspective as classes could communicate with each other in the same process rather than across messages.

ROS includes a number of tools for navigating the many packages within the PR2 software system. The ROS launch file system was an extremely effective way to launch all of the nodes and get the system ready for action. Dependencies in ROS are resolved within the each node's package description, making including code from other components straightforward. As a tool for accessing and navigating a broad software system updated by numerous development groups around the world, ROS was very effective.

Transitioning from simulation to the real robot was virtually transparent. Unfortunately, setting up the BakeBot task in simulation was impractical. The amount of objects (cookie sheet, mixing bowl, ingredient bowls, etc.) in the simulator limited the speed of the simulation considerably (to, at best, 0.5x realtime on our workstations). The impracticality of modeling the ingredients meant that the simulation had little value in predicting how Bake-

Bot interacts with the world. The simulator was useful, however, as a sanity check of the assembled software system. The state machines were dry-run in the simulator, with hardware calls commented out, to make sure that everything transitioned appropriately before booking time on the robot for actual testing.

6.3.2 Programming Languages

ROS supports two languages, Python and C++. We decided to implement BakeBot in Python because it is a better tool for quickly scripting robot action sequences. There were also key manipulation examples written in Python, such as the `pick_and_place_manager`, that we were able to base our code on. Several low-level components, such as the compliant control and mixing code, by necessity, were written in C++. This code was minimized whenever possible and we designed ROS messages to transition control to and from the high level Python executive classes.

`SMACH` was a great tool for assembling state machines of robot actions. Its lightweight state machine representation saved a considerable amount of development time. We were able to quickly prototype sequences of motion primitives and error-recovery states.

`SMACH` and Python together suffered from an inability for the system to self-check before execution on the robot. Countless instances of the robot proceeding through most of the baking task only to crash on a variable namespace error occurred during testing. A way to "compile" the code before running it would eliminate this problem. While some tools exist for Python to help with this, the `SMACH` `userdata` system complicates the namespaces considerably, rendering these tools useless.

6.3.3 Best Practices

Based on our experience developing BakeBot, we suggest the following best-practices:

- **Abstract away everything you don't need and use your own interfaces whenever possible**: our client classes dramatically reduced development time by giving us a single point of entry into the ROS system. This insulated us against changes to the

ROS system, reduced our memory footprint by enabling singleton connections, and provided a consistent and well-understood API.

- **Bottom-up development**: Our client classes did almost all of the "heavy lifting". Higher level classes reused the client classes and managed failure recovery and task execution. This enabled scripted sequences to test task execution to be quickly ported to static and dynamic state machines with minimal additional coding. Having a solid bottom-layer enabled us to focus our development and debugging effort on task execution.

- **Implement breakpoints**: Breakpoints let us resume the BakeBot system from most configurations, enabling us to restart the system from where it left off, rather than having to start from the beginning every time. Implementing a robust breakpoint system is absolutely worth the development time. This let us focus on testing new code rather than waiting for the robot to execute previously-tested components before getting to the new action sequences.

- **The robot is often the fastest simulator**: We can't emphasize enough how valuable it was to perform the majority of our testing on the PR2. Our system, by design, deals with unmodeled aspects of the real world. Simulation was slow and did not faithfully represent the complexity of the challenges faced by BakeBot.

6.3.4 Feature Requests

We also have a few features we'd love to see integrated into the ROS PR2 system:

- **Hierarchy**: The breadth of ROS is perhaps its greatest feature. However, we found it difficult to know which components to use for a particular task, or what messages or topics would give us the information we need. Implementing a few "bottleneck" nodes that provide single points-of-entry for common manipulation or perception tasks would be very useful. A "Clippy the Paperclip" tutorial aide, "It looks like you're trying to control the end-effector with orientation constraints, I can help with that!", would also accomplish this goal.

- **Type-checking**: We wasted a lot of time running tests that crashed due to easily preventable namespace or variable type issues. While Python was great for rapidly developing BakeBot, the overall size of the system made it difficult to manage. Adding a type checking system to sanity check code in the ROS system would be very useful. Adding ROS support for Java would also address this.

6.4 Scope and Generality

The BakeBot system, as is, can perform a wide variety of simple baking recipes. Support for other cooking subtasks to enable a wider variety of kitchen tasks to be performed by the robot, such as greasing a pie pan, cutting vegetables, and using a skillet can be added by implementing additional motion primitives for these actions and by extending the KitchenState MDP used by the Recipe Processing system. As the capabilities of the system increase, so do the challenges of: inferring robot action sequences from plain text recipes, manipulating kitchen tools, and perceiving of an increasingly complex environment.

The `smachforward` framework can be used to create dynamic state machines of motion primitives for any manipulation task on the PR2 platform. The `ee_cart_imped` controller can be used to define motion primitives that require compliant manipulation or force-controlled actions. The general system framework, composed of a natural language processing system coupled with a compiler that generates executable robot action plans, can be reapplied to other household tasks or to tasks within a manufacturing setting.

6.5 Future Research

The existing end-to-end BakeBot system opens up a number of research avenues. Future research can speed up the BakeBot manipulation pipeline by utilizing more advanced inverse kinematic and motion planners. The integration of external monitoring systems, via motion capture, vision, or state estimation can be explored to increase the system's ability to respond to difficult-to-sense failures. Human-robot interaction can be further utilized for error recovery and to expand the repertoire of the cooking system. We are excited about

the continuing research enabled by the functioning end-to-end system.

6.6 Conclusion

This thesis presented BakeBot, an end-to-end robotic system which is able to read and execute simple recipes. BakeBot combines compliant control techniques, motion planning, task planning, natural language processing and object recognition. We tested BakeBot extensively, successfully producing cookies at the end of 21 baking attempts.

BakeBot is a first step towards the general RoboChef, a robot which can cook any recipe for the benefit of its human partners. We are excited about extending BakeBot by adding support for additional kitchen tasks and improving the manipulation and perception systems, enabling the system to cook a wider array of recipes without human intervention and bringing us another step closer to RoboChef.

Appendix A

Running BakeBot

A.1 System Requirements

BakeBot requires a PR2 robotic manipulation platform. The software system was designed for the `Diamondback` version of ROS.

The natural language processing system was developed by the Robust Robotics Group in MIT CSAIL. It has not yet been released outside of the MIT community.

A.2 Running the System in Simulation

The BakeBot system can be started in simulation but cannot run end-to-end within the simulator due to the slow simulation speed and the inability of the simulator to reasonably model the ingredients within the bowls. We do not recommend that the system be run in simulation.

That being said, there are two options for running BakeBot in simulation:

1. running the full Gazebo simulator on the local machine,

2. running a headless Gazebo simulator on a server

A.2.1 Running the simulator locally

To simulate the BakeBot system locally, which we don't recommend if you are running on a computer without a lot of RAM and many processor cores:

1. In Terminal 1 (local): `roslaunch bakebot sim_local_gazebo_server.launch`

2. In Terminal 2 (local): `roslaunch bakebot bakebot_services.launch`

3. In Terminal 3 (local): `rosrun bakebot bakebot_recipe_service.py`

4. In Terminal 4 (local): `rosrun bakebot recipe_tester.py baking_instructions.txt`, where `baking_instructions.txt` is a text file with the baking instructions to be executed.

A.2.2 Running the simulator on a server

To simulate the BakeBot system with Gazebo running on a powerful server (with the ROS parameters correctly configured so that the server is the ROS master):

1. In Terminal 1 (server): `roslaunch bakebot sim_headless_server.launch`

2. In Terminal 2 (server): `roslaunch bakebot bakebot_services.launch`

3. In Terminal 3 (local): `rosrun bakebot bakebot_recipe_service.py`

4. In Terminal 4 (local): `rosrun bakebot recipe_tester.py baking_instructions.txt`, where `baking_instructions.txt` is a text file with the baking instructions to be executed.

A.3 Running the System on the PR2

We strongly recommend running the system on the PR2 rather than in simulation. The PR2 should not be used near food or liquids without its protective covering.

A.3.1 Covering the PR2

Figure A-1 shows the steps of covering the BakeBot. Standard surgical gowns are used, these were ordered from a local dentist office. The steps to cover BakeBot to protect it during a cooking task are:

1. Cut a surgical gown up the front of the garmet, splitting it evenly almost up to the collar. Drape each half of the gown over the upper arms of BakeBot and rubber band into place.

2. Cut the lower sleeves off of another surgical gown. Slide them over each of the forearms, with the elastic at the wrist. Cut two patches of gown, about $30cm \times 60cm$, and wrap these patches around the upper arms, ensuring that the arms are completely covered.

3. Tape the wrapped piece of gown in place, being sure that it is not taped to the other coverings. Test that the joints have full rotation. Use rubber bands as necessary to hold all the coverings to their arm segments. Test that the joints have full rotation. Place garbage bag over the robot base. Add full surgical gown over the front of the robot. Tape hanging gown sleeves to the back of the robot. Tape the sides of the front gown to the back of the robot torso.

4. Remove the finger pads on the right manipulator.

5. Insert two 3mm bolts through the exposed holes. Push a piece of foam over the bolts.

6. Slide the spatula onto the bolts, add another piece of foam, and close the manipulator. Ensure the bolts go through both holes on each side of the manipulator.

7. Slide up the plastic covering over the whole end effector. Rubber band in place. Tape the bottom to ensure food particles cannot enter.

8. Open left manipulator. Place inside plastic bag.

9. Attach plastic bag to manipulator with rubber bands. Wrap fingertips in rubber bands.

10. Put surgical cover over garbage bag to cover the whole base. Tape in place.

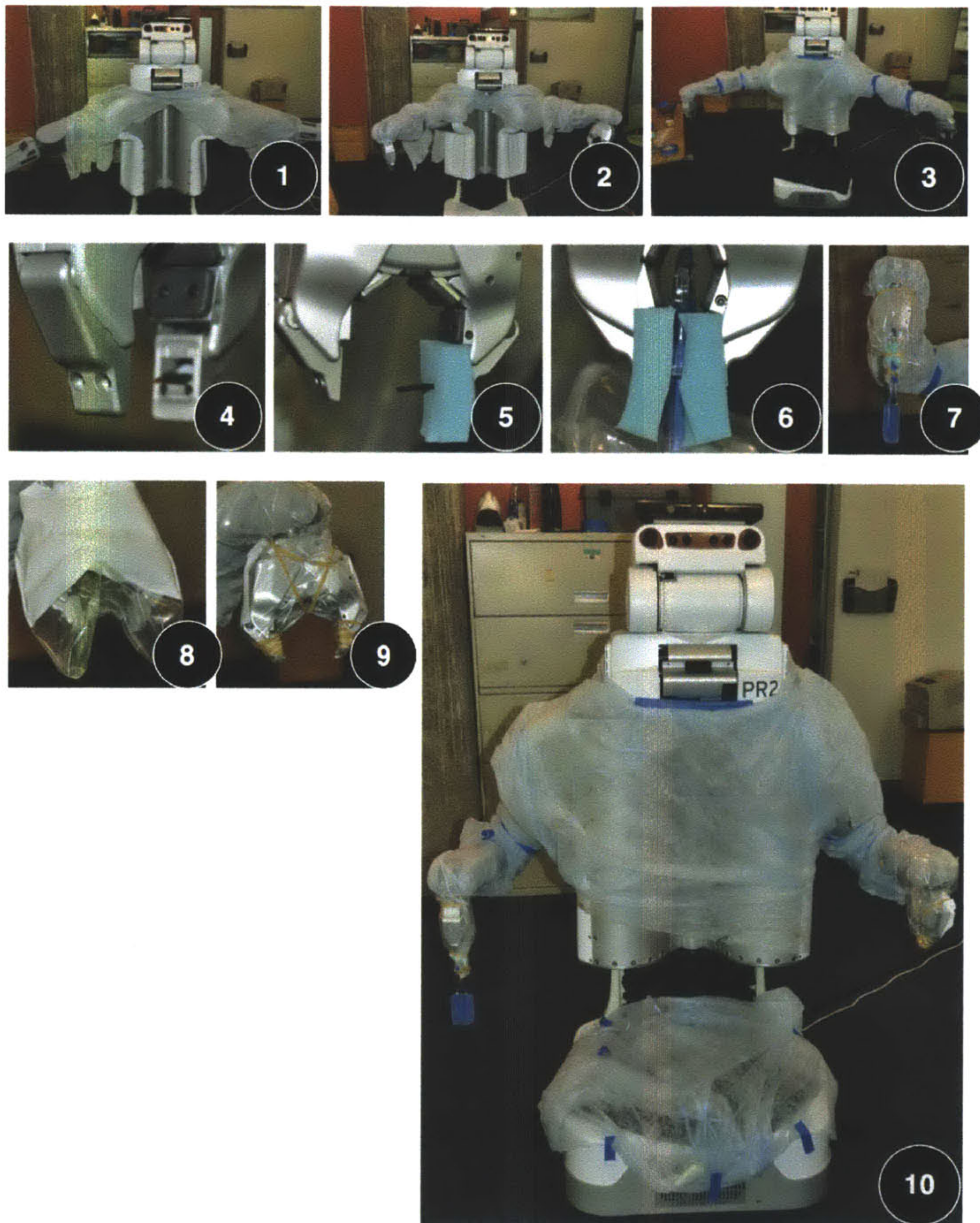


Figure A-1: The steps of covering the PR2 in its protective garmet.

A.3.2 Executing a Baking Instruction Set

To execute a set of *baking instructions* with the dynamic state machine:

1. In Terminal 1 (PR2): `robot start` (make sure that this launches ROS Diamondback)
2. In Terminal 2 (PR2): `roslaunch bakebot robot_tabletop.launch`
3. In Terminal 3 (PR2): `roslaunch bakebot bakebot_services.launch`
4. In Terminal 4 (PR2): `rosrun bakebot bakebot_recipe_service.py`
5. In Terminal 5 (PR2): `rosrun bakebot recipe_tester.py baking_instructions.txt`, where `baking_instructions.txt` is a text file with the baking instructions to be executed.

A.3.3 Executing Plain-Text Recipes

Executing plain-text recipes requires the installation of the SLU code from the Robust Robotics Group at MIT onto the local machine. To execute a plain-text recipe with the dynamic state machine:

1. In Terminal 1 (PR2): `robot start` (make sure that this launches ROS Diamondback)
2. In Terminal 2 (PR2): `roslaunch bakebot robot_tabletop.launch`
3. In Terminal 3 (PR2): `roslaunch bakebot bakebot_services.launch`
4. In Terminal 4 (PR2): `rosrun bakebot bakebot_recipe_service.py`
5. In Terminal 5 (local): From within the `slu/pytools/kitchen` directory: `rake train; rake recipe_corpus_browser`. Navigate to the recipe you'd like to execute. Click the recipe, then select "Infer Plan". Then click "Send to Robot."

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Software Overview

BakeBot software is divided among several ROS packages:

1. `bakebot`: containing all of the BakeBot clients, servers, and state machines. More information and links to the code can be found at `http://www.ros.org/wiki/bakebot`.

2. `smachforward`: containing the dynamic state machine framework. More information and links to the code can be found at `http://www.ros.org/wiki/smachforward`.

3. `ee_card_imped`: containing the force-compliant controller. More information and links to the code can be found at `http://www.ros.org/wiki/ee_cart_imped`.

4. and `opendoors_executive`: containing the code for opening the oven. More information and links to the code can be found at `http://www.ros.org/wiki/opendoors`.

The Recipe Processing code is within the SLU framework developed by the Robust Robotics Group at MIT CSAIL and is not included within any ROS package.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Afghan biscuits. http://australianfood.about.com/od/bakingdesserts/r/AfghanBiscuits.htm.

[2] Reduce melaboo mixing bowl, 5-piece set. http://www.amazon.com/Reduce-Melaboo-Mixing-Bowl-5-Piece/dp/B002D8X5RW/ref=sr_1_20?ie=UTF8&qid=1296672265&sr=8-20.

[3] Smach - ros wiki. http://www.ros.org/wiki/smach.

[4] H. Asada and J.J.E. Slotine. *Robot analysis and control.* Wiley-Interscience, 1986.

[5] J. Becker, C. Bersch, D. Pangercic, B. Pitzer, T. Rühr, B. Sankaran, J. Sturm, C. Stachniss, M. Beetz, and W. Burgard. The pr2 workshop-mobile manipulation of kitchen containers.

[6] Michael Beetz, Ulrich Klank, Ingo Kresse, Alexis Maldonado, Lorenz Mösenlechner, Dejan Pangercic, Thomas Rühr, and Moritz Tenorth. Robotic Roommates Making Pancakes. In *IEEE-RAS International Conference on Humanoid Robots*, 2011.

[7] Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. A maximum entropy approach to natural language processing. *COMPUTATIONAL LINGUISTICS*, 22:39–71, 1996.

[8] M. Bollini, J. Barry, and D. Rus. Bakebot: Baking cookies with the pr2 (in submission). In *Intelligent Robots and Systems, 2012. IROS 2012. IEEE/RSJ International Conference on.* IEEE, 2012.

[9] David L. Chen and Raymond J. Mooney. Learning to interpret natural language navigation instructions from observations. In *Proc. AAAI*, 2011.

[10] M. Ciocarlie, K. Hsiao, E.G. Jones, S. Chitta, R.B. Rusu, and I.A. Sucan. Towards reliable grasping and manipulation in household environments. In *Proceedings of RSS 2010 Workshop on Strategies and Evaluation for Mobile Manipulation in Household Environments*, pages 1–12, 2010.

[11] B.J. Cohen, S. Chitta, and M. Likhachev. Search-based planning for manipulation with motion primitives. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2902–2908. IEEE, 2010.

[12] F. Gravot, A. Haneda, K. Okada, and M. Inaba. Cooking for humanoid robot, a task that needs symbolic and geometric reasonings. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 462–467. IEEE, 2006.

[13] J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. *Arxiv preprint arXiv:1106.0675*, 2011.

[14] N. Hogan. Impedance control: An approach to manipulation. In *American Control Conference, 1984*, pages 304–313. IEEE, 1984.

[15] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical Planning in the Now. In *IEEE Conference on Robotics and Automation*, May 2011.

[16] C.C. Kemp, A. Edsinger, and E. Torres-Jara. Challenges for robot manipulation in human environments [grand challenges of robotics]. *Robotics & Automation Magazine, IEEE*, 14(1):20–29, 2007.

[17] Thomas Kollar, Stefanie Tellex, Deb Roy, and Nicholas Roy. Toward understanding natural language directions. In *Proc. ACM/IEEE Int'l Conf. on Human-Robot Interaction (HRI)*, pages 259–266, 2010.

[18] M.T. Mason. Compliance and force control for computer controlled manipulators. *Systems, Man and Cybernetics, IEEE Transactions on*, 11(6):418–432, 1981.

[19] Cynthia Matuszek, Dieter Fox, and Karl Koscher. Following directions using statistical machine translation. In *Proc. ACM/IEEE Int'l Conf. on Human-Robot Interaction (HRI)*, pages 251–258, 2010.

[20] D. Nau, T. C. Au, O. Ilghami, U. Kuter, W. J. Murdockand D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *International Journal of Artificial Intelligence*, 20:379–404, 2003.

[21] P. Pastor, M. Kalakrishnan, S. Chitta, E. Theodorou, and S. Schaal. Skill learning and task outcome prediction for manipulation. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3828–3834. IEEE, 2011.

[22] A. Perez, S. Karaman, M. Walter, A. Shkolnik, E. Frazzoli, and S. Teller. Asymptotically-optimal manipulation planning using incremental sampling-based algorithms. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2011.

[23] R.B. Rusu, I.A. Sucan, B. Gerkey, S. Chitta, M. Beetz, and L.E. Kavraki. Real-time perception-guided motion planning for a personal robot. In *Intelligent Robots and Systems, 2009.*, pages 4245–4252. IEEE, 2009.

[24] W. Selby, P. Corke, and D. Rus. Autonomous aerial navigation and tracking of marine animals. In *Proceedings of the 2011 Australasian Conference on Robotics and Automation*, pages 1–7. Australian Robotics & Automation Association, 2011.

[25] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. Teller, and N. Roy. Approaching the symbol grounding problem with probabilistic graphical models. *AI Magazine*, 32(4):64–76, 2011.

[26] Jason Wolfe, Bhaskara Marthi, and Stuart Russell. Combined Task and Motion Planning for Mobile Manipulation. In *Interational Conference on Automated Planning and Scheduling*, 2010.