Program Steering: Improving Adaptability and Mode Selection via Dynamic Analysis

by

Lee Chuan Lin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2004 [September 2004]

Author ....
Department of Electrical Engineering and Computer Science
August 12, 2004

Certified by ....
Michael D. Ernst
Assistant Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Program Steering: Improving Adaptability and Mode Selection via Dynamic Analysis

by

Lee Chuan Lin

Submitted to the Department of Electrical Engineering and Computer Science
on August 12, 2004, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

A multi-mode software system contains several distinct modes of operation and a controller for deciding when to switch between modes. Even when developers rigorously test a multi-mode system before deployment, they cannot foresee and test for every possible usage scenario. As a result, unexpected situations in which the program fails or underperforms (for example, by choosing a non-optimal mode) may arise. This research aims to mitigate such problems by training programs to select more appropriate modes during new situations. The technique, called program steering, creates a new mode selector by learning and extrapolating from previously successful experiences. Such a strategy, which generalizes the knowledge that a programmer has built into the system, may select an appropriate mode even when the original programmer had not considered the scenario. We applied the technique on simulated fish programs from MIT's Embodied Intelligence class and on robot control programs written in a month-long programming competition. The experiments show that the technique is domain independent and that augmenting programs via program steering can have a substantial positive effect on their performance in new environments.

Thesis Supervisor: Michael D. Ernst
Title: Assistant Professor

Acknowledgments

Portions of this thesis were previously published in the 2003 Supplementary Proceedings of the International Symposium on Software Reliability Engineering [LE03] and the 2004 Proceedings of the International Symposium on Software Testing and Analysis [LE04].

First and foremost, I would like to thank my advisor, Michael Ernst. During my last four years at MIT, he has been a great teacher, mentor, and colleague, and I am extremely lucky to have worked with him. He has demonstrated great leadership and patience, guiding and training me through the years when I was a student, a UROP, a teaching assistant, and finally a graduate researcher. His enthusiasm and energy, his brilliant insight and intelligence, and his passion and high standard of excellence constantly inspired me to perform my best.

I owe thanks to all the members in the Program Analysis Group for all their support and encouragement. In particular, I would like to thank Jeremy Nimmer, Mike Harder, Nii Dodoo, Ben Morse, Alan Donovan, Toh Ne Win, Steve Garland, David Saff, Stephen McCamant, Derek Rayside, Robert Seater, Yuriy Brun, Jeff Perkins, Adam Keizun, Carlos Pacheco, Meng Mao, and Galen Pickard.

I must especially thank Meng and Galen for their useful work in continuing the research project after I leave. Meng made valuable contributions to the reinforcement learning ideas and experiments found in Chapter 6.

I would also like to thank Bill Thies, Jasper Lin, Adrian Solis, Hubert Pham, Kathryn Chen, and Justin Paluska for their feedback and ideas.

Contents

List of Figures

Chapter 1

Introduction

Software failures often result from the use of software in unexpected or untested situations, in which it does not behave as intended or desired [Wei02]. Software cannot be tested in every situation in which it might be used. Even if exhaustive testing were possible, it is impossible to foresee every situation to test. This research takes a step toward enabling multi-mode software systems to react appropriately to unanticipated circumstances.

A multi-mode system contains multiple distinct behaviors or input–output relationships, and the program operates in different modes depending on characteristics of its environment or its own operation. For example, web servers switch between handling interrupts and polling to avoid thrashing when load is high. Network routers trade off latency and throughput to maintain service, depending on queue status, load, and traffic patterns. Real-time graphical simulations and video games select which model of an object to render: detailed models when the object is close to the point of view, and coarser models for distant objects. Software-controlled radios, such as cell phones, optimize power dissipation and signal quality, depending on factors such as signal strength, interference, and the number of paths induced by reflections. Compilers select which optimizations to perform based on the estimated run-time costs and benefits of the transformation.

In each of these examples, a programmer first decided upon a set of possible behaviors or modalities, then wrote code that selects among modalities. This code, the part of the system that handles mode selection and mode transitions, is the controller. The policy for selecting modes may be hard-coded or dependent upon configuration settings. We hypothesize that, for the most part, programmers effectively and accurately select modalities for situations that they anticipate. However, the selection policy may perform poorly in unforeseen circumstances. In an unexpected environment, the built-in rules for selecting a modality may be inadequate. No appropriate test may have been coded, even if one of the existing behaviors is appropriate (or is best among the available choices). For instance, a robot control program may examine the environment, determine whether the robot is in a building, on a road, or on open terrain, and select an appropriate navigation algorithm. But which algorithm is most appropriate when the robot is on a street that is under construction or in a damaged building? The designer may not have considered such scenarios. As another example of applicability, if a software system relies on only a few sources of information, then a single sensor failure may destabilize the system, even if correlated information is available. Alternately, correlations assumed by the domain expert may not always hold.

1.1 Program Steering: Motivation

Program steering is a technique for creating more adaptive multi-mode systems. The primary goal is to help controllers choose modes more intelligently, rather than creating new modes. Specifically, program steering is a series of programming, testing, and analysis steps that prepares systems for unexpected situations: The system first trains on known situations, then extrapolates knowledge from those training runs, and finally leverages that knowledge during unknown situations.

The technique is general enough to be applicable to a wide range of multi-mode systems and does not rely on any specific knowledge about the target system's problem domain. As a result, applying program steering to a system does not rely heavily on a domain expert. Previous approaches to adaptive systems using control theory or feedback optimization require human input or specific knowledge of the problem domain.

Although program steering does attempt to improve on the original programmer's code, the technique is designed to complement, not replace, any existing knowledge in the system. If the programmer knows about important constraints or other crucial facts about the original system, the new upgraded system can abide by those rules as well.

1.2 Sample applications

This section describes three application areas — routers, wireless communications, and graphics — in more detail. Our experiments (Chapters 4 and 5) evaluate two other domains: controllers for autonomous robots in a combat simulation and controllers for self-organizing fish programs.

Routers. A router in an ad hoc wireless network may have modes that deal with the failure of a neighboring node (by rebuilding its routing table), that respond to congestion (by dropping packets), that conserve power (by reducing signal strength), or that respond to a denial-of-service attack (by triggering packet filters in neighboring routers). It can be difficult for a programmer to determine all situations in which each of these adaptations is appropriate [BAS02], or even to know how to detect when these situations occur (e.g., when there is an imminent denial-of-service attack). Use of load and traffic patterns, queue lengths, and similar properties may help to refine the system's mode selector.

Wireless communications. Software-controlled radios, such as cell phones, optimize power dissipation and signal quality by changing signal strength or selecting encoding algorithms based on the bit error rate, the amount of interference, the number of multihop paths induced by reflections, etc. Modern radio software can have 40 or more different modes [Cha02], so machine assistance in selecting among these modes will be crucial. Additionally, a radio must choose from a host of audio compression algorithms. For instance, some vocoders work best in noiseless environments or for voices with low pitch; others lock onto a single voice, so they are poor for conference calls, music, and the like. A software radio may be able to match observations of its current state against reference observations made under known operating conditions to detect when it is operating in a new environment (e.g., a room containing heavy electrical machinery), being subject to a malicious attack (e.g., jamming), or encountering a program bug (e.g., caused by a supposedly benign software upgrade).

Graphics. Real-time graphical simulations and video games must decide whether to render a more detailed but computationally intensive model or a coarser, cheaper model; whether to omit certain steps such as texture mapping; and which algorithms to use for rendering and other tasks, such as when to recurse and when to switch levels of detail. Presently (at least in high-end video games), these decisions are made statically: the model and other factors are a simple function of the object's distance from the viewer, multiplied by the processor load. Although the system contains many complex parameters, typically users are given only a single knob to turn between the extremes of fast, coarse detail and slow, fine detail. Program steering might provide finer-grained, but more automatic, control over algorithm performance — for instance, by correlating the speed of the (relatively slow) rendering algorithm with metrics more sophisticated than the number of triangles and the texture.

1.3 Daikon: Dynamic Program Property Detector

Our technique requires a method of determining properties about each mode in the target system during training. We use the Daikon invariant detector, which reports operational abstractions that appear to hold at certain points in a program based on the test runs. Operational abstractions are syntactically identical to a formal specification, in that both contain preconditions, postconditions, and object invariants; however, an operational abstraction is automatically generated and characterizes the actual (observed) behavior of the system. The reported properties are similar to assert statements or expressions that evaluate to booleans (Chapter 2 gives some simple examples.). These properties can be useful in program understanding and testing conditions for switching between modes.

Daikon produces operational abstractions by a process called dynamic invariant detection [ECGN01]. Briefly, it is a generate-and-check approach that postulates all properties in a given grammar (the properties are specified by the invariant detection tool, and the variables are quantities available at a program point, such as parameters, global variables, and results of method calls), checks each one over some program executions, and reports all those that were never falsified. As for any dynamic analysis, the quality of the results depends in part on how well the test suite characterizes the execution environment. The results soundly characterize the observed runs, but are not necessarily sound with respect to future executions.

1.4 Outline

Chapter 2 gives an introduction to program steering and an example of the technique applied to a simple multi-mode system. Chapter 3 describes in detail the four steps of program steering: training, modeling, mode selector creation, and augmentation. Chapters 4 and 5 discusses two sets of experiments utilizing program steering, the first demonstrating that the technique is domain independent, and the second demonstrating that the technique improves adaptability in new and unexpected situations. Chapter 6 explains a different implementation of program steering that relies on unsupervised reinforcement learning for modeling and mode selection. Chapter 7 describes related work in creating adaptive systems and improving mode selection. Chapter 8 describes directions for future work, both in improving the program steering implementations and in finding new subject programs. Chapter 9 concludes.

Chapter 2

Program Steering Example

This chapter presents a simple example of program steering applied to a laptop display controller. Program steering starts from a multi-mode program. We illustrate the four steps in applying program steering — training, modeling, creating a mode selector, and integrating it with the original program — and show how the augmented program performs.

2.1 Original Program

Our original laptop display program has three possible display modes: Normal Mode, Power Saver Mode, and Sleep Mode. Normal mode is a standard operating mode that allows the user to set the display brightness anywhere between 0 and 10. Suppose that three data sources are available to the controller program: battery charge (which ranges from 0 to 1 inclusive), availability of DC power (true or false), and brightness of the display (which ranges from 0 to 10 inclusive). If the laptop is low on battery, then the laptop automatically changes to a Power Saver Mode where the maximum possible brightness setting is 4. The user may also manually invoke a Sleep Mode, which turns off the screen completely by setting brightness to 0.

2.2 Properties Observed over Training Runs

The first step collects training data by running the program and observing its operation in a variety of scenarios. Suppose that the training runs are selected from among successful runs of test cases; the test harness ensures that the system is performing as desired, so these are good runs to generalize from.

The second step generalizes from the training runs, producing a model of the operation of each mode. Suppose that after running the selected test suite, a program analysis tool infers that the following properties are true in each mode.

Standard Mode	Power Saver Mode	Sleep Mode
$brightness > 0$	$brightness > 0$	$brightness = 0$
$brightness \leq 10$	$brightness \leq 4$	
$battery > 0.2$	$battery > 0.0$	$battery > 0.0$
$battery \leq 1.0$	$battery \leq 0.2$	$battery \leq 1.0$
	$DCPower = \text{false}$	

Notice that the battery life in Standard Mode is always greater than 0.2, which suggests either that Standard Mode can never be chosen when the battery life is less than 0.2, or that the test suite did not adequately cover a case when Standard Mode would have been chosen under low battery life. Upon inspection, one might realize that a laptop could easily reach a state that would contradict all three models, where brightness is high, the remaining battery charge is low, and DC Power exists. We must assume the test suite was incomplete.

2.3 Mode Selection Policy

The third step builds a mode selector from the models, which characterize the program when it is operating as desired. At run time, the mode selector examines its program state and determines which model matches current conditions. When a program state does not perfectly fit into any of the available models, the system must determine which mode is most appropriate by computing some ordinal similarity metric between

Current Program State	Standard Mode	Power Saver Mode	Sleep Mode
brightness: 8	**brightness > 0**	**brightness > 0**	*brightness* = 0
	brightness ≤ 10	*brightness* ≤ 4	
battery: 0.1	*battery* > 0.2	**battery > 0.0**	**battery > 0.0**
	battery ≤ 1.0	**battery ≤ 0.2**	**battery ≤ 1.0**
DCPower: true		*DCPower* = false	
Score	75%	60%	66%

Current Program State	Standard Mode	Power Saver Mode	Sleep Mode
brightness: 8	**brightness > 0**	**brightness > 0**	*brightness* = 0
	brightness ≤ 10	*brightness* ≤ 4	
battery: 0.1	*battery* > 0.2	**battery > 0.0**	**battery > 0.0**
	battery ≤ 1.0	**battery ≤ 0.2**	**battery ≤ 1.0**
DCPower: false		**DCPower = false**	
Score	75%	80%	66%

Figure 2-1: Similarity scores for the three possible modes of the laptop display program, given two different input program states. Properties in boldface are true in the current program state and contribute to the similarity score.

models and the program state. One simple metric is the percentage of properties in the model that currently hold. Figure 2-1 illustrates the use of this metric in two situations. When the brightness is 8, the battery charge is 0.1, and DC power is available, the mode selector chooses standard mode. In a similar situation when DC power is not available, the mode selector chooses power saver mode.

The fourth step is to integrate the mode selector into the target system. As two examples, the new mode selector might replace the old one (possibly after being inspected by a human), or it might be invoked when the old one throws an error or selects a default mode.

After the target system has been given a controller with the capability to invoke the new mode selector, the system can be used just as before. Hopefully, the new controller performs better than the old one, particularly in circumstances that were not anticipated by the designer of the old one.

In this example, the *battery* and *DCPower* variables are inputs, while *brightness* is an internal or output variable. Our technique utilizes both types of variables. Examining the inputs indicates how the original controller handled such a situation, and the internal/output variables indicate whether the mode is operating as expected. For example, if the laptop were to become damaged so that brightness could never be turned above 4, then there is more reason to prefer Power Saver Mode to Standard Mode.

Chapter 3

Program steering

Program steering is a technique for helping a software controller select the most appropriate modality for a software system in a novel situation, even when the software was not written with that situation in mind. Our approach is to develop models based on representative runs of the original program and use the models to create a mode selector that assigns program states to modes. We then augment the original program with a new controller that utilizes the mode selector.

Figure 3-1 diagrams the four-stage program steering process:

1. Collect training runs of the original program in which the program behaved as desired.
2. Use dynamic program analysis or machine learning to build a model that captures properties of each mode during the training runs.
3. Build a mode selector that takes as input a program state and chooses a mode based on similarity to the models.
4. Augment the original program with a new controller that utilizes the new mode selector.

This chapter describes, in turn, policies for each of the steps of the program steering process. It concludes by exploring several potential applications and discussing limits to the applicability of program steering.

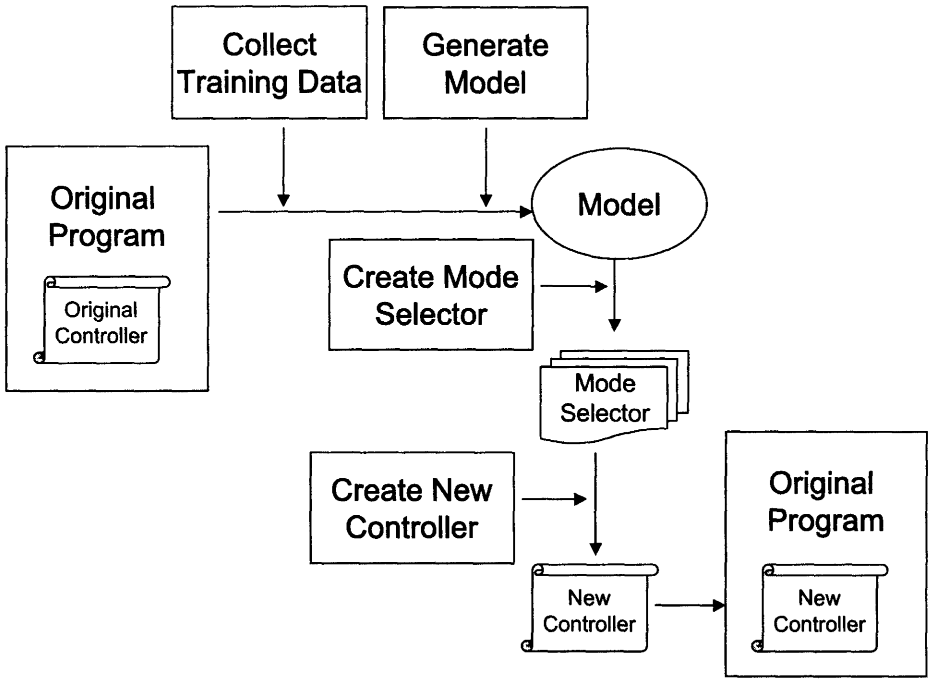

Figure 3-1: The program steering process consists of four steps: executing the original program to produce training data; generalizing from the executions to a model for each mode; creating a new mode selector based on the models; and augmenting the program's controller to utilize the new mode selector.

3.1 Training

The modeling step generalizes from the training data, and the final mode selector bases its decisions on the models. Therefore, better training runs yield more accurate models of proper behavior. If the training runs exercise bugs in the original program, then the resulting models will faithfully capture the erroneous behavior. Therefore, the training runs should exhibit correct behavior. High quality performance over the test suite aids the construction of good models, so we believe it is at least as important as code coverage as a criterion for selecting training runs.

The user might supply canonical inputs that represent the specific situations for which each mode was designed, tested, or optimized. Alternately, the training runs can be collected from executions of a test suite; passing the tests indicates proper behavior, by definition. For non-deterministic systems, the training runs could be selected from a large pool of runs, using only the runs with the best results, as determined by a domain expert of programmed objective function.

There is a danger in evaluating end-to-end system performance: even though the overall system may have performed well on a particular run, certain mode behaviors and transitions may have been sub-optimal, creating a false-positive. The false-negative situation may also occur, when only one poor mode choice out of many excellent choises results in poor end-to-end behavior (false-negatives are less of a concern than false-positives, because they never effect the model; they only slow down the training process). Alternately, the mode transitions may have been perfect, but because of bad luck the overall performance failed to meet the acceptance threshold.

The dynamic analysis tool in the model creation step dictates the required amount of training data. As a minimum, however, the training runs should exercise all of the target system's modes and transitions between modes in order to capture all of the original system's functionality. This can be viewed as a form of code coverage, tuned to multi-mode programs.

Exceptions can be made to the coverage requirement when certain modes or transitions in the original system are undesirable and should not be allowed in the upgraded system. Examples of inappropriate modes are those that exist only for development purposes, such as manual override modes for autonomous systems or modes that output useful debugging information. These modes are never appropriate for a deployed system and should not be part of the new mode selector's valid choices. Furthermore, training runs that include these types of modes are not accurate examples of how a deployed system should run. Because there is no relation between the system's program state and when the developers choose to use the debugging modes, the resulting model is unlikely to provide an accurate description of that mode's behavior.

3.2 Modeling

The modeling step is performed independently on each mode. Training data is grouped according to what mode the system was in at the moment the data was collected, and a separate model is built from each group of training data. The result is one model per mode. Use of multiple models is not a requirement of our technique — we could use a single complicated model that indicates properties of each mode — but creating smaller and simpler models plays to the strengths of machine learners.

Each model represents the behavior of the target system in a particular mode; it abstracts away from details of the specific runs to indicate properties that hold in general. More specialized models could indicate properties not just of a mode, but of a mode when it is switched into from a specific other mode.

The program steering technique does not dictate how models should be represented. Any representation is permitted so long as the models permit evaluation of run-time program states to indicate whether the state satisfies the model, or (preferably) how nearly the state satisfies the model.

The modeling step may be sound or approximate. A sound generalization reports properties that were true of all observed executions. (The soundness is with respect to the learner's input data, not with respect to possible future executions.) An approximate, or statistical, generalization additionally reports properties that were usually true, were true of most observed executions, or were nearly true. For example, a statistical generalization may be able to deal with noisy observations or occasional anomalies. A model may also indicate the incidence or characteristics of deviations from its typical case. These techniques can help in handling base cases, special cases, exceptions, or errors.

3.3 Mode selection

The mode selector compares each model to the current program state and environment (inputs). It selects the mode whose model is most similar to the current state. The mode selector does not explicitly prepare for unanticipated situations, but it can operate in any situation to determine the most appropriate mode. Program steering works because it generalizes the knowledge built into the program by the programmer, possibly eliminating programmer assumptions or noting unrecognized relationships.

Some machine learners have an evaluation function, such as indicating how far a particular execution is from a line (in the model) that divides good from bad runs. Another approach is to execute the modeling step at run time (if it is sufficiently fast) and compare the run-time model directly to the pre-existing per-mode models. Other machine learners produce an easily decomposable model. For instance, if the abstraction for a particular mode is a list of logical formulas, then these can be evaluated, and the similarity score for the model can be the percentage that are true.

A decomposable model permits assigning different weights to different parts. As an example, the properties could be weighted depending on how often they were true during the training runs. As another example, some properties may be more important than others: a non-null constraint may be crucial to avoid a dereferencing error; a stronger property (such as $x = y$) may be more significant than one it subsumes (such as $x \geq y$); weakening a property may be more important as an indicator of change than strengthening one. Weights could even be assigned by a second machine learning step. The first step provides a list of candidate properties, and the second uses genetic algorithms or other machine learning techniques to adjust the weights. Such a step could also find relationships between properties: perhaps when two properties are simultaneously present, they are particularly important.

The new mode selector is likely to differ from the original mode selector in two key ways; one is a way in which the original mode selector is richer, and the other is a way in which the new mode selector is richer. First, the original mode selector was written by a human expert. Humans may use domain knowledge and abstractions that are not available to a general-purpose machine learner, and the original mode selector may also express properties that are beyond the grammar of the model. A machine learner can only express certain properties, and this set is called the learner's *bias*. A bias is positive in that it limits false positives and irrelevant output, increases understandability, and enables efficient processing. A bias is negative in that it inevitably omits certain properties. Our concern is with whether a model enables effective mode selection, not with what the bias is *per se* or whether the model would be effective for other tasks.

Second, the new mode selector may be richer than the original mode selector. For example, a programmer typically tests a limited number of quantities, in order to keep the code short and comprehensible. By contrast, the training runs can collect information about arbitrarily many measurable quantities in the target program, and the automated modeling step can sift through these to find the ones that are most relevant. As a result, the mode selector may test variables that the programmer overlooked but that impact the mode selection decision. Even if the modeling step accesses only the quantities that the programmer tested, it may note correlations that the programmer did not, or strengthen tests that the programmer wrote in too general a fashion [LCKS90].

3.4 Controller augmentation

The new mode selector must be integrated into the program by replacing or modifying the original controller. The controller decides when to invoke the mode selector and how to apply its recommendations. Some programs intersperse the controller and the mode selector, but they are conceptually distinct.

One policy for the controller would be to continuously poll the new mode selector, immediately switching modes when recommended. Such a policy is not necessarily appropriate. As noted above, the original mode selector and the new mode selector each have certain advantages. Whereas the new mode selector may capture implicit properties of the old one, the new one is unlikely to capture every aspect of the old one's behavior. Furthermore, we expect that in anticipated situations the old mode selector probably performs well.

Another policy is to leave the old controller intact but substitute the new mode selector for the old mode selector. Mode changes only occur when the controller has decided that the current mode had completed or was sub-optimal.

A third policy is to retain the old mode selector and override it in specific situations. For example, the new mode selector can be invoked when the original program throws an exception, violates a requirement or assertion, deadlocks, or times out (spends too much time attempting to perform some task or waiting for some event), and also when the old mode selector chooses a passive default mode, has low confidence in its choice, or is unable to make a decision. Alternately, anomaly detection, which aims to indicate when an unexpected event has occurred (but typically does not provide a recommended course of action), can indicate when to use the mode selector. The models themselves provide a kind of anomaly detection.

Finally, a software engineer can use the new mode selector in verifying or fine-tuning the original system, even if the new mode selector is never deployed in the field or otherwise used in practice. For example, the programmer can examine situations in which the two mode selectors disagree (particularly if the new mode selector out-performs the old one) and find ways to augment the original by hand. Disagreements between the mode selectors may also indicate an inadequate test suite, which causes overfitting in the modeling step.

3.5 Applicability of the approach

As noted above, the program steering technique is applicable only to multi-mode software systems, not to all programs, and it selects among existing modes rather than creating new ones. Here we note two additional limitations to the technique's applicability — one regarding the type of modes and the other regarding correctness of the new mode selector. These limitations help indicate when program steering may be appropriate.

The first limitation is that the steering should effect discrete rather than continuous adaptation. Our techniques are best at differentiating among distinct behaviors, and selecting among them based on the differences. For a system whose output or other behavior varies continuously with its input (as is the case for many analog systems), approaches based on techniques such as control theory will likely perform better, particularly since continuous systems tend to be much easier to analyze, model, and predict than discrete ones.

The second limitation is that the change to the mode selector should not affect correctness: it may not violate requirements of the system or cause erroneous behavior. We note three ways to satisfy this constraint. First, if the system is supplied with a specification or with invariants that must be maintained (for instance, a particular algorithm is valid only if a certain parameter is positive), then the controller can check those properties at runtime and reject inappropriate suggestions. If most computation occurs in the modes themselves, such problems may be relatively rare. Second, some modes differ only in their performance (power, time, memory), such as selecting whether to cache or to recompute values, or selecting what sorting algorithm to use. Third, exact answers are not always critical, such as selecting what model of an object to render in a graphics system, or selecting an audio compression algorithm. Put another way, the steering can be treated like a hint — as in profile-directed optimization, which is similar to our technique but operates at a lower level of abstraction.

3.6 Human Steps during Program Steering

This section discusses the human steps required for applying program steering. Fortunately, many of the steps are already automated. Our modeling tool, Daikon, automates the process of instrumenting source code at mode transitions to log data about the program's behavior during training. Daikon also reads and interprets the trace files created during training and generates operational abstractions describing each mode, fully automating the modeling step. Although Daikon alone does not automatically create the new mode selector, there is an option to output the operational abstractions as boolean Java expressions. We have written scripts that merge these boolean expressions into source code, thereby automating the mode selector creation.

When applying the technique to an existing program, the steps that still require human effort are identifying where the modes and mode transitions exist, refactoring the program to be compatible with the modeling tools, selecting good training runs, and determining when to override the original controller and use the new mode selector (augmentation phase). For new systems, the mode identification and refactoring should be unnecessary if the programmers write the code intending to apply program steering upon completion. Specifically, all the modes and mode transitions are already easily identified and extracted into method calls.

3.6.1 Mode Identification

Before the training process begins, the system modes must be identified and the transitions clearly delineated. In the future, mode detection might be automated with the help of programmer annotations or a specified coding style (similar to Javadoc comment tags and Junit method-naming conventions respectively). Such additions would have a minimal impact on the programmers, who presumably already write comments and documentation in the code and use consistent coding and naming conventions. Section 7 also describes related work in program analysis and classification that shows promising results in automatically identifying modes from program execution during test suites.

For now, mode identification is the most human-time consuming process of applying program steering to an existing system because it requires a substantial human effort when the original programs are unable to provide assistance. All but one of the experimental programs in this thesis were written by other programmers, requiring a human to read and understand the code before attempting to identify modes and mode transitions. We have not attempted to study the actual amount of human time each step takes, but each of our subject programs, all less than 2500 lines of code, required at least half a day of human time. We believe that the original programmers, however, would need significantly less time.

The goal of the mode identification phase is to simultaneously accomplish two separate tasks. First, one must identify and isolate modes and mode transitions in order to enable mode selection later in the process. The other task is trying to convert the programs to work well with the chosen modeling tool, which may require additional refactoring.

Mode Finding Heuristics

Programs vary widely in both coding style and functional organization, but we have found some heuristics for quickly identifying modes in unfamiliar programs.

First, one should look for the existence of a centralized mode selector or mode transition point. These are usually large switch statements or if-else chains and might be part of while(true) loop (or other pseudo-infinite loop with a termination condition that can only be met when the program expects to exit). The bodies inside the switch statements are usually modes. Sometimes the bodies are already single method calls and other times they are code blocks that should be extracted into methods.

Next, some programs may have explicit fields that keep track of the mode state. Look for all possible assignments and comparison checks against such a field to find out where modes and mode transitions exist. Even if a program is not structured this way, it might be a good idea to refactor the code in order to avoid an accumulating stack if mode transitions usually occur as a method call from the previous mode. The next Refactoring subsection discusses this problem in more detail.

The main rule of thumb is to declare a block of code a mode if it is reasonable to expect that mode to be useful in an unexpected situation and if the mode demonstrates a strategy or other intelligent behavior rather than performing a low level instruction or computation. In general, a block of code is not a mode if it simply performs some computation to gather information or sensor data. Code is also not a mode if it simply executes some low-level task such as, "move forward two steps" or "turn left". An example of a high level mode is, "Move from point A to point B with the intention of picking up any resources along the way and retreating from any enemies" constitutes a mode. Section 4.4 provides experimental results justifying this preference for higher level versus lower level modes.

Refactoring

We consider the predicate clauses in the `if (...)` or switch statements to be part of the original controller and mode selector, while the body of these blocks are the actual modes. With IDEs such as Eclipse, it is easy to refactor a code block and extract it into a method.

All modes should have the same signature so that it is possible to easily switch between modes. Otherwise, some modes may require additional prerequisite information in order to function properly, limiting the flexibility of the new mode selector. The simplest solution is to refactor all method transitions to require no parameters. That requires a refactoring that lifts all method arguments into global fields and changes the caller of any mode to properly set the global field before the mode change. To facilitate more accurate mode selection, the caller should also assign the global back into null or some uninitialized value immediately after the mode completes, since that would simulate the argument dropping out of scope (the program structure may require that the values are reassigned at a place other than the caller, especially if the program is written to avoid stack accumulation).

3.6.2 Daikon-specific Refactoring

Depending on the modeling technique chosen, the programs may require additional refactorings or modifications in order to complete the training and modeling steps. As mentioned before, we use the Daikon Invariant Detector tool, described earlier in Section 1.3

By design, Daikon does not report properties over local variables, because programmers are typically only interested in the input-output specifications of a function or module rather than the internal details of the implementation. Because the original program may not have been designed for certain mode transitions, some of the information regarding how a mode operates may exist only inside the scope of the mode. It is necessary to lift local variables to globals in order to have all the relevant data for making mode decisions. Even if the modeling technique could compute properties over local variables, some refactoring would be required for the mode selector to handle cases where the local variable was in the scope of some modes and not others.

A complication arises when the controller uses variables for the mode decision where the variables are calculated through several levels of indirection. By default, the Daikon Java Front End only calculates up to two levels of indirection. For example, the trace file will include data about *this.next.prev* but not about *this.next.prev.next*. Setting the level limit higher may find more of the relevant properties, but doing so is likely to create even more problems. First, the number of uninteresting properties found is likely to increase, rendering the models less useful for distinguishing the differences in behavior between the modes. Also, even with an infinite instrumentation level limit, the most interesting properties may involve calculations computed in an uninstrumented part of the code, for example, relationships between library calls that are not part of the system in question. Some programs may require a human or tool to identify calculations that use several levels of indirection and created a new explicit member field to store and update the calculation. In all of our subject programs, we never saw a situation where recalculating and updating these fields caused side-effects that altered the system behavior.

After refactoring, methods implementing the modes may include a significant amount of tail recursion or may call other modes directly without returning to the main infinite loop. Because the methods never terminate, there is a danger of a stack overflow. Also, if the program terminates using System.exit or runs in a Thread that dies when the program exits, the program trace files will not contain data about the exit program points. One workaround to the problem is to refactor so that there is a helper method that contains a `while (true)` loop which calls the actual mode implementing method. To transition to the mode, the previous mode always call the helper instead. This change ensures that the code reaches the exit program point each time through the loop and greatly reduces the size of the accumulating stack if the program previously relied on tail recursion. The helper method is also an area where the code can update added member fields discussed above.

By default, Daikon calculates properties about the program at every entry and exit point for functions and methods. In order to distinguish between each of the modes when analyzing the training run behavior, we use method calls to delineate mode transitions. Then we use a built-in Daikon option to only instrumented the mode transition methods (the names all match a particular regexp), in order to save space with smaller trace files and computation in calculating properties. The mode selector creation is also easier, because the Daikon output file will only contain properties relevant to mode transitions.

Chapter 4

Self-Organizing Fish Experiments

The next two chapters discuss two sets of experiments we performed to evaluate whether program steering is domain-independent and whether program steering can improve adaptability in new situations.

The subject programs in this chapter already adequately perform their intended task, and our goal was to determine whether our new mode selector selector could completely replace the original mode selector and still complete the required tasks.

We experimented on four student solutions for the Self-Organizing Fish Research Assignment from MIT's Spring 2003 Embodied Artificial Intelligence graduate class. These programs were intended to cause fish to form schools while avoiding obstacles. Each submission implemented its own simulated world, fish control programs, and obstacles, all of which were not well specified, leaving a lot of variation between each submission. The purpose of the class assignment was to have students create fish that follow very simple instructions for how to move and interact with other nearby fish, but then demonstrate that when the all the fish follow the simple rules, the entire school of fish as a single entity appears to display a high level of intelligence.

4.1 Program Details

Every submission successfully completed the assignment by simulating fish equipped with limited-range visual sensors to detect nearby fish and obstacles. The fish controlled their velocity and autonomously chose between several modes, such as swimming with the rest of the school, diverting from the school to avoid obstacles, and exploring to find other friends to join. All of the programs were written in Java, and the fish were instances of a Fish objects, one instance per fish.

For basic movement with no obstacles in sight, the fish would compute the velocity of its four closest neighbors and then adjust its own velocity to be the average of those four neighbors. Each program had slightly different rules for what to do when fewer than four neighbors could be found, but in the absence of neighbors, the fish would maintain their current direction.

With obstacles nearby, the fish would alter course and possibly break away from their schools to avoid obstacle collisions. Given the specified basic movement algorithm, it was possible for only a few of the fish in the outer perimeter of the school to actively attempt to avoid the obstacles; the remaining fish, simply by adjusting its velocity to that of its neighbors, would follow the lead of its surrounding fish and keep the school intact.

The overall metric of success was the quality with which the fish form and maintain their schools. The assignment measured quality in the ability to complete several subgoals. The fish must successfully find and join other fish to form the schools, which sometimes required merging two schools moving in differing directions. The fish must also avoid colliding with obstacles and the school should remain intact before and after encountering the obstacle. For example, schools that split into two smaller schools after traveling around an obstacle is less desirable behavior compared to schools that reform back into a single school at the other side of the obstacle or schools that completely divert themselves along one side of the obstacle.

The size of the obstacles relative to the size fish as well as the affects of colliding with the obstacles were different for each program. In one program, a fish colliding with an obstacle would become injured and unable to move for the rest of the simulation. At the other end of the spectrum, fish in another program were able to pass directly through obstacles as if the obstacle didn't exist (the fish never exploited this ability in the student's test cases; we discovered the phenomenon later after altering the fish).

The fish interacted in a simulated toroidal world; fish traveling off one edge of the world will reappear on the opposite edge. The shape allows fish and other objects to be uniform randomly distributed across the entire world. Students were also instructed to simulate a current that pushed fish from West to East, in order to increase the challenge of avoiding obstacles and maintaining large schools.

4.2 Training and Modeling

For each program, a human identified the fish modes and refactored the programs as described in Sections 3.6.1 and 3.6.2.

We ran the fish in the original students' test cases, which covered all the modes of fish execution and demonstrated intelligent fish behavior. The assignment did not specify a common API or other mechanism for interoperability between different student programs, so there was no way to pool together test environments or easily introduce new environments. The tests were not automated, so a human manually ran the tests until observing the required fish behavior for completing the assignment. Although some student simulations were non-deterministic, we did not selectively choose only a subset of the runs for training, as we believed every simulation was successful. All four fish programs received A grades, and the human tester visually verified the correctness of the fish behavior.

4.3 Mode Selection and Augmentation

For the purposes of measuring domain independence, our new controller relied entirely on the new mode selector for mode decisions, completely replacing the old controller and mode selector. During each simulation clock cycle, all the fish invoke their new mode selectors and execute the chosen mode. Each fish sets its velocity for that clock cycle (based on the mode instructions) and then the simulator updates all the fish positions, calculating obstacle collisions if necessary. This controller policy of depending entirely on the new mode selector is probably a poor choice for most systems. We expect most program steering controller implementations to utilize a combination of the old and new mode selectors, where the controller invokes the new mode selector only when there is evidence that the old mode selector might have chosen a suboptimal mode. We implemented a slightly modified version of the uniform weighting mode selection policy from Section 3.3 in order to compensate for the frequent mode selector invocations.

The mode selector computes similarity scores for each model based on the percentage of the model properties satisfied. In order to maintain inertia and prevent frivolous mode transitions, the mode selector includes a modification for deciding which mode is best after calculating the scores. If the highest similarity score x_k for Mode M_k was greater than any others by a large enough margin, then the controller switches to mode M_k. Otherwise, the fish remains in the current mode, even if it has a lower similarity score than other modes. A human determined a hard-coded margin for each fish program by trial and error until the fish behaved as closely to the original as possible. For the two successfully steered programs described below, we found the ideal mode selector margin was approximately 0.1.

4.4 Results

We evaluated the upgrades by comparing the schooling quality of the original fish versus the upgraded fish. We ran separate simulations for each and never tried mixing the original and upgraded fish into the same simulation. We evaluated the frequency of fish colliding with obstacles quantitatively; better avoidance results in fewer collisions. Because a school of fish is not well defined, we only evaluated the ability to form and maintain schools qualitatively: In addition to remaining intact when when avoiding obstacles, better schools are more tightly packed (allowing better maneuvering around obstacles), merge smoothly with other schools (no fish become separated from the new school), and retain all of the fish throughout the simulation (no fish ever break off and swim away from the school).

Our observations show that the success of our new mode selector depends primarily on the original program's test environment and mode representation. For two of the programs, the new controller worked well, observing the environment, making appropriate mode choices, and achieving the fish schooling goals as well as the original controllers did. There was no significant difference in the schooling ability of the new and old fish, and in both cases, the frequency of fish colliding with obstacles was the same (about 1 or 2 fish out of a population of 50 per simulation).

The other two new controllers fared poorly in new environments. Upon investigation, we discovered that the poor performing programs were tested in a single initial configuration and contained hard-coded logic, for instance assuming specific obstacle locations. The original programs only displayed the proper fish intelligence under the exact conditions of the submitted test environments and were not the least bit adaptable. The new fish were no better or worse at adapting to new environments. They chose their modes based on irrelevant properties such as their absolute pixel locations on the screen rather than on data from the visual sensors.

Another reason for the poor performance in the two unsuccessful programs was the representation of the modes. The human identifying the modes could only find low-level modes that required a lot of precise, well-timed decisions for switching modes rather than modes employing an overall strategy. In one case, the only three identifiable modes were Turn-Left-Mode, Turn-Right-Mode, and Move-Straight-Mode. The program steering mode selector was unable to properly choose between three modes, even after trying a 0.0 margin for switching which removed any intentionally inserted inertia that might have hindered the mode transitions timings.

Part of the problem with handling low-level modes is that the program steering mode selector builds in some of its own inertia. Some of the properties in the models are mode outputs, or they are true because the system is operating in that particular mode. For example, there may be a flag that explicitly stores the system's current mode. Properties over the flag will always be true in the current mode but not in any of the other modes, which boosts the similarity score of the current mode. We believe that inertia can be very helpful, because in general the new mode selector should only override the original mode selector when there is a strong possibility that a different mode is more optimal.

Another problem with low-level modes is that the models created are indistinguishable; the general data about the surrounding environment is unlikely to have any correlation with mode selection. Because Turn-Left mode, Move-Straight mode, and Turn-Right mode are general navigational operations, they can occur under any conditions, so all the properties discovered during the dynamic analysis will simply be universally true properties. As a result, all three models will be nearly if not entirely identical.

We conclude from this experiment that the initial program needs to be of at least decent quality. The training data should be general enough for the modeling tools to extract relevant properties for making intelligent decisions and avoid over-fitting to hard-coded environments. Also, the technique works best at helping multi-mode systems containing high-level strategies such as choosing between rock avoidance and fish schooling, rather than low-level modes in which information about the surrounding environment and internal state are not helpful for determining the optimal mode.

Chapter 5

Droid Wars Experiments

Our second set of experiments applied program steering to robot control programs built as part of a month-long Droid Wars competition known as MIT 6.370 (`http://web.mit.edu/6.370/www/`). The competition was run during MIT's January 2003 Independent Activities Period; 27 teams — about 80 undergraduate and graduate students — competed for $1400 in prizes and bragging rights until the next year.

Droid Wars is a real-time strategy game in which teams of virtual robots compete to build a base at a specified goal location on a game map. The team that first builds a base at the location wins; failing that, the winner is the team with a base closer to the goal location when time runs out. Each team consists of four varieties of robot; all robots have the same abilities, but to differing degrees, so different robot types are best suited for communication, scouting, sentry duty, or transport. Robot abilities include sensing nearby terrain and robots, sending and receiving radio messages, carrying and unloading raw materials and other robots, attacking other robots, repairing damage, constructing new robots, traveling, and rebooting in order to load a different control program. The robots are simulated by a game driver that permits teams of robots to compete with one another in a virtual world.

The robot hardware is fixed, but players supply the software that controls the robots. The programs were written in a subset of Java that lacks threads, native methods, and certain other features. Unlike some other real-time strategy games, human intervention is not permitted during play: the robots are controlled entirely

Program	Total lines	NCNB lines	Modes	Properties
Team04	920	658	9	56
Team10	2055	1275	5	225
Team17	1130	846	11	11
Team20	1876	1255	11	26
Team26	2402	1850	8	14

Figure 5-1: Statistics about the Droid Wars subject programs. The Total lines column gives the number of lines of code in the original program. The NCNB Lines column gives the number of non-comment, non-blank lines. The Modes column gives the number of distinct modes for robots in that team. The Properties column gives the average number of properties considered by our mode selector for each mode.

by the software written by the contestants. Furthermore, there is no single omniscient control program: each robot's control program knows only what it can sense about the world or learn from other robots on its team.

Many participants wrote software with different modes to deal with different robot types, tasks, and terrain. For instance, a particular robot might have different modes for searching for raw materials, collecting raw materials, scouting for enemies, attacking enemies, relaying messages, and other activities. The organizers encouraged participants to use the reboot feature (which was also invoked at the beginning of the game and whenever a robot was created) to switch from one control program to another. Another strategy was to write a single large program with different classes or methods that handled different situations. Some of the programs had no clearly identifiable modes, or always used the same code but behaved differently depending on values of local variables.

We augmented the programs of teams 04, 10, 17, 20, and 26 with program steering. Figure 5-1 gives some details about these programs. These teams place 7th, 20th, 14th, 22nd, and 1st, respectively, among the 27 teams in a round-robin tournament using the actual contest map and conditions. We chose these teams arbitrarily among those for which we could identify modes in their source code, which made both training a new mode selector and applying it possible.

5.1 Applying program steering

As described in Chapter 3, applying program steering to the robot control programs consisted of four steps: training, modeling, creating a mode selector, and augmenting the controller. Some (one-time) manual work was required for the training and controller augmentation steps, primarily in identifying the modes and applying the refactorings from Section 3.6. The lack of documentation and the possibility of subtle interactions (each robot ran in its own virtual machine, so global variables were very frequently used) forced us to take special care not to affect behavior. The mode identification and refactoring steps required about 8 hours of human time per team. We hypothesize that the original authors of the code would need significantly less time.

5.1.1 Training

We collected training data by running approximately 30 matches between the instrumented subject team against a variety of opponents. Training was performed only once, in the original environment; training did not take account of any of the environmental changes of Section 5.2, which were used to evaluate the new mode selectors. We did not attempt to achieve complete code coverage (nor did we measure code coverage), but we did ensure that each mode was exercised.

We only retained training data from the matches in which the team appeared to perform properly, according to a human observer. The purpose of this was to train on good executions of the program; we did not wish to capture poor behavior or bugs, though it is possible that some non-optimal choices were made, or some bugs exposed, during those runs. The human observer did not examine every action in detail, but simply watched the match in progress, which takes well under 5 minutes. An alternative would have been to train on matches that the team won, or matches where it did better than expected. Another good alternative is to train on a test suite, where the program presumably operates as desired; however, none of the robot programs came with a test suite.

5.1.2 Modeling

In our experiments, the modeling step is carried out completely automatically, and independently for each mode, producing one model per mode. The models included properties over both environmental inputs and internal state variables, as described in Chapter 2.

We made some small program modifications before running the modeling step. A few quantities that were frequently accessed by the robot control programs were available only through a sequence of several method calls. This placed them outside the so-called instrumentation scope of our tools: they would not have been among the quantities generalized over by the Daikon tool, as discussed in Section 3.6.2. Therefore, we placed these quantities — the number of allies nearby, the amount of ore carried, and whether the robot was at the base — in variables to make them accessible to the tools.

5.1.3 Mode selection

Given operational abstractions produced by the modeling step, our tools automatically created a mode selector that indicates which of the mode-specific operational abstractions is most similar to the current situation.

Our tools implement a simple policy for selecting a mode: each property in each mode's operational abstraction is evaluated, and the mode with the largest percentage of satisfied properties is selected. This strategy is illustrated in Figure 2-1.

Figure 5-2 shows code for a mode selector that implements our policy of choosing the mode with the highest percentage of matching properties.

5.1.4 Controller augmentation

Finally, we inserted the automatically-generated mode selector in the original program. Determining the appropriate places to insert the selector required some manual effort. We did not completely replace the old controller and mode selector as we did in the fish experiments (see Section 4.3). Instead, we augmented the original controller

```
int selectMode() {
  int bestMode = 0;
  double bestModeScore = 0;

  // Compute score for mode 1
  int mode1Match = 0;
  int mode1Total = 4;
  if (brightness > 0)   mode1Match++;
  if (brightness <= 10) mode1Match++;
  if (battery > 0.2)    mode1Match++;
  if (battery <= 1.0)   mode1Match++;
  double mode1Score =
      (double) mode1Match / mode1Total;
  if (mode1Score > bestModeScore) {
    bestModeScore = mode1Score;
    bestMode = 1;
  }

  // Compute score for other modes
  ...

  // Return the mode with the highest score
  return bestMode;
}
```

Figure 5-2: Automatically generated mode selector for the laptop display controller of Chapter 2. The given section of the mode selector evaluates the appropriateness of the display's Standard Mode.

to use the new mode selector in several strategic situations. The new mode selector was invoked when the program threw an uncaught exception (which would ordinarily cause a crash and possibly a reboot), when the program got caught in a loop (that is, when a timeout occurred while waiting for an event, or when the program executed the same actions repeatedly without effect), and additionally at moments chosen at random (if the same mode was chosen, execution was not interrupted, which is a better approach than forcing the mode to be exited and then re-entered). Identifying these locations and inserting the proper calls required human effort.

5.2 Environmental changes

We wished to ascertain whether program steering improved the adaptability of the robot control programs. In particular, we wished to determine whether robot controllers augmented with our program steering mechanism outperformed the original mode selectors, when the robots were exposed to different conditions than those for which they had been originally designed and tested. (Program steering did not affect behavior in the original environment; the augmented robots performed just as well as the unaugmented ones.)

We considered the following six environmental changes, listed in order from least to most disruptive (causing difficulties for more teams):

1. New maps. The original tournament was run on a single map that was not provided to participants ahead of time, but was generated by a program that is part of the original Droid Wars implementation. We ran that program to create new maps (that obeyed all contest rules) and ran matches on those maps. This change simulates a battle at a different location than where the armies were trained. We augmented all the environmental changes listed below with this one, to improve our evaluation: running on the single competition map would have resulted in near-deterministic outcomes for many of the teams.
2. Increased resources. The amount of raw material (which can be used to repair damage and to build new robots) is tripled. This environmental change simulates a battle that moves from known terrain into a new territory with different characteristics.
3. Radio jamming. Each robot in radio range of a given transmission had only a 50% chance of correctly receiving the message. Other messages were not received, and all delivered messages contained no errors. (Most of the programs simply ignored corrupted messages.) This environmental change simulates reduction of radio connectivity due to jamming, intervening terrain, solar flare activity, or other causes.
4. Radio spoofing. Periodically, the enemy performs a replay attack, and so at an

arbitrary moment a robot receives a duplicate of a message sent by its team, either earlier in that battle or in a different battle. This environmental change simulates an enemy attack on communications infrastructure. The spoofing reduces in likelihood as the match progresses, simulating discovery of the spoofing attack or a change of encryption keys.

5. Deceptive GPS. Occasionally, a robot trying to calculate its own position or navigate to another location receives inaccurate data. This environmental change simulates unreliable GPS data due to harsh conditions or enemy interference.
6. Hardware failures. On average once every 1000 time units, the robot suffers a CPU error and the reboot mechanism is invoked, without loss of internal state stored in data structures (which we assume to be held in non-volatile memory). All robots already support rebooting because it is needed during initialization and is often used for switching among modes. A match lasts at most 5000 time units, if neither team has achieved the objective (but most matches end in less than half that time). In a single time unit, a robot can perform computation and move, and can additionally attack, repair damage, mine resources, load, unload, reboot, or perform other activities. This environmental change simulates an adverse environment, whether because of overheating, cosmic rays, unusually heavy wear and tear, enemy action, or any other circumstance that might make the hardware less reliable.

5.3 Evaluation

We evaluated each new program by running a tournament with the original programs (in the new environment), then running a new tournament in which we replaced the original program by the augmented version (the other tournament participants were identical). We compared the rank of the original program with the rank of the augmented program in their respective tournaments.

We always compare the ranks of the unmodified and modified robots in a tournament run in the same environment. The rank of the unmodified robot (that is, without the program steering augmentation) in the new environment is not necessarily the same as that of the robot in the original environment, because different robots are affected in different ways by environmental changes. If the original mode selector's choices are not appropriate for the new environment (or the original controller itself fails), then creation of a new mode selector may be able to improve the situation. If the modes themselves are not appropriate for the new environment (for instance, their algorithms no longer achieve their goals), then no amount of improvement to the mode selector can restore the system to good performance.

The Droid Wars competition used a double-elimination tournament. Our evaluation uses a round-robin tournament. This requires additional time to run the much larger number of matches, but it permits more accurate ranking. In particular, we played each team against each other team approximately 10 times, and determined which team of the pair won the most games. (We played multiple matches per pair of teams because each match used a randomly-generated map, as described in Section 5.2.) We used the summary results (one per pair of teams) to rank all the teams; we ranked teams according to the number of other teams that the team defeated.

Figure 5-3 on page 52 shows the results of each set of tournaments, giving the original rank of each team and the rank after the program steering upgrades. The Original column gives the tournament rank of the original team (smaller is better), and the Upgraded column gives the rank (in a separate tournament) of the team when upgraded with program steering. The Change column shows the improvement in ranking (the difference between the Original and Upgraded columns). The Rand column gives the change in ranking (from the Original column) when using the new controllers with a random mode selector.

The positive results reported in the Change column of Figure 5-3 might be attributed to two different sources. They might be a result of a high-quality mode selector constructed by our technique, or they might be a result of a high-quality controller, which chooses to use the new mode selector in exactly the right situations. If the latter explanation is true, then just getting the robot program unstuck or out of a bad mode might be the major benefit, and the mode selector's choice would be of secondary importance. To investigate this hypothesis, we evaluated another variant of the target programs that was identical to the upgraded versions, except that instead of using our mode selector, we used a random mode selector. As indicated in the Change and Rand columns of Figure 5-3, our mode selector substantially outperformed the random mode selector.

5.4 Discussion

Overall, program steering aided Team26 least. There were two reasons for this. First, Team26 already performs very well (it placed first in the actual tournament), so there is less opportunity for improvement. It also uses a very simple control program that leaves little room for modification or enhancement. Team26's robots do not rely on centralized knowledge or decision-making. The robots initially sweep across the entire map to forage for raw materials and replicate. Then, the robots self-organize by meeting at a designated location at a hard-coded time; when enough have arrived, they mass to attack. No communication is required, because a default location is hard-coded into the robot's program. However, if a robot discovers a more strategic meeting point (such as the location of the enemy base or an important enemy convoy), then it notifies the base, which relays the message to the rest of the team.

Team04, Team10, and Team20 use a different architecture. They implement centralized intelligence: the base collects information, decides strategy, and issues instructions to the other robots. The non-base robot control programs are relatively simple, because they are designed primarily to follow the base's directives. They are sometimes unable to react appropriately if an unexpected situation arises while the robot is out of contact with the base. Program steering has essentially distilled simple autonomous programs for them, automatically producing a version consistent with the base's control program.

Team17 uses the time elapsed from the beginning of the battle to determine when to switch modes and how to assign initial modes. The mode-change times are carefully crafted and over-fit to the tournament rules which specify 5000 clock ticks per match.

New Maps				
Program	Original	Upgraded	Change	Rand
Team04	7	7	0	−2
Team10	20	20	0	0
Team17	14	14	0	−3
Team20	23	18	+5	−3
Team26	2	2	0	−1

Increased Resources				
Program	Original	Upgraded	Change	Rand
Team04	13	10	+3	0
Team10	17	16	+1	+1
Team17	18	16	+2	0
Team20	16	11	+5	−4
Team26	2	3	−1	−1

Radio Jamming				
Program	Original	Upgraded	Change	Rand
Team04	12	7	+5	+ 2
Team10	22	19	+3	+ 3
Team17	16	12	+4	− 1
Team20	18	11	+7	0
Team26	6	6	0	−10

Radio Spoofing				
Program	Original	Upgraded	Change	Rand
Team04	19	12	+7	+7
Team10	23	23	0	−1
Team17	11	9	+2	0
Team20	16	10	+6	0
Team26	26	26	0	−1

Deceptive GPS				
Program	Original	Upgraded	Change	Rand
Team04	12	9	+ 3	−5
Team10	23	8	+15	+5
Team17	20	9	+11	0
Team20	22	7	+15	+1
Team26	16	13	+ 3	−4

Hardware Failures				
Program	Original	Upgraded	Change	Rand
Team04	11	5	+ 6	+2
Team10	20	16	+ 4	−1
Team17	15	9	+ 6	−5
Team20	21	6	+15	−2
Team26	17	13	+ 4	−5

Overall Averages				
Program	Original	Upgraded	Change	Rand
Team04	12.3	8.3	+4.0	+0.7
Team10	20.8	17.0	+3.8	+1.2
Team17	15.7	11.5	+4.2	−1.5
Team20	19.3	10.5	+8.8	−1.3
Team26	11.5	10.5	+1.0	−3.7

Figure 5-3: Difference in performance between the original programs and versions upgraded with program steering.

The original developers created a program that only performed well when specific assumptions about the environment were true. The program steering mode selector was better equipped for adapting to changes by extracting and generalizing the hard-coded knowledge from the training examples.

5.4.1 Details of Behavior in the New Environments

We now briefly discuss results for each of the new environments listed in Figure 5-3.

New maps. The new maps environment differs only marginally from the original one, yet program steering substantially helps Team20. While investigating this effect, we discovered a programming error that can cause its robots to enter an infinite loop. When a robot discovers two caches of raw materials that are very close to one another, the robot navigates to one of them but attempts to pick up the other one. It does not check whether the attempt to pick up the raw materials was successful, but immediately returns to the base, where the team maintains a centralized stockpile. At the base, the robot unloads its cargo (which is nothing, in this case), and then returns to the site of the raw materials (since it knows that some raw materials remain there), and again unsuccessfully attempts to pick up some of the raw materials. The new controller eventually times out of the infinite loop and invokes the new mode selector, which chooses a different task for the robot, preventing it from continuing the fruitless repetition. The other upgraded teams were largely unaffected by the new maps. The new mode selectors usually agreed with the original mode selectors when invoked.

Increased resources. The original Team04 and Team20 control programs assign modes to robots based on assumptions about the size of the army, which is correlated with resources available for constructing units. The control programs are sub-optimal when the assumptions are incorrect. For example, too many robots are assigned to search for resources rather than to attack or defend. The Team17 upgrade provides a slight improvement because the hard-coded times in the original program are worse, given the faster battle progression.

Radio jamming. Team26's strategy does not depend on radio messages, so radio jamming had little effect on the team's performance. The units use the default rendezvous point to launch an effective attack. The teams with centralized intelligence suffered significantly when the base could not reliably issue commands to its less intelligent allies. With program steering, the robots autonomously chose the mode consistent with what the base instructed them to do during similar training examples. For this and the remaining new environments, the the hard-coded original Team17 strategy copes poorly, while the adaptive mode selector continues to perform well.

Radio spoofing. The radio spoofing environment had a strong negative impact on Team26, which program steering was unable to reverse. The spoofed messages were replay attacks collected from previous battles, including ones on different maps. The robots did not authenticate the messages, so different robots received messages specifying different rendezvous points. The team's strategy hinged on gathering a large army, which was disrupted by the spoofing. The new mode selector indicated that something was wrong, because the *Attack Mode* usually involved many nearby allies. Unfortunately, the hard-coded meeting time was set late in the match, and by this time the robots were scattered. There was not enough time to recover from the mistake, so program steering did not improve Team26's overall performance in this environment.

Deceptive GPS. Program steering helped every team, including Team26, in the Deceptive GPS environment. Robots using the original Team26 control program did not all arrive at the rendezvous point because some navigation systems were inaccurate, resulting in a weaker army. The misled robots with upgraded controllers noticed a lack of nearby allies and had time to travel to the (relatively nearby) intended destination. The teams with centralized intelligence had even more serious problems with the deceptive GPS environment. After completing a task the robots frequently returned back to the base to await the next instructions. The deceptive GPS mislead robots into traveling to a different location outside of the radio transmission range of the base, where they would wait for messages with no hopes of ever receiving one. The upgraded controllers would trigger a timeout and the mode selector would notice that the robot should be in the *Go Home Mode.*

Hardware Failures. Many teams in the tournament, including the five we upgraded, only expected program reboots while the base was in radio range or when a nearby ally issued a specific command. The hardware failures environment caused reboots to occur in other situations, sometimes causing the hardware to assume a passive state or some other default. Team20 drastically improved with the upgrades because the robots could frequently infer the correct mode and complete the task at hand. Team26 saw significant but less substantial improvement because the attack phase of its strategy did not take effect until late in the game, requiring the robots to withstand several hardware failures. Team10 did not improve as much because many of its modes required certain preconditions to be met or would fail during the course of executing the mode. For example, there is no reason to enter *Go Home Mode* without knowing the location of home. Those preconditions were discovered in our modeling step, but the selector sometimes chose modes without the preconditions satisfied. Our mode selection weighted each property equally and Team10 models contained many properties (over 200 per mode). Given some of the refined mode selection techniques discussed in Section 3.3, the preconditions should carry more weight.

5.4.2 New Behavior Using Original Modes

We observed another way in which program steering affected the operation of the programs. Some of the programs followed a fixed sequence of modes in a fixed order. As a simple example, after picking up ore, the robot might always return to base — even if the robot had the capacity to pick up more ore along the way, or even if it encountered a vulnerable enemy robot. The new mode selector sometimes executed one of the modes without executing the other one; even though both were always executed together in the training runs, the modeling step discovered additional connections. This gave the robot with program steering a larger range of behaviors than the original robot, and some of those behaviors appear to be valuable ones, even though the original designers had not anticipated the modes interacting in that way.

Our technique can also create a mode selector that chooses mode transitions that could never be chosen before. For example, suppose that the original mode selector for the laptop display example of Section 2.2 was as follows:

```
if (battery <= 0.2 && DCPower == false)
  return PowerSaverMode;
else
  return StandardMode;
```

This selector never chooses sleep mode — perhaps that mode is triggered manually by the user. The new mode selector (Figures 2-1 and 5-2) not only tests a variable (brightness) that original did not, but it can also choose a mode that existed in the system but was not previously accessible. For instance, the display can sleep when turned to brightness 0, which might save even more power than power saver mode would.

Chapter 6

Reinforcement Learning Approach

The results from our experiments are very promising, but we believe that other policies for modeling and mode selection can provide even more improvement, or at least alternatives when the other policies are ineffective. This chapter describes a different set of policies for the modeling and mode selection steps of program steering based on reinforcement learning. Reinforcement learning is an unsupervised machine learning technique that attempts to maximize the total payoff awarded by some payoff function through trial and error, constantly gathering knowledge about the payoff function and storing the data in what we call a genome.

The models in this chapter are sets of properties describing program states and the mode selection is a reinforcement learning solution to the classic N-armed bandit problem, described in Section 6.1. Section 6.2 explains how the N-armed bandit problem relates to mode selection and program steering and Section 6.3 details the steps for implementing a reinforcement learning mode selector. Section 6.4 proposes a slight variation on the mode selection and modeling by leveraging our hypothesis that the optimal model for similar program states are likely to be the same. Section 6.5 proposes a different technique for creating a payoff function and assigning payoffs that has proved successful in other problem domains. We implemented each of the ideas in this chapter for the Droid Wars framework except for the delayed payoffs described in Section 6.5. We discuss our preliminary experiences implementing and utilizing the technique in section 6.6, but have not yet completed an extensive evaluation and comparison with the previous program steering implementation.

6.1 N-armed Bandit Problem

A one-armed bandit is a slot machine. Therefore, an N-armed bandit is a slot machine with N levers to pull. A gambler must decide which lever to pull on any given play and attempts to maximize his or her long-run expected payoff by playing the machine many times and learning the effects of each lever. The only information available is the model of the state of the machine: specifically, the positions of all of the reels from the previous play.

The gambler notices that each lever has its own unique and fairly consistent effect on the reels. The challenge is learning which lever will give the best expected payoff given some current state (based on the model). Adding to the difficulty is that the payoffs are randomized based on some initially unknown probability distribution, so even if the gambler runs into the same modeled state twice, he or she can not be assured that pulling the same lever will provide the same payoff.

One final problem is that the learning must occur online, which means that any strategy to maximize payoff must take into account the cost of learning from mistakes. It would not be wise to simply try every lever for every state multiple times just to learn as much as possible about the payoff function. Instead, a clever solution will strike a balance between exploiting past knowledge as well as exploring new choices. If the gambler encounters state A, pulls lever 1, and receives a payoff of say 20, then what should he or she do when encountering state A again? Does the gambler try lever 2 instead, or was 20 good enough? What if this time lever 1 gives a payoff of 10, or -5, or 100?

One solution to this problem is a form of reinforcement learning, which specifies a class of learning algorithms to solve the problem. One big difference between reinforcement learning and several other types of machine learning is that it uses unsupervised learning. There is no oracle that says, "You chose lever 1, but actually lever 3 was best." Instead, the only feedback is a cardinal evaluation, "You chose lever 1, that scores a 20 this time."

6.2 Program Steering and the N-armed Bandit

The program steering mode selection problem shares some traits with the N-armed bandit problem. The models are descriptions of program states, which we call model states. A model contains a set of properties and whether or not the each property is true in that program state. All the models within the same program describe the same set of properties. The levers are different modes we can choose based on the current model state.

The models should be as precise as possible; ideally, there would be a one-to-one mapping of program states to model states. However, when attempting to distill all the possible nuances of a complex system into true/false properties, there will clearly be some loss of information. Fortunately, the many-to-one mapping from program states to model states is not a problem as long as all the program states mapped to a model state should be treated the same with respect to mode selection. Otherwise, the models are too coarse and are missing important information required for making mode decisions.

The challenge in fitting a standard reinforcement learning algorithm to the mode selection problem is our inability to assign payoffs to each mode choice. We simulate the payoffs by running the training alongside an actual run of the original program in the original environment. Every time the original program makes a mode decision, the training program also selects a mode, obtaining a 0.00 payoff if it disagrees with the mode selector and a 1.00 payoff if it agrees. The training builds upon the previous genome and carries over the genome to the next simulation. After many simulations, the mode selection algorithm should reach a fixed point, where further training has no affect and almost all mode decisions agree with the original controller (we discuss the few exceptions in Section 6.6).

6.3 Implementing a Solution to N-armed Bandit Mode Selection

This section describes the procedure for implementing a reinforcement learning version of program steering, which alters the modeling and mode selection steps described in Sections 3.2 and 3.3. The implementation still uses operational abstractions inferred over successful example runs, but instead of creating models that describe the execution of modes, the models describe program states. The modeling step includes a new training process similar to the training for an N-armed bandit problem. The mode selection step simply exploits the knowledge learned during the modeling.

The first step in the procedure is to upgrade the target program through the uniform weighted program steering described in Chapter 3.3, which creates a refactored version of the program code so that mode transitions are easy to identify (call it ORIG). There is also the upgraded version of the code that contains a mode selector based on uniform weighted properties discovered through dynamic analysis during training (call it UNIFORM).

6.3.1 Genome Data Structure

The genome is the accumulated knowledge store of how a particular payoff function assigns payoffs. Our genome is a mapping of model states to vectors of payoff histories. A model state represents an equivalence class of program states that satisfy the same properties found during the training phase (and hopefully should be treated the same in mode selection).

Figure 6.3.1 shows a conceptual diagram of a sample genome. Each state is represented as a bit string, for example: Each bit index represents one of the properties found during the original program steering modeling step from Section 3.2. A "1" means the corresponding property is true in that program state. For each model state, there is an associated knowledge entry, the lists of expected payoffs for each mode. The genome also stores the number of times the selector chose each mode for

```
001001010101   (State A)  --->  <Knowledge Entry A>
001001010100   (State B)  --->  <Knowledge Entry B>
001001010111   (State C)  --->  <Knowledge Entry C>
001001010000   (State D)  --->  <Knowledge Entry D>
101010100010   (State E)  --->  <Knowledge Entry E>
.....

<Knowledge Entry A> = (Mode 1, 0.05), (Mode 2, 0.13), (Mode 3, 0.86)
<Knowledge Entry B> = (Mode 1, 0.08), (Mode 2, 0.11), (Mode 3, 0.97)
...
```

Figure 6-1: A conceptual diagram of a sample genome. The number of occurrences of each mode for each knowledge entry are not shown.

each state in order to properly compute the running average. Each payoff represents the probability that the that mode selector is correct (to the best of this genome owner's knowledge).

Associated with each model state is a knowledge entry, where a knowledge entry is a list of expected payoffs for selecting each mode given that model state. Our current implementation initializes each expected payoff to $\frac{1}{N}$, where N is the number of modes, although in the long run the initialization values should have no effect, provided that all values are positive and each mode begins with the same value. With each update, the values will eventually converge to the probability that the mode is the correct choice given the model state.

6.3.2 Training the Mode Selector

Our reinforcement learning program steering implementation creates two new programs. One program trains the genome using a reinforcement learning solution to the N-armed bandit problem. Call this program REFLEARN. The second program, which we will call REFDEPLOY, uses the trained solution for mode selection.

REFLEARN behaves exactly like ORIG during operation, but also trains its genome at the same time. Initially, the genome contains zero knowledge. With each execution, REFLEARN trains its genome so that the behavior of the new mode selector will mimic the behavior of ORIG, ultimately generalizing and extrapolating the experiences to new environments. Whenever REFLEARN switches modes based on the ORIG, we perform an iteration of training the N-armed bandit problem by asking our mode selector what mode it would take in the current situation.

Our current implementation only allows payoff scores of 1.00 and 0.00 for agreeing or disagreeing with the original mode selector. If a more fine-grained spectrum of payoffs were available, then the training should be more accurate and occur more quickly.

One disadvantage of the reinforcement learning scheme is the additional dependence on the original program's correctness. The mode selection portion of the upgraded fish and Droid Wars programs only used the original program to extract models of behavior from training. The reinforcement learning approach proposed relies much more heavily on the original program, because the genome training evolves to emulate the original program's behavior. The original program and any bugs or unintelligent behavior is more likely to carry over to the resulting program. Also, there may be a greater danger of over-fitting to the specific training environment.

Our learning algorithm uses a standard solution to the N-armed bandit problem. A training step consists of computing the model state from our program state and then loading our prior recorded knowledge entry for that model state. Recall that the knowledge entry consists of the expected payoffs of selecting each mode given that model state. The payoff for agreeing with the original mode selector is 1.00, and the payoff for disagreeing is 0.00. The new expected payoff becomes:

$$X_{n+1} = \frac{n}{n+1} * X_n + \frac{1}{n+1} p$$

where X_n is the previous expected payoff, X_{n+1} is the new expected payoff, and p is the payoff from our last move. The formula is a running average of the payoffs observed so far given the same exact model state and mode selection. Several variations on the learning algorithms exist, such as placing more weight on the most recent data. Because our approach separates the learning and deployment into two separate stages, our choice of learning algorithm may need to differ from traditional reinforcement learning algorithms, which take into account the need for both exploration and exploitation simultaneously. Section 6.5 describes a way to combine the learning and deployment stage for program steering, which would share more similarities with the standard N-armed bandit problem.

6.3.3 Utilizing the Reflearn Mode Selector

REFDEPLOY uses the knowledge from REFLEARN to adapt to new situations based on previous experiences. Recall that UNIFORM contains the original uniform weighting mode selector based on models describing each mode. UNIFORM also uses an augmented controller as described in Section 3.4, which calls its new mode selector. REFDPLOY differs from UNIFORM in three ways. First, REFDEPLOY loads the genome created while training REFLEARN. Second, REFDEPLOY must contain a new procedure that can compute its model state at runtime, allowing the program to fetch knowledge from its genome. Finally, REFDEPLOY uses a different mode selector from UNIFORM, described below.

The REFDEPOY mode selector computes its model state, the bit string describing which of the properties are true. Next, the mode selector fetches the corresponding knowledge entry for that state from its genome, which contains the expected payoffs associated with each mode selection. Lastly, the mode selector chooses a mode based on the knowledge entry and the selection policy. The selection policy for a standard solution to the N-armed bandit problem normalizes all the expected payoffs in that knowledge entry so that the values sum to 1.00, then selects a mode so that the probability of selecting each mode is equal to its normalized value. Given that this implementation of REFDEPLOY only exploits the knowledge from the genome and does not learn new knowledge, then different selection policies may be more appropriate. For example, the mode selector could simply select the mode with the highest expected payoff.

6.4 K-Nearest Neighbor Selection

A variation for reinforcement learning selects from the genome the k-nearest model states to the current program state and combines each of the knowledge entries for mode selection. The benefits of this modification apply primarily during deployment, although the same modification can also be made during training; our initial experiments were inconclusive in determining which is more advantageous.

Combining the knowledge entries from similar model states provides two advantages. First, if the system encounters a model state never seen during training (because the system encountered an unexpected scenario), then calculating the nearest neighbors can provide knowledge entries from similar situations. Second, if two different program states map to the same model state, then the neighboring state strings might give a better idea as to the more appropriate mode to choose (although this line of reasoning assumes that similar model states imply similar program states).

In our experiments, we have used Hamming distance for calculating the nearest neighbors between other state strings. The Hamming distance between two equal length bit strings is the number of differing indices. For example, State A and State B above have a Hamming distance of 1, because only the right-most bit differs. State A and State E have a Hamming distance of 8. Other distance functions may take into account the actual properties and weight them differently.

6.5 Delayed Payoffs

The reinforcement learning algorithms train to achieve the highest possible expected payoffs, so it is essential to have a payoff function that appropriately correlates high scores with successful behavior. To achieve more accurate payoff scores, one might consider delaying payoff assignments for mode selections until after more information exists about the quality of the choice. An an example of delaying in the Droid Wars programs, the payoffs would not be assigned immediately after each mode selection, but after each match. A similar reward delaying variation to reinforcement learning, temporal difference learning [Tes95], successfully trained an expert Backgammon machine player.

The delaying technique for training a mode selector would enable payoff functions based on how well the system performs rather than on how well the system emulates the original mode selector. Using the Droid Wars example, a simple payoff function could be 1 if the team wins the match, 0 if the team ties, and -1 if the team loses. The score can also take into account more fine-grained details regarding the victory, using heuristics such as the amount of time that passed before the game ended and the number of droids created and destroyed. The more desirable the victory, the higher the payoff. Notice that unlike the simple case of only two possible payoffs as described in the previous section, the cardinal value of the payoffs now have significance, not just the ordinal value. The N-armed bandit solutions attempt to maximize expected value, so payoff functions should be chosen with care.

Another advantage of delayed payoffs is that the learning can occur post-deployment. Because the payoff function is part of the program, the program can evaluate for itself how well it is doing, even in new environments. An interesting set of experiments would measure not only how well programs perform immediately after deployment in new environments, but also how quickly and how well the programs adapt to the new environment in the long term; that is, will the program change its behavior through feedback from lower payoffs in the new environment? If so, how many runs are required for the program to reach its new optimal mode selection? Performing well on both metrics, the pre-deployment and post-deployment learning, requires a delicate balance between the exploration and exploitation portions of reinforcement learning. An algorithm too heavily biased toward exploration may adapt quickly to new environments, but may make poor exploratory mode choices in a stable environment. An algorithm that has lots of exploitation but little exploration may be too slow to adapt to changes.

It is also possible to incorporate a combination of both the immediate reinforcement learning payoffs based on the original mode selector and the delayed payoffs based on the quality of the match performance. A script could examine the difference between the payoff knowledge before and after the training run, and then magnify all of those differences based on the end-to-end performance.

For example, suppose for one particular expected payoff, the value of X changed from 0.5 to 0.6, a difference of 0.1. If the team lost the match, the update script will modify the new changes to move in the opposite direction, so the values actually change to 0.4. On the other hand, if the team won very quickly, then the update script will double or triple the change amount, say 0.7 or 0.8 instead of 0.6.

6.6 Preliminary Results

We applied a reinforcement learning version of program steering in two of the Droid Wars teams, incorporating the two-stage training and deployment as well as nearest neighbor lookups for knowledge entries. We have not yet implemented a delayed payoff scheme.

The training in both cases reached a fixed point after approximately 3000 matches, meaning that additional runs did not affect the mode selector anymore. This training takes about 100 hours of machine time on a 3.0 GHz Pentium 4. To speed up the training process, we train on several different machines and then merge together the resulting genome. Our merging creates a new genome containing the union of all the discovered model states from each of the original genomes. The merging algorithm recalculates each of the knowledge entries to simulate the results of a serial computation. The pseudo-code in Figure 6.6 describes the specifics of the knowledge entry merging.

Because each training run depends on modifications to the genome from all previous runs, the mode selections from a merged genome are not the same as the mode selections from training a single genome over the same number of total runs. The payoff function, however, is the same regardless of whether the training occurred in parallel or on a single machine. Therefore, the resulting knowledge entries in a merged genome are just as valid as those in serially created genomes. The only difference might be that the distribution of occurrences may be more sparse, resulting

```
// genomeModelStates: all model states in the new genome
// programModes: the set of program modes
// n: number of original genomes
MergePayoffs (genomeModeStates, programModes, n) {
  for all (S in genomeModelStates) {
    for all (M in programModes) {
      payoffSum = 0;
      totalOccurrences = 0;
      for all (i from 1 to n) {
        payoffSum += payoff(S,M,i) * occurrences(S,M,i);
        totalOccurrences += occurrences(S,M,i);
      }
      mergedPayoff(S,M) = payoffSum / totalOccurrences;
    }
  }
}
```

Figure 6-2: Pseudo-code for merging multiple genomes: payoff(S,M,i) is the payoff for mode M given state S for original genome i. occurrences(S,M,i) is the number of times mode M was chosen given state S for original genome i. mergedPayoff(S,M) is the new expected payoff value for mode M given state S

in the payoff values converging more slowly.

In our experience, the difference between a merged genome file and a serially created genome file with the same number of total runs is minimal; we did not find a noticeable difference in mode selection or convergence.

One concern with the reinforcement learning technique is the possibility of too many model states. One of the Droid Wars teams has a model containing 205 properties, which allows for 2^{205} states. Keeping track of all the knowledge from a genome containing 2^{205} possible states may require too much memory for the technique to be practical. Fortunately, our experiments show that after after 3000 training runs (enough for all of the knowledge entries to converge), only a few hundred of the 2^{205} representable model states are ever reached. After looking at the code by hand, we found that the vast majority of the 2^{205} states are impossible configurations because of constraints on the relations between properties; some contradict or imply each other. It is still possible, however, that placing the system in a new environment will introduce new program states, and hence new model states, that could never exist in the training environments (for the two Droid Wars teams, we did not experience this new model states problem).

In the genome files for one of the teams, team20, over 98 percent of the knowledge entries clearly prefer only a single mode, meaning for that model state, the expected payoff values for one mode converges to 1.00 during training while the other modes converge to 0.00. The remaining knowledge entries show a preference for multiple modes, where two different modes have non-zero expected payoffs. For team17, the amount of knowledge entries showing preference for only one mode is much lower, about 80 percent. After inspecting the team17 properties and source code, we believe that the properties do not accurately capture all the important mode decision making details, causing two different program states to map to the same model states. Because those two program states map to the same model state, then the training will be unable to distinguish between the two cases and can only narrow the decision down to the pair of modes.

It may be possible to use the percentage of ambiguous model states for determining whether the properties are sufficient for capturing all necessary mode selection decisions. Even if the deployment of the reinforcement learning is not successful, at the very least this technique could give a valuable heuristic for evaluating the original program steering implementation. When almost all of the knowledge entries prefer only one mode, then one can conclude that the model states truly are equivalence classes with respect to program state mode selection.

Chapter 7

Related Work

Current approaches to the design of adaptive software make it difficult and expensive to build systems that adapt flexibly to a wide variety or range of changes in operating conditions, but that also operate efficiently under normal operating conditions. Approaches based on control theory, for example, require designers to identify important system inputs and outputs, and to model precisely how changes in input behavior affect output behavior. Systems built using such approaches generally operate efficiently in normal or near-to-normal operating conditions, but fail to adapt to extreme or unanticipated changes in operating conditions. Approaches used for intrusion detection require designers to distinguish between normal and abnormal patterns of use. Systems built using such approaches deal poorly with unanticipated patterns of use. Approaches based on fault tolerance require designers to anticipate the kinds and numbers of faults that may occur, although the designers need not model the precise effects of those faults. Systems built using such approaches may tolerate a wider range of changes in operating conditions, but operate less efficiently under normal conditions because they are always prepared for the worst.

Our technique shares some similarities with profile-directed optimization [BGS94, CMH91]. Profile-directed, or feedback-guided, optimization uses information from previous runs. For instance, if a set abstraction typically contains very few elements, it might be represented as a list rather than as a hash table. Or, if two pointers are rarely aliased, it may be worthwhile to check whether they are aliased and, if not, load them into registers rather than manipulate them in memory. Such optimizations must be checked at run time unless they do not affect correctness. Value profiling [CFE97, CFE99] examines values returned by load instructions. Our work is at a higher level of abstraction: we compare relationships between variables and data structures in the programming language rather than at the level of machine instructions. For example, if a list is usually sorted, it may be worthwhile to perform an $O(n)$ sortedness check before invoking an $O(n \log n)$ sorting routine. In this respect, our work shares similarities with work on specification-based program optimization [VG94], where a small amount of automated theorem proving at run time determines whether a precondition is satisfied for using an efficient, specialized version of a particular procedure.

Most previous work on program steering has addressed interactive program steering, which provides humans the capability to control the execution of running programs by modifying program state, managing data output, starting and stalling program execution, altering resource allocations, and the like [GVS94, ES98]. The goal is typically performance, and human observation, analysis, and intuition is inherent to the approach. By contrast, we are interested in automatic program steering. Miller et al. [MGKX01] discuss how to safely execute steering operations, by ensuring that they occur between, not during, program transactions, at a point when the program state is consistent. Debenham [Deb00] suggests checking properties (similar to anomaly detection) in order to note when a mode is inappropriate; if no solution can be found, a human operator is asked for assistance.

Reactive systems change their behavior or state in response to events their environment, but the changes are typically programmed in from the start. Likewise, much research in the AI field that automatically chooses among behaviors focuses on low-level control such as activating a motor rather than selecting a high-level goal; the latter is typically performed by a less adaptive high-level control program (if it is explicit at all). Or, a hard-coded hierarchy may indicate priority levels among modes, but the hierarchy is unlikely to be appropriate in every circumstance. Liu et al. [LCBK01] discuss the problems of incompatible mode specifications and proposes a constraint system to solve it; they improved performance of a simulated power-critical Mars rover application. Ghieth [GS93] discusses policies for intercepting object invocation and rerouting the invocations to a specific implementation. Richter et al. [RZE$^+$99] discuss selecting among modes of operation during system design and enabling run-time switching, but do not automatically provide policies for switching.

7.1 Execution Classification

Bowring et al. [BRH04] implemented a technique for automatically classifying code execution; they analyze code execution paths over a test suite and for each execution, create a model out of the executions branching statistics. A cluster analysis groups together similar model to form a new models describing generalized branching statistics for executions in each cluster. The models enable new executions to be classified into one of the clusters and then the cluster model itself adjusts itself based on the new information. Their technique could provide an online algorithm for automatically identifying program modes.

7.2 Machine Learning

Extracting knowledge from successful training examples is a goal in several fields of machine learning. Program steering shares characteristics found in both supervised and unsupervised learning problems. In unsupervised learning, the subject program makes decisions attempting to maximize a fitness function. With any set of decisions, the program will find out the quality of the end result but will not be told whether the decisions were optimal. Section 6.5 discussed one idea for enabling programs to dynamically adapt to new environments using an online unsupervised learning technique.

In supervised learning, the subject program makes decisions and receives feedback on whether the decision was correct or incorrect. The program attempts to modify its decision making process so that under the same conditions, the chances of making a correct decision are higher. Supervised learning may benefit program steering by training the new mode selector to use more appropriate similarity score policies other than uniformly weighting the model properties. Section 8.4 provides more details.

Chapter 8

Future Work

The results from the current implementation of program steering on the Droid Wars and fish programs have been very encouraging. Important steps for the future include exploring other subject programs in new problem domains, refining the modeling techniques to cope with noise, improving mode selection policies, and exploring ways to automate controller augmentation.

8.1 New Subject Programs

Applying program steering to other systems is essential for determining the strengths and weaknesses of program steering across a broad range of problem domains. Both the fish and Droid Wars programs were written by students and were never meant to be production quality code. Although the students' grades or contest performance were at stake, there was never any significant pressure to ensure best practices during development. In some cases, the code was not very robust or well tested, even without changing the environments. We would like to experiment with public domain systems that have already undergone a significant amount of programmer scrutiny and evaluation. Improving these programs may have a greater impact on both the acceptance of the technique and on benefiting the community using those systems.

One promising lead is the set of publicly available machine players for the popular Quake first-person shooter computer game, where human and machine players control a simulated warrior in a dungeon filled weapons, traps, and treasure. The Quake engine is freely available and open source `http://www.idsoftware.com/business/techdownloads/`, and as a result recreational programmers have created over two dozen machine players of varying skill, some regularly defeating the best human players (not a surprising result given that the game requires a lot of hand-eye coordination and motor skills). Laird et al. [Lai01] improved a Quake machine player by giving the AI the ability to analyze its own internal tactics and use the data to anticipate its opponents' actions. By exploring an opponent's likely future strategy, the AI could select its own favorable counter strategy .

There are several other candidate games in which machine players and humans players interact (and the machine player source code is freely available). The once-popular Bolo tank game by Stuart Chesire inspired over a dozen Bolo brain writers and at least two research projects in machine learning [WI95, Cor03]. Multi-user dungeon games such as Nethack and Angband also have machine players or human assistant "cyborg" players available.

Some hedge funds and financial research groups utilize autonomous financial trading systems to spot statistical arbitrage opportunities or other exploitable factors in financial markets. Computerized autonomous trading also has the advantages of fast data processing and judgment unimpaired by human emotions. One could try to could combine several of the autonomous trading strategies into a single system that chooses the most appropriate strategy based on market conditions and available data.

Another class of systems that can benefit from program steering are those whose behavior depends on preconfigured policy files or user preferences. Instead of making decisions about switching modes at runtime, a program could learn instead to switch between policy files. As examples, a graphical interface may learn to automatically switch between various sets of preconfigured settings based on what the user is attempting to accomplish, or a router might select one of several of its congestion control policies based on network traffic patterns.

A very ambitious goal is to upgrade a physical robot, such as the robots from the Annual MIT Autonomous Robot contest (6.270), the MIT MAS Lab contest robots (6.186), or the Sony soccer playing dog robots (RoboCup). The task would include all the challenges found in a typical program steering implementation as well as additional restrictions on computing power, programming language support, automation, and reproducibility.

8.2 Mode Coverage Experiments during Training

We hypothesized that training runs are very critical to generating models. An interesting experiment could measure how much the models change when given examples of less successful behavior.

The training runs can be restricted to only canonical executions of the mode, or could also include as much coverage within a mode as possible. For example, when attempting to create models for a robot navigation program, should the training for carpet-mode take place only on standard carpets where all the carpets are similar in nature except with just enough variation to avoid over-fitting? Alternatively, should the robot train on as many types of rugs as possible to gain as much knowledge as possible.

8.3 Sampling Data during Modeling

Program steering mode selectors classify program states into one of several available models, and the choice of classification determines the chosen mode of operation. The current modeling implementations construct one model for each mode, but the classification might be more effective with multiple models associated with each mode. To generate more models, the modeling tool could analyze several randomly selected samples from program trace data from the training runs and then generate models for each smaller sample.

There are three potential benefits to sampling and generating multiple models. The sampling provides noise resistance because outliers are unlikely to be chosen as part of the sample. The models are likely to be more detailed and specific because the models only attempt to describe a small number of data points relative to the entire data set. Having multiple models available also gives the mode selection policies more flexibility. The selector can calculate the similarity scores between the program state and all of the models and choose the mode with the closest matching model. Alternatively, the selector can average the similarity scores for all the models associated to each mode and choose the mode with the highest average. Another possibility is combining the multiple models associated with each mode back into one model, with the advantage that the resulting model contains the union of all the information from the more specific and noise-resistant models.

8.4 Refining Property Weights

The mode selector in Section 3.3 gives all properties in the model equal weight when computing a similarity score. We hypothesize that many programs can benefit by a weighting system that places more emphasis on the more important properties. For example, in the laptop display controller from Section 2.2, the existence of DC power may be much more important than properties about the brightness of the display.

One supervised learning approach uses the original mode selector as a basis for correct decisions in the original environments. The new mode selector can observe the original mode selector using a similar setup to the reinforcement learning training in Section 6.3.2. Each time the original mode selector makes a decision, the new mode selector adjusts its model property weights to raise the similarity score of the correct model and decrease the scores of the incorrect ones. The goal of this training is to construct a new mode selector that makes exactly the same decisions as the original mode selector in original environments. If the training successfully achieves that goal, we hypothesize the new mode selector is better equipped to make correct decisions in new environments because it takes in account many more properties than the original.

8.5 Projection Modeling

This sections describes Projection modeling, a different modeling technique that may provide more accurate mode selection.

So far, our modeling techniques focus on modeling the program behavior when operating in specific modes. Programmers then use the models to create the mode selector and attempt to find opportunities for calling it. Under such a scheme, we decouple two separate problems: determining when the program is operating in an unexpected state, and determining which of the program's modes are most appropriate for that situation.

Projection modeling combines both of those problems into one, so that the models explicitly contain information about where the program should switch modes. We define the Environment of a mode to be all the input conditions that are true when operating in that mode, for example, in the laptop display example:

```
Env(Power-Saver-Mode) = { batteryLife < 0.2 , DCPower = false }
```

Define the Projection of Mode A onto Environment B, abbreviated as $Proj(A_B)$, to be all the properties that are true when operating in Mode A inside Environment B. Note that $Proj(A_B)$ may contain properties that are not found in either Environment A or Environment B. As a simple example, any projection onto its own environment, such as the Projection of Power-Saver-Mode onto Env (Power-Saver-Mode), is exactly the same modeling previously described in Section 3.2. The models in the fish and Droid Wars experiments consisted of all the modes projected onto their own environments.

The more interesting cases are projections of modes onto different environments. In the laptop display controller example from Section 2.2 on page 20, the Projection of Normal-Mode onto Env (Power-Saver-Mode) would include the properties:

```
brightness > 0
brightness <= 10
batteryLife < 0.2
DCPower = false
```

The projection of Mode A onto Environment B contains all the properties true when the program is currently operating in Mode A, but *should* be operating in Mode B (where should is defined by the original programmer's intention in the original environment). In other words, the projection of Mode A onto Environment B includes the input-output relations the system exhibits when operating in Mode A, but with all other information indicating that Mode B is optimal.

In the above example of the Projection of Normal-Mode onto Env (Power-Saver-Mode), the display controller operating in Normal-Mode will exhibit the properties `brightness > 0` and `brightness <= 10`. However, `batteryLife < 0.2` and `DCPower = false` are properties true in Power-Saver-Mode. When the system is operating in Normal-Mode and the program state is similar to this particular model, that is, when the brightness is set to a high value but the batteryLife is low and DCPower is unavailable, then Power-Saver-Mode is likely to be the appropriate mode.

For this example, the mode selection choice is no different from the example in Figure 2-1. Using projection modeling, however, the system could simultaneously deduce that Normal-Mode was not an optimal mode for the program state and that Power-Saver-Mode is the best suited mode. In other systems, the mode selection using Projection models may not be the same as mode selection using the modeling technique from Section 3.2. Experiments in the future may determine the specific situations to use one versus the other (or whether one modeling technique dominates the other).

After modeling the projection of every mode onto every environment, a total of N^2 models given N modes, so the mode selection should be more accurate than the previous modeling and selection implementations. Projection modeling couples together both the anomaly detection and the mode selection. Each individual model contains the destination environment for that situation, which we defined as the most appropriate mode determined by the original programmer.

At run time, the mode selector calculates all of the various projections from its current mode onto each of the environments. The most appropriate mode is the environment whose projection from the current mode provides the most similar match. Suppose a system with N modes is operating in mode k, and we have modeled all N^2 projections. To determine whether the program should switch modes, the controller computes similarities scores between its current program state and each of the projections of k onto the other environments, that is, $Proj(k_1)$, $Proj(k_2)$, $\ldots$, $Proj(k_k)$, $\ldots Proj(k_N)$. If $Proj(k_k)$ is most similar to the program state, then the program is in the correct mode. If any of the other $Proj(k_i)$ models are most similar, where $k \neq i$, then the program switches to Mode i.

8.5.1 Transforms to Achieve Projection Modeling

A programmer can implement the projection modeling policy without too much additional work beyond the program steering implementation described in Chapter 3. Because training can only occur in expected environments, the original subject program should never reach a situation when the program is operating in an incorrect mode. Therefore, without modifications to the source code, the dynamic analysis will not be able to compute the operational abstractions for any of the projection models other than the trivial case of the projection of a mode onto its own environment.

To enable the dynamic analysis to model $Proj(A_B)$, a human could replace every transition to Mode B in the original source code with a transition to Mode A. Also, the code would need to contain a new field to help distinguish between normal executions of Mode A and the artificial executions. The field is simply a boolean flag identifying the execution of Mode A was supposed to be an execution of Mode B. When running under expected conditions with this change, then all of the original transitions to Mode B become specially tagged executions of Mode A. During the dynamic analysis step of program steering, the artificial Mode A calls, which help in generating the model $Proj(A_B)$, should be analyzed separately from the normal calls, which help in generating $Proj(A_A)$.

The same rewriting and tagging idea can scale to N modes by introducing an artificial execution flag for each pair of modes. Computing each projection model separately could be time-consuming both in machine-time and human-time; the number of projection models grow quadratically in the number of modes.

It may be tempting to minimize the number of code rewrites and test runs by computing multiple projections at once. For example, instead of simply replacing all transitions into Mode B might be replaced with a transition to Mode C, giving Proj(C, Env B), the transitions to Mode C and also be replaced to transitions to Mode B, simultaneously giving Proj(B, Env C). More ambitious modifications can attempt to cover all the projection models in a single run with only one rewrite, where all transitions have a random change of being swapped to another mode.

These shortcuts might not produce the same projection models as computing each model separately. Simultaneous computing creates a larger deviation from the original behavior, therefore undermining the quality of the Env() properties. We hypothesize that individually calculating each projection model will produce the most desirable models.

8.6 Other Modeling Tools

Program steering does not depend on use of operational abstractions or dynamic invariant detection. Program steering can utilize any method of extracting properties from training input and any representation of the resulting properties, so long as a distance metric exists either between the models and a program state, or among the models themselves. We have had good success with dynamic invariant detection, but it would be worthwhile to compare the results when using other modeling strategies. Do all machine learners perform equally well? Are there certain characteristics of dynamic invariant detection that make it particularly attractive (or not) for program steering?

Chapter 9

Conclusion

This thesis proposes an approach to making software more adaptable to new situations that its designers and developers may have neither foreseen nor planned for. The technique, called program steering, is applicable to multi-mode systems in which a controller selects an appropriate mode based on its inputs or its own state. Program steering generalizes from observations of training runs on which the software behaved well, and produces a new mode selector that, given a concrete program state, selects the mode whose past executions were most similar to the given state.

Program steering reduces dependence on developer assumptions about what the controller should and should not take into account. Instead, the new mode selector uses all the available information gathered during the modeling step. The technique requires no domain specific knowledge or human direction, only an existing controller that works well in expected situations and a way to determine which test runs are successful enough to become training runs.

We have implemented program steering and performed two sets of preliminary experiments to evaluate its efficacy. The first set of experiments, applied on simulated fish control programs, show that the technique is domain independent and can recreate an already successful controller without any help from the original.

The second set of experiments upgraded five multi-mode robot control programs from a real-time combat simulation and evaluated the new mode selectors in six new environments. Use of the new mode selectors substantially improved robot performance, as measured by improvements in rank in a 27-team tournament.

We also propose potential improvements to our original program steering policies; one alternative models program states instead of program modes and trains to select modes using a reinforcement learning algorithm. The preliminary results show that the technique is computationally feasible and can be useful for both creating more precise mode selection or for validating whether the operational abstractions discovered are adequate models. Other new policies include refining property weights to improve on uniform weighting and more fine-grained modeling to capture more details during mode transitions.

The positive results warrant future work in program steering. To learn more about the strengths and applicability of the the technique, program steering should be applied to more subjects programs; possibilities include multi-player online games, financial applications, programs that rely on preconfigured policy files, and hardware controllers.

Bibliography

[BAS02] Rajesh Krishna Balan, Aditya Akella, and Srini Seshan. Multi-modal network protocols. *SIGCOMM Comput. Commun. Rev.*, 32(1):60–60, January 2002.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[BRH04] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 195–205. ACM Press, 2004.

[CFE97] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-97)*, pages 259–269, San Jose, CA, December 1–3, 1997.

[CFE99] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, March 1999. `http://www.jilp.org/vol1/`.

[Cha02] John Chapin. Personal communication. CTO, Vanu Inc., 2002.

[CMH91] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 1991.

[Cor03] Kyle Corcoran. Temporal difference corpus learning: Experience based learning in a strategic real-time game for inproved computer play. Princeton University, Senior Thesis, `http://www.cs.princeton.edu/ugprojects/projects/c/corcoran/senior/thesi%s.pdf`, 2003.

[Deb00] John Debenham. A multi-agent architecture for process management accommodates unexpected performance. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 15–19, Como, Italy, March 19–21, 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[ES98] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 10–20, Welches, OR, August 3–4, 1998.

[GS93] Ahmed Ghieth and Karsten Schwan. CHAOS-Arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[GVS94] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, September 1994.

[Lai01] John E. Laird. It knows what you're going to do: adding anticipation to a quakebot. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the Fifth International Conference* *on Autonomous Agents*, pages 385–392, Montreal, Canada, 2001. ACM Press.

[LCBK01] Jinfeng Liu, Pai H. Chou, Nader Bagherzadeh, and Fadi Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In *Ninth International Symposium on Hardware/Software Codesign*, pages 153–158, Copenhagen, Denmark, April 25–27, 2001.

[LCKS90] Nancy G. Leveson, Stephen S. Cha, John C. Knight, and Timothy J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, April 1990.

[LE03] Lee Lin and Michael D. Ernst. Improving reliability and adaptability via program steering. In *Fourteenth International Symposium on Software Reliability Engineering, Supplementary Proceedings*, pages 313–314, Denver, CO, November 17–20, 2003.

[LE04] Lee Lin and Michael D. Ernst. Improving adaptability via program steering. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 206–216, Boston, MA, USA, July 12–14, 2004.

[MGKX01] David W. Miller, Jinhua Guo, Eileen Kraemer, and Yin Xiong. On-the-fly calculation and verification of consistent steering transactions. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 1–17, Denver, CO, USA, November 10–16, 2001.

[RZE+99] K. Richter, D. Ziegenbein, R. Ernst, L. Thiele, and J. Teich. Representation of function variants for embedded system optimization and synthesis. In *Proceedings of the 36th ACM/IEEE Design automation conference*, pages 517–522, New Orleans, LA, USA, June 21–25, 1999.

[Tes95] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[VG94] Mark T. Vandevoorde and John V. Guttag. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94)*, pages 121–127, New Orleans, LA, USA, December 6–9, 1994.

[Wei02] Robert K. Weiler. Automatic upgrades: A hands-on process. *Information Week*, March 2002.

[WI95] Andrew Wilson and Stephen Intille. Programming a bolo robot: recognizing actions by example. Class Project Report, `http://web.media.mit.edu/~intille/inducting_indy.html`, 1995.