Improving the Adaptability of Multi-mode Systems via Program Steering

Lee Lin Michael D. Ernst

MIT Computer Science & Artificial Intelligence Lab
Cambridge, MA 02139 USA
{leelin,mernst}@csail.mit.edu

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 creating a new mode selector that examines the current situation, then chooses a mode that has been successful in the past, in situations like the current one. The technique, called program steering, creates a new mode selector via machine learning from good behavior in testing or in successful operation. Such a strategy, which generalizes the knowledge that a programmer has built into the system, may select an appropriate mode even when the original controller cannot. We have performed experiments on robot control programs written in a month-long programming competition. Augmenting these programs via our program steering technique had a substantial positive effect on their performance in new environments.

Categories & Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.4 [Software Engineering]: Software/Program Verification
General Terms: performance, design, reliability, experimentation
Keywords: program steering, mode selection, multi-mode systems, adaptability

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 [18]. 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 software systems to react appropriately to unanticipated circumstances.

Our research focuses on multi-mode software systems. 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. The mechanism for selecting modes is fixed when the program is coded. We hypothesize that, for the most part, programmers effectively and accurately select modalities for situations that they anticipate. However, the selection mechanism 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.

We aim to provide flexible and automatic ways to select among modalities. Our approach, called program steering, is to first train the system, correlating modes with program inputs and internal state, and then to use the resulting models to select modes at run time. We select an operating mode by comparing the system's current behavior to the system's observed behavior in representative runs for each of the modes, and by steering the system to the most similar mode. The effect is to extend the programmer's built-in knowledge to analogous situations and improve upon sub-optimal or overspecialized built-in decisions.

This paper is organized as follows. Section 2 gives a simple, concrete example of the program steering process. Section 3 describes program steering in detail, including policies for the parts of a program steering tool. Section 4 presents our experimental methodology and results. The paper concludes with related work

ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.

Current Program State	Standard Mode	Power Saver Mode	Sleep Mode
$brightness$: 8	$\mathbf{brightness > 0}$ $\mathbf{brightness \leq 10}$	$\mathbf{brightness > 0}$ $brightness \leq 4$	$brightness = 0$
$battery$: 0.1	$battery > 0.2$ $\mathbf{battery \leq 1.0}$	$\mathbf{battery > 0.0}$ $\mathbf{battery \leq 0.2}$	$\mathbf{battery > 0.0}$ $\mathbf{battery \leq 1.0}$
$DCPower$: true		$DCPower = \text{false}$	
Score	75%	60%	66%

Current Program State	Standard Mode	Power Saver Mode	Sleep Mode
$brightness$: 8	$\mathbf{brightness > 0}$ $\mathbf{brightness \leq 10}$	$\mathbf{brightness > 0}$ $brightness \leq 4$	$brightness = 0$
$battery$: 0.1	$battery > 0.2$ $\mathbf{battery \leq 1.0}$	$\mathbf{battery > 0.0}$ $\mathbf{battery \leq 0.2}$	$\mathbf{battery > 0.0}$ $\mathbf{battery \leq 1.0}$
$DCPower$: false		$\mathbf{DCPower = false}$	
Score	75%	80%	66%

Figure 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.

(Section 5), future work (Section 6), and a summary of contributions (Section 7).

2. EXAMPLE

This section presents a simple example of program steering. 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.

As an example of a multi-mode system, we use a hypothetical laptop display controller that contains three modes: standard mode, power saver mode, and sleep mode. 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). The original mode selection code need not necessarily make use of all (or any) of these variables.

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 the model is an operational abstraction (see Section 4.1.2), a set of mathematical statements about program variables that was satisfied by all the observed executions. The resulting models might be:

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

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 the current state of the program and its environment and determines which model most closely matches current conditions. One simple metric is the percentage of properties in the model that currently hold. Figure 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.

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 2 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 section 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.

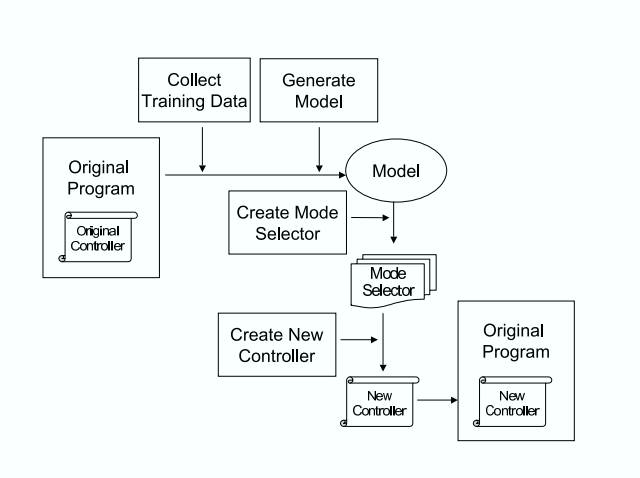

Figure 2: The program steering process consists of four steps: executing the original program to produce training data; generalizing from the executions to a model for each mode; creating a new mode selector based on the models; and augmenting the program's controller to utilize the new mode selector.

3.1 Training

The modeling step generalizes from the training data, and the mode selector bases its decisions on the models. Therefore, better training runs yield better overall performance. 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.

The user might supply canonical inputs that represent the situations for which that 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.

Sometimes a test suite or specification is not available, particularly for non-functional requirements. (For example, module-level specifications may guarantee that each part of a system performs in a particular way, but not that the overall system achieves its goals.) In this case, the program can be run and its performance observed. This process can be performed manually by a domain expert, or it can be automated if there is an objective function for evaluating the quality of a run. 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. (The reverse situation may also occur.) Alternately, the mode transitions may have been perfect, but because of bad luck the overall performance failed to meet the acceptance threshold.

3.1.1 Counterexamples

The training runs can also include counterexamples, or instances where specific modes were inappropriate. This information may help at runtime to indicate when the program has made (or is about to make) a poor mode selection. Additionally, many machine learners require, or perform better when supplied with, counterexamples. However, depending on the target system, useful counterexamples may be difficult to produce. If a programmer or tester thinks of a counterexample, the code is likely to be updated to correct the oversight; counterexamples by definition come from unforeseen circumstances. And as noted above, good (respectively, bad) overall system performance does not indicate that every mode choice was good (respectively, bad).

When counterexamples are used, they can be incorporated into the model, or additional models can be created from bad executions of a mode. In the latter case, the additional models describe the program's environment and behavior when the program is operating in one mode, but should be operating in another one.

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 the current program state and environment (inputs) to each model. 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 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 [12].

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 to customers 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 outperforms 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 Sample applications

Section 1 noted several domains to which program steering might be applied. This section describes three application areas — routers, wireless communications, and graphics — in more detail. Our experiments (Section 4) evaluate another domain: controllers for autonomous robots in a combat simulation.

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 [2], 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 [6], 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.

3.6 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.

4. DROID WARS EXPERIMENTS

In order to evaluate program steering, we applied it to robot control programs built as part of a month-long Droid Wars competition known as MIT 6.370. 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.[1]

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.

[1] A separate paper [13] describes another experiment, which used solutions to a research assignment in MIT's Embodied Artificial Intelligence graduate class. The assignment objective was to program simulated fish that could self-organize into a school without any communication and while avoiding rocks. Program steering, when applied to the best programs, produced controllers that did just as well. When applied to low-performing programs (such programs had no high-level modes), the results of program steering also performed equally to the original programs, for the modeling step had been unable to make good generalizations.

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 3: 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.

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 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 3 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.

4.1 Applying program steering

As described in Section 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 for refactoring the original programs to isolate the modes and the mode selector.

4.1.1 Training

Mode selection requires a multi-mode program, and our implementation of program steering further requires that each mode transition is represented by calling a distinct method. (Not every method necessarily represents a mode, however.) We read the programs to identify the modes. Sometimes this required simply noting the names of the methods that represented mode shifts. In other cases, modes were embedded in other code, so we refactored the code

(without affecting its functionality) in order to make mode transitions into method calls. A different implementation of program steering would not have required this refactoring step. We hypothesize that mode identification, and refactoring if necessary, would have been relatively easy for the original authors of the code. It was harder for us, but still no more than a few hours of work. 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.

For each program, we collected training data by running approximately 30 matches 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 4.2, 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 retained only the runs from 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.

4.1.2 Modeling

In our experiments, the modeling step is carried out completely automatically, and independently for each mode, producing one model per mode.

Our current implementation uses the Daikon invariant detector (`http://pag.csail.mit.edu/daikon`) for the modeling (machine learning) step. The resulting models are *operational abstractions*, which are sets of program properties expressed as logical formulae, each associated with a method call that represents a mode transition. (Section 2 gives some very simple examples.) An operational abstraction is 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 models included properties over both environmental inputs and internal state variables, described in Section 2.

Operational abstractions are produced by a process called dynamic invariant detection [9]. 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.

The program steering technique 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?

```
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 4: Automatically generated mode selector for the laptop display controller of Section 2. The given section of the mode selector evaluates the appropriateness of the display's Standard Mode.

We made one small program modification 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. 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.

4.1.3 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 1.

Figure 4 shows code for a mode selector that implements our policy of choosing the mode with the highest percentage of matching properties.

4.1.4 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 akin to the refactoring noted in Section 4.1.1. We did not replace the old mode selector by our new one, but augmented it with the new one. 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.

4.2 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.

4.3 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 won the most games. (We played multiple matches per pair of teams because each match used a randomly-generated map, as described in Section 4.2.) 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 shows the results of each set of tournaments, giving the original rank of each team and the rank after the program steering upgrades.

The positive results reported in the Change column of Figure 5 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, our mode selector substantially outperformed the random mode selector.

4.4 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

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: Difference in performance between the original programs and versions upgraded with program steering. 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.

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. 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.

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

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.

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 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 1 and 4) 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.

5. 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 [1, 5]. 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 [3, 4] 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 [17], 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 [11, 8]. 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. [15] 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 [7] 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. [14] 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 [10] discusses policies for intercepting object invocation and rerouting the invocations to a specific implementation. Richter et al. [16] discuss selecting among modes of operation during system design and enabling run-time switching, but do not automatically provide policies for switching.

6. FUTURE WORK

Our preliminary experiments suggest that, at least in some circumstances, program steering can generalize training runs to create a new mode selector that significantly outperforms the original system in new environments. However, additional work is required before we can conclude that the technique applies to real-world situations, and future research will also indicate which sorts of programs and environmental changes the technique is best (and worst) at handling. We believe that our preliminary results justify additional investigation, and here outline some directions for future work.

First, the technique needs to be run on more programs, and programs of more types, to better indicate its strengths and weaknesses. This will address the major threat to external validity that our current research suffers. Additionally, a survey of application domains could indicate how much real-world software in each domain has (or can easily be refactored to have) modes and a mode selector, which our technique requires. As noted earlier, we believe that it can be applied to enough real software to be worthy of further investigation.

The design space for program steering is large; Section 3 only partially explores it, and we have evaluated only one point in the space. That point appears to provide good results, but other points may be even better. We are eager to try different modeling approaches (particularly noise-resistant ones), different mode selection techniques (assigning different weights to different parts of a model seems particularly promising), and different policies for when a controller invokes the new mode selector.

One challenge to our methodology is achieving reasonable response time. Constructing a model is time-consuming for many machine learning techniques, but such a process happens only once, offline. The cost of evaluating a model is much lower, but may be a limiting factor in resource-constrained environments.

In future work, we would like not only to steer programs among existing modes, but also to introduce new modes. For instance, if two situations yield very different models, then perhaps the system can be optimized for those two situations, much as in profile-directed optimization. Such a task will require both recognizing different states and then introducing optimizations based on them.

7. CONCLUSION

We have proposed 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 preliminary experiments to evaluate its efficacy. We applied our program steering tool to 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 ranking in a 27-team tournament.

Acknowledgments

We thank Steve Garland for fruitful discussions that aided in the conception and clarification of the ideas of program steering, and the anonymous referees for their thoughtful comments on this paper. This research was supported in part by NSF grants CCR-0133580 and CCR-0234651 and by a gift from IBM.

REFERENCES

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[2] R. K. Balan, A. Akella, and S. Seshan. Multi-modal network protocols. *SIGCOMM Comput. Commun. Rev.*, 32(1):60–60, Jan. 2002.

[3] B. Calder, P. Feller, and A. Eustace. Value profiling. In *MICRO-97*, pages 259–269, Dec. 1–3, 1997.

[4] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1, Mar. 1999. `http://www.jilp.org/vol1/`.

[5] P. P. Chang, S. A. Mahlke, and W.-m. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 1991.

[6] J. Chapin. Personal communication. CTO, Vanu Inc., 2002.

[7] J. Debenham. A multi-agent architecture for process management accommodates unexpected performance. In *SAC*, pages 15–19, Mar. 2000.

[8] G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *SPDT*, pages 10–20, Aug. 1998.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, Feb. 2001.

[10] A. Ghieth and K. 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, Apr. 1993.

[11] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, Sept. 1994.

[12] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE TSE*, 16(4):432–443, Apr. 1990.

[13] L. Lin and M. D. Ernst. Improving reliability and adaptability via program steering. In *ISSRE Supplementary*, pages 313–314, Nov. 2003.

[14] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In *CODES*, pages 153–158, Apr. 2001.

[15] D. W. Miller, J. Guo, E. Kraemer, and Y. Xiong. On-the-fly calculation and verification of consistent steering transactions. In *SC2001*, pages 1–17, Nov. 2001.

[16] K. Richter, D. Ziegenbein, R. Ernst, L. Thiele, and J. Teich. Representation of function variants for embedded system optimization and synthesis. In *DAC*, pages 517–522, June 1999.

[17] M. T. Vandevoorde and J. V. Guttag. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *FSE*, pages 121–127, Dec. 1994.

[18] R. K. Weiler. Automatic upgrades: A hands-on process. *Information Week*, Mar. 2002.