Improving Reliability and Adaptability via Program Steering

Lee Lin   Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab
200 Technology Square
Cambridge, MA 02139 USA
{leelin,mernst}@csail.mit.edu

1. Introduction

Software systems often contain several discrete modes of operation and a mechanism for switching between modes. Even when a multi-mode system has been tested by its developer, it may be unreliable in the field, because the developer cannot foresee and test for every possible scenario; unexpected situations in which the program fails or underperforms (for example, by choosing a non-optimal mode) are certain to arise. This research mitigates such problems by creating adaptive modal programs that handle unanticipated scenarios by autonomously selecting an appropriate mode.

The technique creates a new mode selector via machine learning by training on good behavior in anticipated situations. The new controller can augment or replace the old one. Preliminary experiments indicate that our technique can re-derive ideal controllers and can improve the reliability and performance of good ones.

2. Program Steering

We assume as input a modal program: one with multiple discrete behaviors or input–output relationships. For example, cell phones behave differently depending on signal strength, interference, and other factors. A robot may operate differently on roads, in buildings, and on open terrain — but the designer may not have considered which mode is best in a damaged building or a road that is under construction.

We use program steering to improve modal programs. We build a mode selector by training on successful runs of the original program. Then, we give the program unlimited access to the selector at runtime.

Figure 1 illustrates the four-stage Program Steering process, which creates a new mode selector:

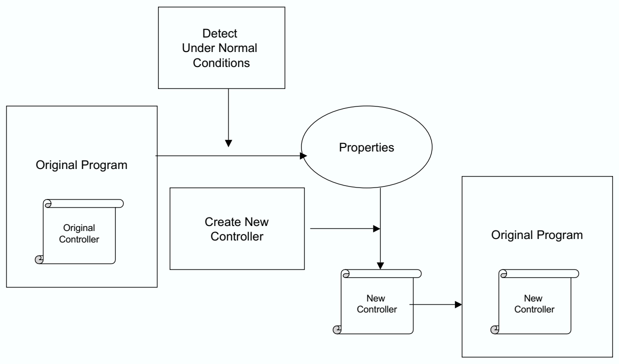

Figure 1. Program steering process

1. Collect training runs of the original program in which the program behaved as desired.
2. Use a dynamic program analysis tool to determine the properties that are true in each mode during the training runs.
3. Build a mode selector that takes as given the current program state and assigns a similarity score between the current state and each available mode. The selector picks the mode with the highest score. At present, our similarity score formula is the percentage of mode properties detected during the training stage that are satisfied at run-time, but we plan to experiment with additional formulas.
4. Augment the original program with a new controller that utilizes the mode selector.

There are many potential policies for the new controller that utilizes the mode selector. A system may completely replace its original controller. Alternately, the system may use the selector only after its original controller fails by throwing an exception, entering a passive mode, or reaching a system timeout. Programmers can also use the mode selector for debugging and verification purposes.

2.1. Property Detection

Our current implementation uses the Daikon tool to detect program properties from training runs. However, the program steering technique should be applicable to any method of extracting properties from training input.

The Daikon invariant detector reports properties that hold over program executions [1]. The reported properties are similar to assert statements or formal specifications. Daikon operates by running the program and examining the values it computes, looking for patterns and relationships among those values. It reports properties that were true for all the executions that it observed. There is no guarantee that those properties will hold on future runs, but the results are generally accurate, even with a modest test suite.

2.2. Comparison to previous approaches

Our approach is complementary to previous approaches to creating adaptive software, and holds promise to perform more reliably in unanticipated situations. 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. Approaches used for intrusion detection require designers to distinguish between normal and abnormal 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.

3. Preliminary Experiments

We ran two experiments to test our technique. One experiment determined whether the new mode selector could mimic a flawless controller. The second experiment determined whether the program steering technique could improve an adequate but imperfect controller.

3.1. Recreation of an Excellent Controller

Our first experiment applied program steering to four programs from MIT's graduate class 6.836 Embodied Artificial Intelligence. These programs were intended to cause fish to form schools while avoiding obstacles. Each fish controlled its velocity and 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 fish to join the school. Each program accomplished the goal. We removed the original controllers to see whether our mode selector could perform equally as well.

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 goal as well as the original controllers. The other two new controllers fared poorly in new environments. Upon investigation, we discovered that those programs were tested in a single initial configuration and contained hard-coded logic, for instance assuming specific obstacle locations: the original programs were not in the least intelligent or adaptable, and the new ones were no worse.

We conclude from this experiment that the initial program needs to be of at least decent quality. Furthermore, the training process is most effective when selecting between distinct, high-level modes (such as avoid-rock or follow-neighbor) and is less effective on low-level modes (such as turn-left-mode and turn-right-mode) that do not even depend on sensor data and whose properties are therefore very similar.

3.2. Adaptability Improvement

Our second experiment used contestant code from a month-long MIT AI programming contest (6.370) in which two teams of virtual robots compete in a real-time strategy simulation to accomplish a set of goals more quickly than the other team. The tasks include exploring the world, identifying and gathering useful resources, transporting allies, attacking enemies, defending the base, and relaying radio messages.

We analyzed a program eliminated during the semi-finals of the 24-team tournament The robot program contained seven active modes and four passive modes. During the passive modes, the robot waits for a message from an ally. After creating the mode selector described in Section 2, we inserted a timeout in each of the passive modes to invoke the mode selector and switch to the suggested active mode.

We matched the original code against the upgraded code with several of the game world parameters randomly altered in each game. The team with the augmented controller won 769 times out of 1000 matches. This result suggests that the upgraded code was better able to adapt to changes in the environment.

4. Conclusions

Our initial observations show three promising results. First, given reasonable test environments and mode architectures, program steering can effectively determine the set of program properties that should be used to make decisions in mode selection. Second, a mode selector built using those properties can successfully map program states to program modes with no help from the original program controller. Lastly, the mode selector can significantly improve how well a program adapts to new and unexpected environments.

References

[1] 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.