FAULT LOCALIZATION IN EMBEDDED CONTROL SYSTEM SOFTWARE

by

KAI LIANG

Submitted in partial fulfillment of the requirements
For the degree of Master of Science

Thesis Advisor: Dr. Soumya Ray

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

May, 2014

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

KAI LIANG

candidate for the Master of Science degree*.

(signed) Soumya Ray

(chair of the committee)

Andy Podurski

M. Cenk Çavuşoğlu

Mehmet Koyuturk

(date) Match 31, 2014

*We also certify that written approval has been obtained for any proprietary material contained therein.

To my friends who show me the light in the darkness

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Soumya Ray, for his great patience and careful guidance through the past three years. He is a knowledgeable person with great passion on everything he works on. It has been a pleasure to meet with him weekly, having discussions on either research or life. He always granted me enough flexibility on scheduling my time, as well as gave me insightful advices on difficulties I encountered. I appreciate these a lot and have learned how to treat other people the same way. It has been great fun to listen to and discuss artificial intelligence and other interesting topics with Soumya. I do learn a lot, and I think it will benefit me in my future life. It is my privilege to work with Soumya during my graduate time.

I am thankful to the group of people with whom I work on this project, Dr. Andy Podgurski, Dr. M. Cenk Çavuşoğlu, Zhuofu Bai. This work will be impossible without their teamwork.

I also would like to show my special thanks to my roomates Ye Xuan and Zhaofeng Gao, who always supports me for any decision I made, and share food and ideas in our life every week. Thank my parents for their supporting my studies and respecting my decisions. Without them I would not get a chance to enjoy this wonderful three years in Case!

LIST OF ABBREVIATIONS

A&A Events: Adverse and Anomalous System Events

API: Application Programming Interface

BHR: Beating Heart Robot

CPD: Conditional Probability Distribution

DBN: Dynamic Bayesian Network

FDA: Food and Drug Administration

FLECS: Fault Localization In Embedded Control System

LIDAR: Laser Radar

PFiC: Probability Of Failure If Covered

PDG: Program Dependence Graph

PPDG: probabilistic program dependence graph

GUI: Graphical User Interface

OOW: Out Of Workspace

RoS Systems: Robotic Surgery Systems

SABiR: Small Animal Biopsy Robot

Fault Localization in Embedded Control System Software

Abstract

by

KAI LIANG

Embedded control systems are built and used everywhere in modern society in many safety-critical applications. Therefore, bug-free control code is an important consideration. Using medical robot control systems, we develop a statistical approach to automatically locate faulty statements in control code. Our approach uses the controller structure and examples of normal behavior in simulation to construct structured probabilistic models that compactly encode the dynamic behavior of the systems. We describe techniques that are used to improve the model, including feature selection and starting with a prior structure defined using the program dependence graph (PDG) of the controller. Given an anomalous behavior sequence, we analyze the values of system state variables to determine which variables are responsible for such behaviors. We use the variables obtained in this way together with the dynamic program dependence graph to determine a small set of potential causes (faulty statements) of the behavior, which are then ranked and presented to the developer. We evaluate our approach on the control systems for two prototype robotic surgery systems developed in our lab and demonstrate its ability to locate faults that cause adverse and anomalous events during the systems' operation.

Chapter 1

Introduction

Embedded systems have become ubiquitous and play an essential role in the upliftment of human society. An embedded system is a combination of computer hardware and software, either fixed in capability or programmable, that is specifically designed for a particular function. Embedded systems which are programmable are provided with programming interfaces, and embedded systems programming is a specialized occupation. Figure 1.1 gives several examples of popular applications, for instance, industrial machines, automobiles, medical equipments, cameras, household appliances, airplanes, vending machines, toys, autonomous cars, etc.

Earlier this century, Google started a project called Google driverless car [5]. Several U.S. states have now legalized the use of self-driven cars for testing purposes. These autonomous vehicles could transform not only the way we travel, but also impact the way our society functions. Figure 1.2 shows a self-driving car. Besides common components like power-train, braking, in-vehicle networking and airbag systems, an autonomous car requires sophisticated components like radar, lane-keeping, LIDAR (laser radar), infrared camera, stereo, GPS/inertial measurement, wheel encoder etc., each with its own underlying embedded system mechanism. These custom embedded systems are required to detect obstacles on the road, especially for autonomous cars.

Embedded systems are widely used in robotics. Robots can be autonomous or semi-autonomous and range from humanoids to industrial robots. In either case, they are always controlled by a computer program. Figure 1.3 shows the da Vinci surgical robot which is used in hospitals all over the world. This robot, made by Intuitive Surgical, was approved by the

Figure 1.1 Some applications of Embedded System [1]

Food and Drug Administration for clinical use in 2000. According to a recent article [6], there were 400,000 surgeries are executed by da Vinci robots in 2012.

Examples such as automated vehicles and surgical robots illustrate that many embedded control systems are deployed in safety-critical environment where faults could have serious consequences. Significant work has been done to locate and prevent *hardware* failures in such systems [7, 8, 9, 10, 11]. Here the primary focus is placed on restricting the hardware components in stable regions of available configuration space. Comparatively little work exists that looks at failures caused by *software bugs* in their control code. In some scenarios, auto-coding [12] can be used to ensure that the controller code is free of faults. However, this is not always possible especially for complex systems such as robots. In our work, we focus on systems where the controllers are designed and written manually, and thus, anomalous behavior of

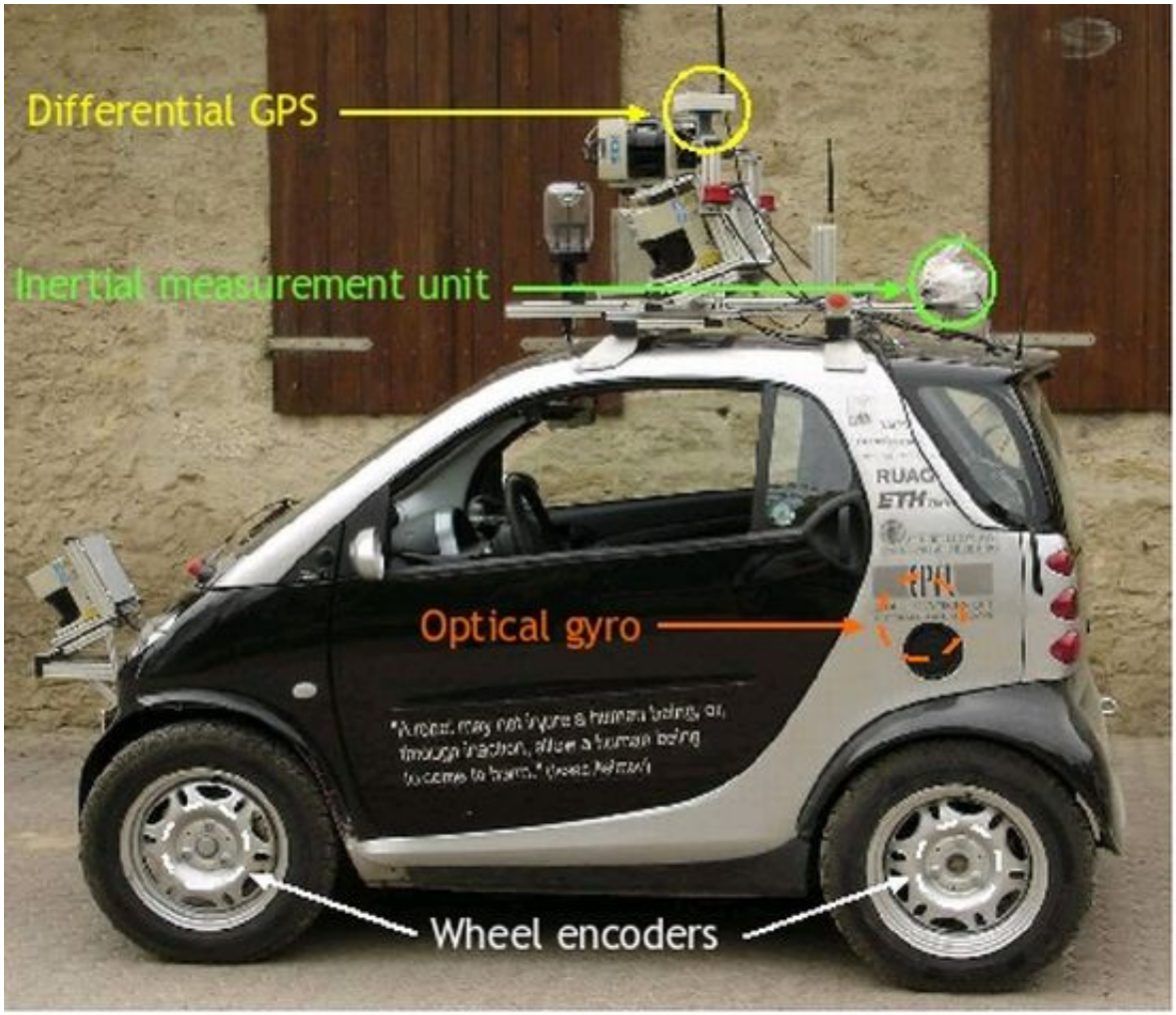


Figure 1.2 An autonomous car [2]

the systems due to software bugs is a possible concern. We present an approach to locating potentially faulty statements in the control code of embedded systems. This can help developers debug the control code, and so reduce or eliminate instances of anomalous behavior for these systems. We evaluate our approach on two testbed systems that are prototype surgical robots.

In the remainder of the thesis, we first provide an overview of the robotic surgery systems with descriptions of our testbed systems, two prototype surgical robots. We describe some work that is related to our method, for instance, parameter synthesis and coverage based fault localization techniques. After that, a description of our algorithm is presented in the third

Figure 1.3 The da Vinci Surgical Robot

chapter. In addition, we show our empirical evaluation and current limitations in the fourth chapter. A conclusion and a discussion of future work are given in the end.

Chapter 2

Background and Related Work

In this chapter, we describe Robotic Surgery Systems in terms of their advantages and disadvantages. Then two testbed controller of robotic surgical systems are described, in terms of robot component and the software controller. In the end, we briefly describe the coverage-based fault localization technique as a comparison with our approach and other related work.

2.1 Robotic Surgical System

RoSs (Robotic Surgery Systems) are modern surgical devices that use robotic systems to aid in surgical procedures. They perform surgery using small tools attached to a robotic arm. The surgeon controls the robotic arm with a computer. RoSs were developed to help perform actions that enhance minimally-invasive surgery and enhance the surgeon's capabilities when performing the surgery. In some circumstances, using such systems can allow access to locations where manual instruments have difficulty reaching to perform image guided surgery [7, 13]. These systems are increasingly becoming popular worldwide and are being deployed for practical use, because they are able to decrease the time in the operating theater, the patient recovery time, and decrease the chance of side effects. In addition, these systems make surgeries that were technically unfeasible or difficult, now possible.

A key issue is that most of these systems are very complex both in terms of their hardware and software. The complexity of the systems creates a possibility for accidents due to hardware malfunctions, software malfunctions, observability limitations (visualization), or human–machine interface problems. Those accidents cannot be ignored since they may cause the death

of the patients. Indeed such events have already occurred, as evidenced by a number of adverse and anomalous (A&A) events reports filed by manufacturers with the Food and Drug Administration (FDA). One such report [14] contains the following description of an event involving the da Vinci Surgical System [15]:

> *[D]uring a da Vinci's beating heart double vessel coronary artery bypass graft procedure at the hospital, there was an unexplained movement on the system arm which had the endowrist stabilizer instrument attached to it. [This] caused the feet at the distal end of the endowrist stabilizer instrument to tip downward resulting in damage to the myocardium of the patient's left ventricle.*

The accompanying "Manufacturer Narrative" states:

> *The investigation conducted by an isu field service engineer found the system to [have] successfully completed all verification tests and to function as designed. No system errors related to the cause of this event were found. Additional investigations conducted by clinical and software engineering were inconclusive as a root cause could not be determined based on the event descriptions reported by several eye witnesses and the review of the system's event logs.*

We say that an adverse and anomalous (A&A) event happens whenever the robot's actual movement deviates from the expected behavior by a certain amount and harm the patient's health. Since the robots we study are not currently employed in a clinical setting, we define an A&A event to be any behavior of the robot that deviates from expected behavior (and so has the *potential* to harm a subject). Our work in this thesis describes an approach to assist developers with automated tools to debug the control systems of these devices.

2.2 Testbed Hardware

In this work, we use two prototype robotic surgery systems which are developed and made at CWRU. One is a Small Animal Biopsy Robot (SABiR) [16]. The other one is a manipulator that will allow minimally invasive beating heart surgery (BHR) [17].

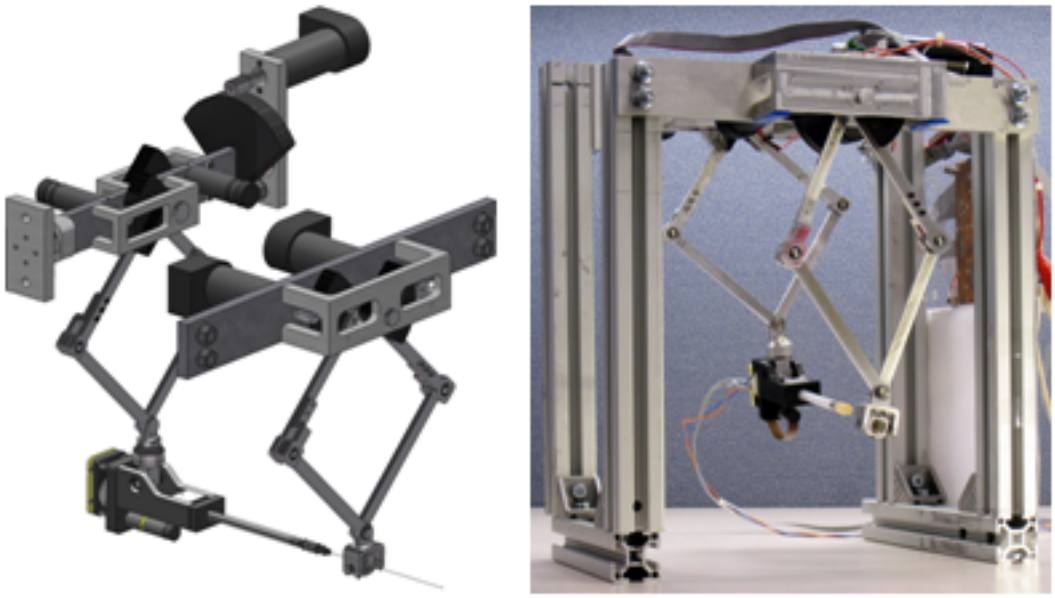


Figure 2.1 SABiR: The robotic system for image-guided needle-based interventions on small animals. Left: CAD model of the system. Right: A prototype system built in our lab.

2.2.1 SABiR

In this paragraph, we describe the Small Animal Biopsy Robot (SABiR). Figure 2.1 shows an image of the robot. It is a five-degree-of-freedom parallel robotic manipulator which is designed to take biopsies or deliver therapeutic drugs at targets in live small animal subjects. It employs a parallel design to achieve low inertia. The robot has high position resolution and can realize dexterous alignment of the needle before insertion. The design is lightweight and has high motion bandwidth, so that biological motion (e.g., breathing, heartbeat, etc) can be canceled while the needle is inserted in tissue. The robot consists of a needle mechanism held by two 5-bar linkage mechanisms, called the front and rear stages. The front stage has two degrees of freedom (up/down, left/right) and the rear stage has three degrees of freedom (up/down, left/right, rotate forward/rotate backward). The stages are driven by five tendon mechanism motors and the joint angles are measured by encoders. The robot's hardware state is characterized by its five joint angles, and there is a one-to-one correspondence between any position and orientation that the needle tip can reach and the set of joint angles.

Figure 2.2 BHR:Beating Heart Robot

2.2.2 BHR

The Beating Heart Robot (BHR) [18] uses a three-degrees-of-freedom (DOF) robotic platform employing a PHANToM Premium 1.5A haptic interface [19] as the robotic mechanism (Figure 2.2). The characteristic of the system makes it suitable for eventual use in minimally

invasive beating heart surgery [17], with sufficient motion and speed for heart motion tracking. The robot is composed of a base rotation and a four-bar linkage, holding a tool arm. The degrees-of-freedom are driven by three tendon mechanism motors (one actuating the base rotation, and two actuating the four-bar linkage in parallel) and the joint angles are measured by quadrature encoders. Its lightweight links, low inertia design and low-friction actuation system allows sufficient motion and speed abilities for tracking the heartbeat signal. The robot's hardware state is characterized by its three joint angles, and there is a one-to-one correspondence between any position that the tool tip can reach and the set of joint angles.

2.2.3 Simulation and Control

Both of the robots in Figure 2.1 and Figure 2.2 have an associated simulation of their hardware and its environment implemented in MATLAB Simulink. Simulations such as these are commonly used in control system deployment, for reasons of safety and practicality. It may not be safe to put an untested controller directly on the hardware, and since testing requires significant data collection, it is generally not practical in terms of time and expense to do this on the actual hardware. Rather, the majority of testing and debugging is done through the simulation, and the code is deployed on the hardware only after this step is satisfactorily completed. If the simulation is accurate, at this point there may be a further minor parameter tuning step, but we expect the resulting controller to work well on the hardware. The simulators are in fact designed to be modular components, so that they can be seamlessly swapped with the interface to the actual robot. We describe more details of simulator below.

First let us explain SABiR's simulator [1]. In prior work [16], we use models for kinematics and inverse kinematics developed to create a simulation of the robot. The simulation is implemented in Simulink, in which the robot's motors are each represented as fifth-order transfer functions. The environment of the simulated robot consists of a gel block (to simulate "tissue") placed in the workspace (Figure 2.3). As the figure shows, the blue block simulates the

[1] The simulators were built by Mark Renfrew, Cenk Cavusoglu et al.

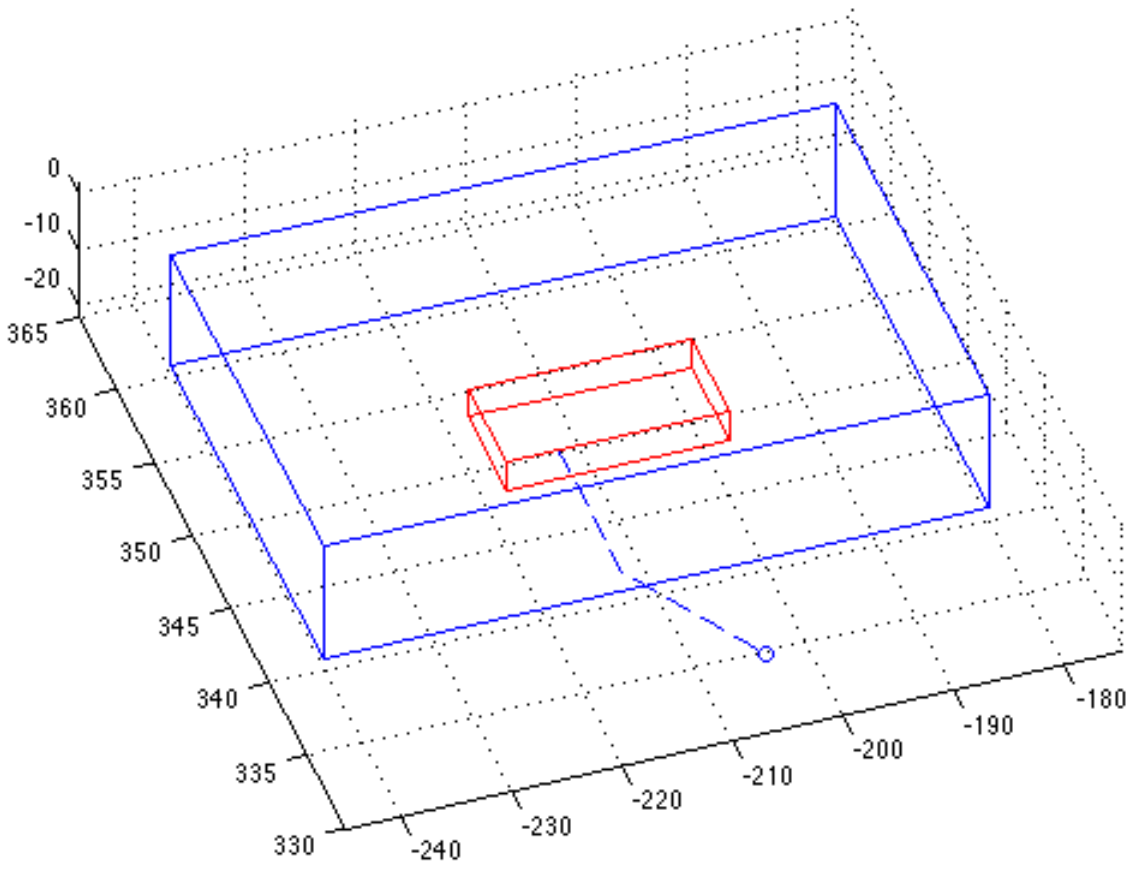


Figure 2.3 A normal trajectory (dashed line) in the simulation environment consisting of two "tissue" blocks (the red and blue cubes)

animal body and the red block simulates a vein inside the body. A stiff non-deformable needle is driven by a needle force model, which provides a resistive force cause by the combined frictional, cutting, and stiffness forces produced when the needle is inside the gel block. The cutting force is caused by the needle tip piercing the gel block and provides a resistance to the needle's motion during insertion into the gel block. The frictional force is produced by the friction between the needle body and the walls of the channel in the gel block, and resists the needle during insertion and extraction. The stiffness force is caused by the gel block's tendency to resist sideways motion of the needle, i.e., any motion not in the direction the needle is pointing. In this way, realistic and distinguishable forces can be produced by any possible motion of the needle.The needle model is described in detail in prior work [20].

After calibration, the robot's initial location is called its "home position." The controller can then be given a new position and orientation to move the needle to. If the new position can be reached, a "reference trajectory" is computed using linear interpolation. Velocities along this trajectory are set to be mainly constant, with fast acceleration and deceleration at the start and end (subject to a desired maximum acceleration). SABiR uses a two-layer motion control algorithm to follow this trajectory to guide the needle to the end point. The inner layer is a pole-placement based joint-level position controller operation at a sampling rate of 2 kHz . The outer layer is a Cartesian-space position controller operating at 100 HZ .

Currently, we use SABiR to do a *single insertion task*. The single insertion task for the robot is to insert the needle to a target position in the inner block(the red block), following a target orientation. Both target position and target orientation are specified by the user. In the task, the needle first moves to a "ready position" which is outside but near the surface of the blue gel block. After it arrives, the needle tip rotates to a specific inserting orientation, and starts insert. Then the controller drives the needle into the blocks until reaching the target position. Then the needle is extracted from the blocks to the ready position which is outside the block. Finally, the needle return to the "home position."

Three sets of parameters are used to specify a single insertion task: ready position, ready orientation (same as target orientation), and insert distance (distance between ready position and target position). Note that position and orientation are specified with 3 values, each correspond to one dimension in space. In Figure 2.3, we show the workspace and trajectory for normal single insertion task. The normal trajectory represents the system's normal behavior.

A supervisory software system is built on top of the low-level controller.[2] The user interacts with the robot through this software. This system has three components: a GUI, a task delegator, and a robot proxy. Figure 2.4 shows the information flow between the components when the robot performs a high level insert/extract needle operation.

For this work, we need to collect the data, both from software and hardware. The data flow here is as follows: the commands (i.e., the parameters specifying the target location and

[2]This part of work was done by Zhuofu Bai.

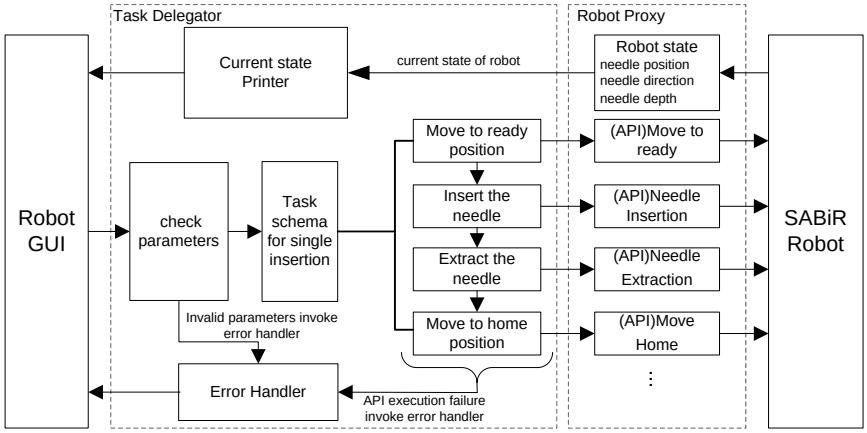


Figure 2.4 SABiR Software architecture

orientation) are passed by calling the application programming interfaces (APIs) provided by task delegator. The task delegator first checks the validity of input parameters for the specified operation; for example, it ensures that target locations are within the robot's workspace. It then decomposes a complex task into a set of basic needle motions that can be accomplished by calls to the API for the robot (in this case, simulator). The delegator is equipped with different schema to decompose different high-level tasks. It then invokes the robot API to handle these decomposed tasks. If an error occurs, it is responsible for stopping the current action and returning an error signal. When the robot proxy gets an API call, it communicates with the real robot (again, in this case, simulator), issuing low-level operations and collecting low-level sensor data from the robot (simulator).

For BHR, the control algorithm is the core of the robotic tools for tracking heart motion during coronary artery bypass graft surgery. During free beating, individual points on the heart move as much as 10 mm. Although the dominant mode of heart motion is in the order of 1-2 Hz, measured motion of individual points on the heart during normal beating exhibit significant energy at frequencies up to 26 Hz. When a predictive controller is used, the reference motion of the heart, which is provided to the feedforward path, is estimated by the controller.

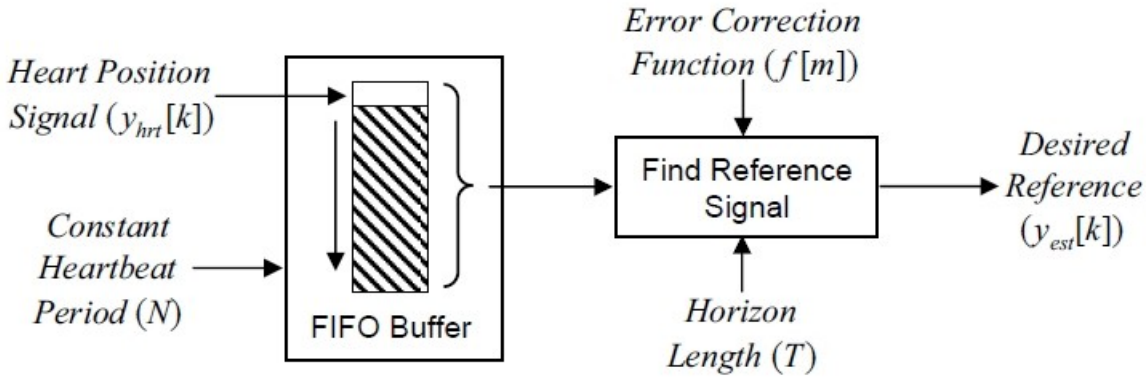


Figure 2.5 Reference Signal Estimation Block Diagram: Buffered Past Heart Position Data were used for estimation with approximated constant heartbeat period [3].

The motion of the heart from the previous cycle was used as a prediction of the next cycle (Figure 2.5). Then BHR could calculate the next position according to the reference heart signals and the current heart feedback. The control architecture is shown in Figure 2.6 [21, 3]. In the figure, sensors collect the blood pressure and electrocardiogram directly from the heart. Then they send the data to multi-sensor information fusion, which combines all the information and estimates the reference trajectory. Finally, the robot motion control block will drive the robot by following the reference estimation.

We use models for the kinematics and inverse kinematics developed in our prior work [3] to create a simulation of the robot, implemented in MATLAB Simulink. In the simulations, the dynamics of each of the robot's degrees of freedom are represented as sixth-order transfer functions. The environment of this robot consists of simulated heart and breathing motions, which the robot arm needs to follow. The BHR system uses a Receding Horizon Model Predictive Control based robotic motion control algorithm. As part of the motion control algorithm, the nonlinearities of the system (i.e., gravitational effects, joint frictions, and Coriolis and centrifugal forces) are also canceled. The controller has a sampling rate of 2 kHz .

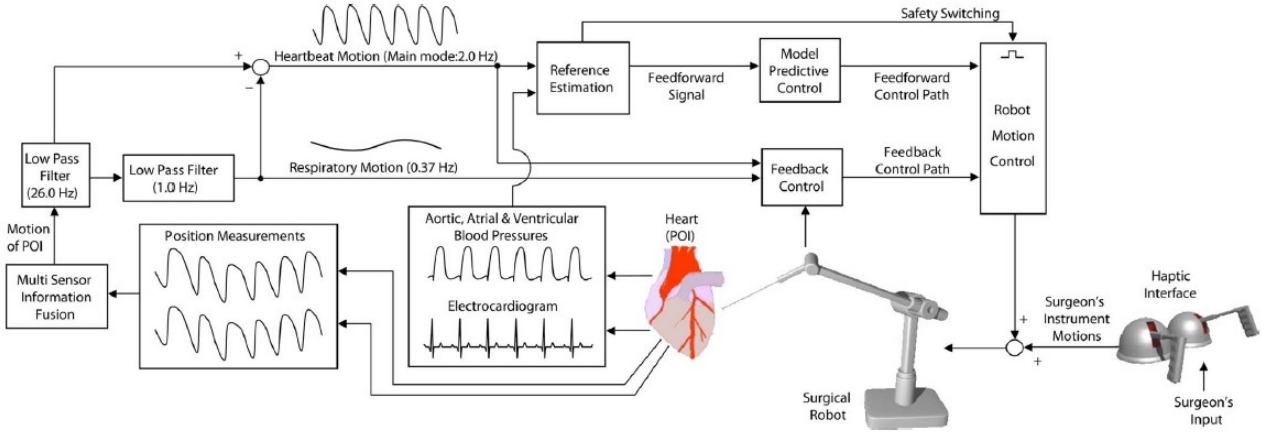


Figure 2.6 Control architecture for designing intelligent control algorithms for Active Relative Motion Canceling on the beating heart surgery

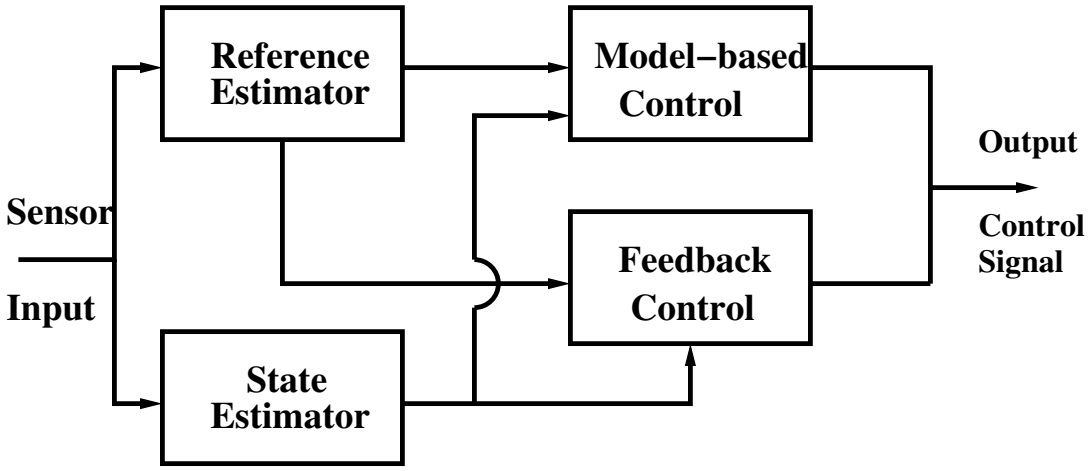


Figure 2.7 Block Diagram of a Typical Controller.

2.2.4 Software Characteristics of the Controllers

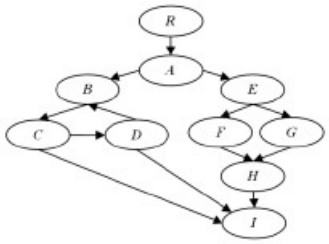
In Figure 2.7 we show a general block diagram of control software [22]. The controller typically will first estimate the current state of the system using the available sensor inputs (e.g., sensed heart and respiratory motion for BHR). In some cases, a reference trajectory may be available, in other cases it may be estimated (e.g., BHR predicts based on observed heart behavior the motion of the heart for the next time periods and uses that to construct a reference trajectory). Then it will generate an error signal that is proportional to how far away the system

currently is from where it needs to be, according to the reference trajectory. This can also take into account a model of the underlying environment, if it exists. Then it will compute the output signals that need to be fed to the actuators to reduce this error and output this to the hardware or simulation. This entire process is repeated at the rate at which samples are collected from the environment, which is 2 kHz in our case. This means that the entire control code has to execute in less than 500 μs . This sort of real-time constraint is also quite typical for such systems.

As a result of the runtime constraint, low-level control system software tends to be compact in terms of lines of code. The SABiR controller has 330 lines of statements in 6 functions, while the BHR controller has 167 lines of statements in 9 functions. It is important to note however that the size here does not reflect complexity; it should be evident from the descriptions above that each controller is carrying out a complex sequence of numerical calculations in order to work. A second point to note is that these controllers are written in MATLAB (a common choice in such cases). This means that many of these statements perform complex operations on entire matrices at once, to take advantage of the speedup provided by vectorization in MATLAB. Finally, the code has very few branch points: the SABiR controller has 11 branches while the BHR controller has no branches at all. This is also quite common in control system software [23, 24] because the core algorithm is about the same (as described above) in most cases, and as can be seen from Figure 2.7, is essentially a linear flow of operations. It is generally only the specific mathematical details of the system and the control algorithm that varies. As an operational benefit, reducing branches speeds up the execution time. However, this fact creates problems for coverage based fault localizers, as we show in our evaluation.

2.3 Coverage-Based Fault Localization

Debugging software is an expensive and mostly manual process. Among all debugging activities, locating faults is the most expensive. Because of the high cost, any improvement in the process of finding faults can significantly decrease the cost of debugging. In practice, it is common that software developers manually localize faults in their program. Usually, the fault



Test #1 (t_1):
$R \Rightarrow A \Rightarrow B \Rightarrow C \Rightarrow D \Rightarrow B \Rightarrow C \Rightarrow D \Leftarrow \Leftarrow \Leftarrow \Leftarrow \Leftarrow \Leftarrow$

Test #2 (t_2):
$R \Rightarrow A \Rightarrow E \Rightarrow G \Rightarrow H \Rightarrow I \Leftarrow \Leftarrow \Leftarrow \Leftarrow \Leftarrow$

Test #3 (t_3):
$R \Rightarrow A \Rightarrow E \Rightarrow F \Rightarrow H \Rightarrow I \Leftarrow \Leftarrow \Leftarrow \Leftarrow \Leftarrow \Rightarrow B \Rightarrow C \Rightarrow D \Leftarrow \Leftarrow \Leftarrow \Leftarrow$

Test #4 (t_4):
$R \Rightarrow A \Rightarrow E \Rightarrow G \Leftarrow \Leftarrow \Leftarrow$

Test #5 (t_5):
$R \Rightarrow A \Rightarrow B \Rightarrow C \Leftarrow \Leftarrow \Leftarrow$

	t_1	t_2	t_3	t_4	t_5
R	1	1	1	1	1
A	1	1	1	1	1
B	1	0	1	0	1
C	1	0	1	0	1
D	1	0	1	0	0
E	0	1	1	1	0
F	0	0	1	0	0
G	0	1	0	1	0
H	0	1	1	0	0
I	0	1	1	0	0
Y	0	0	1	0	0

Figure 2.8 Motivating example [4]. Left: Static Call Graph. Mid: Method Call Sequences. Right: Test Case Coverage Matrix

localization process begins with a failure observation when the developers run the program. The testers then choose a failed test case to run, and place breakpoints iteratively in each section of the program until they find an erroneous state. For large program, this process could take a lot of time.

Inheriting the logic of manual fault localization [25, 26], prior work has developed a coverage-based technique to help developers find faults in the program. Under each software test, testers gather a large amount of data about a software system. The fault localization techniques can be based on the data which contains the information about statements being covered or not being covered, or how often they are covered. A covered statement means that the statement has been executed by a test case. Combined with failure data, a suspiciousness score can be calculated for each statement.

As a standard testing technique, the test coverage is documented as a simple matrix called a test coverage matrix. In this test coverage matrix the test implementation and operations are mentioned in the column and the test cases are mentioned in the row (Figure 2.8, right). Coverage-based fault localization techniques [4] use the information from the coverage matrix and calculate a score for all the statements. Statements are ranked with respect to the score in descending order. In fault localization, whether or not a test case causes a program statement to be covered depends entirely on how the test case affects the branching behavior within the program‘s control structure; it is generally impossible to pick and choose exactly which statements are covered. Here, we describe two coverage-based approaches below.

2.3.1 PFiC

The simplest and most straightforward way of estimating the faultiness of a statement based on the coverage matrix is to estimate the conditional probability of faulty for each statement based on test coverage counts. This is the "Probability of Failure if Covered" or PFiC [26], which is estimated using Equation 2.1:

$$PFiC(s) = N_{fail}(s)/(N_{fail}(s) + N_{success}(s)) \quad [2.1]$$

Since the number of test cases covering s is the dominator in the formula, this equation is only defined on statements that are covered at least once by a test suite.

2.3.2 Ochiai

Ochiai [27], which modifies the PFiC by using the information of the number of passing tests that do not execute the statement we want to estimate, is a relatively new method compared with PFiC. The formula for estimating the score is in Equation 2.2:

$$Ochiai(s) = \frac{N_{fail\&covered}}{\sqrt{(N_{fail\&covered} + N_{fail\&\neg covered}) * (N_{fail\&covered} + N_{success\&covered})}} \quad [2.2]$$

Here s is the the statement in which we are interested.

2.4 Other Related Work

Many techniques have been developed for fault localization, which has gained increasing attention in recent years. The most closely related work [28], which uses probabilistic models to identify bugs in programs, is proposed by Liu et al. The SOBER method in that work builds a probabilistic model for program predicates. If the probability that a predicate P evaluates to true in passing runs is significantly different from the probability that P evaluates to true in failing runs, P is judged to be related to program failure. SOBER models control dependences but not data dependences. Baah et al. propose the probabilistic program dependence graph

(PPDG) [29], which is a probabilistic graphical model based on the program dependence graph (PDG). A PDG is basically a directed graph that captures the control dependences and data dependences between each element of the program. The element could be a statement or a set of statements that constitute a node. Using a PPDG, we could infer the state of a node given the states of its parent nodes. By comparing the behavior of nodes in passing runs with those of failing runs, a score is estimated and a rank is obtained for fault localization.

Other work which focuses on the safety of medical robotic systems in the literature primarily focus on design of intrinsically safe systems, e.g., [7, 8, 9, 10, 11]. For example, a related approach in hybrid systems is parameter synthesis [30]. Parameter synthesis basically perfectly designs the system parameters, such as joint limits, power levels, mass, to produce a well-defined system so that the system operates normally or always fail in a safe manner. This is typically achieved by using actuators with limited power and speed, current limiters, etc. However, these approaches do not consider the software development process, and do not attempt to help the developer to improve the software controller. In contrast, our model could not only detect and predict abnormal behavior of the robot, but also provide a fault localization technique to aid the development of the controller.

Chapter 3

Fault Localization Algorithm

In this chapter, we describe the variables we use to describe the system state for both robots. Then we present the augmented probabilistic model which is used to model the behavior of robotic system. Based on this model, we propose a new technique which can precisely localize faults in the embedded control system. We call our approach Fault Localization in Embedded Control Systems (FLECS).

3.1 State Variables

There are three types of variables that are parts of the system state. They are parameters, hardware variables, and software variables. Variables such as the reference trajectory to be followed by the robot are "parameters." These variables are inputs to the system and do not change over time. Since they are constant, we only use them to help predict other variables in our model, and they are not themselves predicted. Other variables such as the x position of the needle tip in SABiR or x end position in BHR are "hardware variables." The values for these variables are obtained directly from the sensors on the robot. The last type of variables are "software variables." They include variables which cannot be directly sensed in the hardware but can be estimated or derived in software from other variables, such as "force on the needle." Also, these derived variables can estimate other variables and all of them are included in "software variables." All these variables are shown in Table 3.1 and Table 3.2. For each kind of variable, "No." refers to the number of variables of that kind. For example, there are 3 needle forces, one for each direction in SABiR. In practice, not all the variables can be monitored in

Table 3.1 SABiR: State variables in DBNs. First group: parameters, second group: software, third group: hardware. Parameters are used to predict other variables but not predicted themselves. The "No." column shows how many variables of each type there are

Description	No.
Reference position, starting position, end position	18
Position and orientation where insertion begins	6
Insert distance into tissue from ready position	1
High level action, e.g., "Insert Needle"	2
Estimated depth of the needle inside tissue, using estimated forces	1
Estimated depth of the needle inside tissue, assuming fixed tissue geometry	1
Estimated force on the needle	3
Estimated torque on the needle	3
Computation results in controller software	87
Position and orientation of needle tip	6
Positions of 5 motors	5
Torques of 5 motors	5
Error between Actual Position and Reference Position	5
Speeds of 5 motors	5
Error between Actual Speed and Reference Speed	5

the embedded control system. In Section 4.1, we outline a heuristic technique that we use to select the set of software variables described in the tables.

3.2 Augmented Dynamic Bayesian Network

As we describe in the background, we create a simulator to generate data for both of the robots. Based on the collected data, we use dynamic Bayesian networks (DBNs) [31] to model the time-evolution of the state space of the system. These are first order Markov models that represent the probability of the next state given the current one, i.e $Pr(S_{t+1}|S_t)$,where each S_t

Table 3.2 BHR: State variables in DBNs. First group: software, second group: hardware. The "No." column shows how many variables of each type there are

Description	No.
Reference position	3
Position error	3
Radian error	3
End effect reference position	3
End effect error	5
Torque reference	18
Gravity parameter	2
Breath parameter	8
Computation results in controller software	34
BHR actual position	3
Actual end position	3

is described by a vector of variables. The factored state transition distribution is defined by the equation below:

$$Pr(S_{t+1}|S_t) = Pr(\mathbf{V}_{t+1}|\mathbf{V}_t) = \prod_{i=1}^{n} Pr(V_{t+1}^i|\mathbf{V}_t) = \prod_{i=1}^{n} Pr(V_{t+1}^i|\mathbf{V}_t^{par(i)}) \qquad [3.1]$$

where $\mathbf{V}_t = \{V_t^i\}$ denotes all of the variables at time t, $\mathbf{V}_t^{par(i)}$ denotes all of the variables at time t that have an edge to V_{t+1}^i. We list all the variables which are used to construct DBNs in Table 3.1 and Table 3.2.

We show an example of our DBN in Figure 3.1. Each node in the figure is associated with a Conditional Probability Distribution (CPD), denoted as $Pr(V_{t+1}^i|\mathbf{V}_t^{par(V)})$, which is the probability distribution of V^i given all the nodes in the previous time step that have an edge to it. From Equation 3.1, we know that the probability of the current state given the previous state equals the product of the probability of each variable in the current state given relevant variables in the previous state.

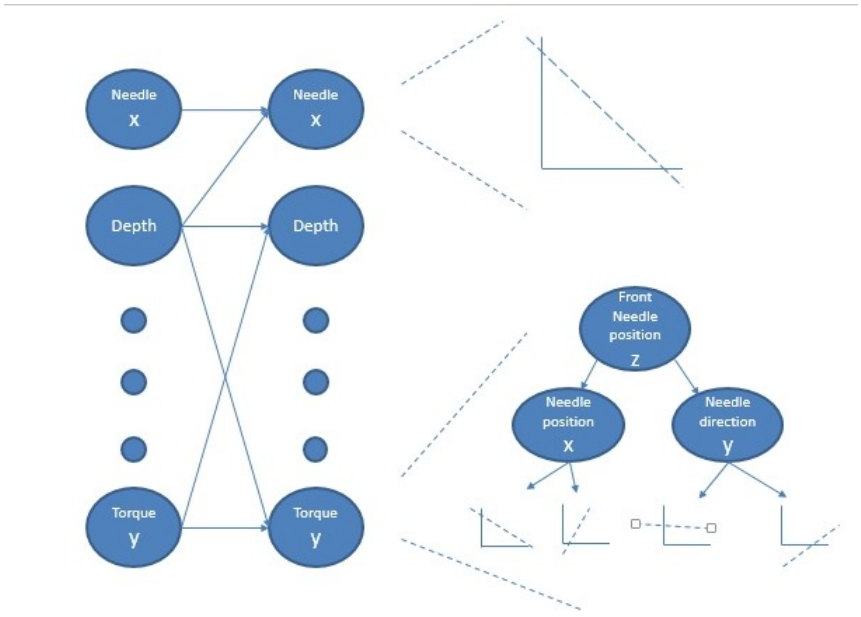


Figure 3.1 Example Dynamic Bayesian Network. The structure identifies influences between variables at successive time steps. Within a node at $t+1$ is a CPD describing the distribution over each variable conditioned on its parents at time t.

Usually, the CPDs of each state variable are unknown a priori. We estimate CPDs from observed trajectories of hardware/software states. There are many ways to represent a CPD. If all the relevant variables are discrete, then one could store the CPD as a big table explicitly. However this is not always possible, since the size of table grows exponentially as the number of parents increases. Also, when the variables are continuous, we cannot store all the probabilities in a table. Thus, alternatively, one could use a parametric or nonparametric model to approximate the CPD. In the following two subsections, we will briefly go over two models we used to represent the CPDs in our approaches.

3.2.1 Linear Gaussian Model

In the state space, we choose a *linear Gaussian model* to model some state variables because they vary at a roughly linear rate from t to $t+1$. For example, the x position of the needle tip for SABiR is one of those variables; because in this case, the robot controllers are designed to maintain a constant velocity in each high level action. A linear Gaussian model

describes the relationship between a dependent variable Y and a set of independent variables $\mathbf{X}$. It assumes that the conditional distribution of Y given $\mathbf{X}$ is a Gaussian distribution, with a mean that depends linearly on $\mathbf{X}$:

$$Y|\mathbf{X}; \mathbf{w}, \sigma_{\mathbf{X}} \sim \mathcal{N}(\mathbf{w}^T \cdot \mathbf{X}, \sigma_X^2) \quad [3.2]$$

Given a set of training data, i.e. $(\mathbf{x}, y)$ pairs, we could find the parameters $\mathbf{w}$ and $\sigma_{\mathbf{X}}$ so that the likelihood of training data in the model is maximum:

$$\mathbf{w}^*, {\sigma_{\mathbf{X}}}^* = arg \max_{\mathbf{w}, \sigma_{\mathbf{X}}} \prod_i \Pr(y_i|\mathbf{x}_i; \mathbf{w}, \sigma_{\mathbf{X}}) \quad [3.3]$$

The optimization step for this objective function is the training procedure for linear Gaussian model. After we find a set of $\mathbf{w}, \sigma_{\mathbf{X}}$ we calculate the CPDs for linear variables.

3.2.2 Regression Tree models

For other nonlinear variables, we use a regression tree model for the CPD. A *regression tree* is one type of decision tree that predicts the value of a target variable based on several input variables. A regression tree handles the situation when the target variable is continuous. A regression tree partitions the whole input data into different disjoint groups, each of which corresponds to a leaf node. Each internal node of the regression tree is a test on some variable at the previous time step. According to these tests, data are split into the children of the node until they hit the leaf node. For instance, in Figure 3.1, the node denotes a test on variable called $frontneedlepositionz$, if $z > 300$, data goes to the left node, otherwise it goes to the right node. Each leaf node is again a linear Gaussian model. One of the significant advantages in our training procedure is that we can query new data from simulator when there is not enough data for regression after we split the original into two nodes. Regression tree models are a very general, nonparametric representation of nonlinear dynamics; they can be expected to work well for a variety of control systems.

Both structures and parameters of the DBNs are learned from data. We generate sequences of normal trajectories from our simulations to form the data. From those trajectories, we estimate the CPDs for the variables using maximum likelihood; for linear Gaussian models, this procedure is equivalent to linear regression and yields closed form solutions. For regression tree models, a standard greedy top-down recursive decomposition approach [32] is applied, and the goodness of a split is computed by the improvement in the r^2 measure, which is defined as follows:

$$r^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \quad [3.4]$$

where y_i is the true value of the i^{th} input data point, $\hat{y}_i$ is the estimated value of y_i from model, and $\bar{y}$ is mean of y_i. At each node, the goodness of each test is calculated in the following way:

$$r^2_{split} = \sum_i \frac{N_i}{N} * r_i^2 \quad [3.5]$$

where N is number of data points in the current node, N_i is the number of data points that go to i_{th} branch after split, r_i^2 is r^2 of model for i^{th} branch. The split with the largest improvement in r^2 will be used. If no such split exists, the current node will become the leaf, and all the data remained in this node will be used to train a linear Gaussian model.

In prior work [33], we have shown that this model and the training procedure yields very accurate models of normal hardware/software dynamics for the SABiR system. We argue that the generality of the model makes it equally suitable for application to other control systems as well.

3.2.3 Feature Selection

Since embedded systems are engineered systems, they are designed to be sparse, i.e. most state variables tend not to depend on too many other variables. Therefore, we apply a modified version of Sparse Candidate feature selection algorithm [34] to limit the number of parents for

Algorithm 1 Feature Selection Algorithm

Require: A set of search data D_1, A set of data D_2 for validation, Prior parent set $Pa(v)$, Decomposable score BIC, Maximum size of candidate variable set M, Variable v to predict, Maximum iteration K, A set of variables V used to predict v

Ensure: A parent variable set $Pa(v)$

for $n = 1 : K$ **do**
  $V \leftarrow V$.remove($Pa(v)$), $Mu =$ new Array
  Restrict
  for each x in V **do**
    $MIC(v, x|Pa(v)) \leftarrow D_1$
    Mu.add(MIC(v, x —Pa(v)))
  end for
  Sort Mu in a decreasing order
  S = size($Pa(v)$), C = new Array, BIC = new Array
  $C \leftarrow$ add corresponding variable from (Mu[$1 : M - S$])
  Maximize
  for i = 1, 2, 3, $\ldots M - S$ **do**
    $Pa(v)$.add($C(i)$)
    $DN_i \leftarrow$ train | Pa_v, D_1
    $BIC(i) \leftarrow BIC(DN_i, D_2)$
    $Pa(v)$.remove(C_i)
  end for
  Sort BIC in a decreasing order
  $x \leftarrow BIC$[1]
  $Pa(v)$.add(x)
end for
return $Pa(v)$

each variable when we are learning the regression tree/linear regression CPDs. The feature selection is shown in Algorithm 1. The algorithm has two fundamental parts: **Restrict** Step and **Maximize** Step.

In the **Restrict** Step, we select a set of variables from the variable pool as the candidate variables set. It gives us a smaller search space in which we can hope to find a good structure quickly. In order to obtain the candidate set, we first select the variable we want to predict, for example v in Algorithm 1. Then we compare the **conditional mutual information** [35] between each variable in the set which we use to predict other variables and v given the parent set of v. The conditional mutual information is defined in following equation:

$$\mathbf{MIC}(\mathbf{v};\mathbf{x}|\mathbf{Pa}(\mathbf{v})) = H(v,x) + H(v,Pa(v)) - H(v,x,Pa(v)) - H(Pa(v)). \qquad [3.6]$$

Here $H(v,x)$, $H(v,Pa(v))$, $H(v,x,Pa(v))$, $H(Pa(v))$ are the joint entropies. A joint entropy is basically a measure of the uncertainty of a set of variables. It is the sum of the product of the joint probability and the log of the joint probability over all the associated variables. We use Equation 3.7 to calculate the joint entropy in Equation 3.6.

$$\mathbf{H}(\mathbf{v},\mathbf{x}) = -\int_v \int_x P(v,x)\log_2[P(v,x)]\,dx\,dv. \qquad [3.7]$$

In the entropy equation, $P(v,x)$ is the joint probability density function of v and x. For each variable, we consider they are obeying a Gaussian distribution. Hence $P(v,x)$ is a multivariate Gaussian distribution. Given the training data, we calculate $P(v,x)$ and use it to calculate joint entropy and the conditional mutual information.

The mutual information basically measures the dependencies between two variables. In this approach, it will always choose the most dependent variable and add it into the candidate set. It is always non-negative and it is equal to 0 when v and x are independent. The higher the mutual information, the stronger the dependence between v and x. Therefore, we rank the conditional mutual information for all the variables and pick the top $M-S$ (M is the maximum

size of parent set we define and S is the size of the current parent set) variables and consider them as a candidate parent set of variable v.

After we obtain a candidate set C, we search in this set. For each variable in the candidate set, we add them into the parent set. Suppose the size of C is S, we then have $M - S$ different parent sets for variable v. For each parent set, we train a DBN Dn_i and use it to calculate the **Bayesian Information Criterion** (BIC) [36], which is defined in Equation 3.8 below:

$$\mathbf{BIC_B} = -2 \ln L + k(\ln(n) + \ln(2\pi)). \quad [3.8]$$

BIC is a criterion for model selection in a finite set of models. It estimates which network is better at fitting the data. In the equation, L is the maximized value of the likelihood function of the model, n is the sample size and k is the number of free parameters to be estimated. In our model, the k is the number of the regressors in the Linear Gaussian model. We use the trained DBN to calculate the likelihood. Because we need a rank of the BIC, the π term is ignored in calculation.

As long as we get the maximum BIC value, we will store the corresponding network as the newest updated structure. If none of the candidate variables can provide a higher BIC for the structure, we just keep the current network and do not update it. Most of the time when we select a candidate set, there are some variables we could use to improve the structure.

Because we are able to access the inner structure of the controller code, it is possible for us to select a set of variables as the prior information as the parent set of each predictable variable, which means we could replace the empty initial network with a prior connected structure in the beginning of the algorithm. We first build the program dependency graph (PDG), which is a directed graph that represents dependencies of all the statements inside the program. For each predictable variable, we select a set of variables as the candidate parents for feature selection by tracing back through the PDG. This procedure means that the set of S candidate parents for a variable will be the S closest ancestors of the variable in the PDG obtained by following data dependence edges. The assumption here is quite intuitive—those closest ancestors have the most effect on the value of corresponding variable.

We list a set of experiments in Section 4.3 to demonstrate that the augmented DBN is able to model normal behavior with high accuracy in SABiR. We hypothesize that they are suitable for application to other control systems as well. We describe how we use our model to localize bugs in the software control code in next two sections.

3.3 Identifying Anomalous Variables

The goal of this project is to localize software bugs in embedded control system. Given our DBN, we are able to find the number of specific abnormal variables in each step by checking the likelihood of the variables. In order to find the buggy statements, we look at trajectories containing A&A events. We label the trajectories as follows: the start point of an anomalous trajectory is identified through human observation or through automatic detection of the trajectory deviating from the reference trajectory by a significant amount. This is analogous to the pass/fail labeling that must be done for test cases in software testing; in simple test cases this can be automated, while for programs producing complex outputs it may require manual inspection.

Algorithm 2 shows the algorithm in this part of our work. First, suppose the start point we identified is denoted by index T. Using the values of the variables in the state at $T-1$, s_{T-1}, and the DBN nodes we have, we can calculate the likelihood of s_T given state s_{T-1}, i.e. $Pr(s_T|s_{T-1})$. In this case, $Pr(s_T|s_{T-1})$ will be a Gaussian distribution according to our model choice. Since this point is an anomalous point, the state s_T is expected to have low likelihood. Each state is composed of a vector of variables. Some of the variables' value in state s_T are expected to have a low likelihood in our model, which will happen if these variables' values deviate from their "expected" values (the mean of the linear Gaussian distribution in our model). In other words, using the DBNs, we can identify the potential causes of the anomalous event as low-likelihood variables in the DBN. Therefore, if we pick the last few variables after we rank all variables in s_T by likelihood, they should be good candidates for causing the anomalous behavior.

To decide which are anomalous variables given all the likelihoods, we need to set a threshold to select low likelihood variables. To do this, we maintain a range of normal likelihoods which we calculate from the training data that was used to train DBNs. Given the training data, which are all normal data points, we go through each pair of points and calculate the negative log likelihood (NLL) for all the variables. The negative log likelihood for a state s_{t+1} given its previous state s_t is computed as follows:

$$NLL(s_{t+1}|s_t) = \sum_{i \in \{Pred\}} \frac{(v_i^{pred}(t+1) - v_i(t+1))^2}{Var(i)} \quad [3.9]$$

where $Pred$ is the set of indices of predictable variables in the DBN, $v_i^{pred}(t+1)$ is the predicted value for variable i at time $t+1$, $v_i(t+1)$ is the real observed value for variable i at time $t+1$. $Var(i)$ is the variance of the CPD (in our case either regression tree model or linear Gaussian model) over training data. Then we pick the highest NLL among all the normal data points as the threshold. Whenever the variable's NLL is larger than the threshold, we consider it to be an anomalous variable. Using this approach, we can pick a set of anomalous variables.

Using this outside-normal-range criterion, it is possible that we get an empty anomalous variable set at time T. This could happen when there is a bug that results in a small error which will be accumulating over time, and at the starting point, the likelihood is still outside of the range. In this scenario, we will check the state s_{T+1}, s_{T+2}, in turn until we find a state which give us a non-empty set of variables.

3.4 Identifying Faulty Statements

After we get a set of anomalous variables which are identified by the algorithm above, what we need next is to translate those variables into controller code to locate the faults. To do this, we choose the controller's dynamic PDG, which was induced by traversing the PDG from s_{T-1}, a normal input, to s_T, an anomalous output. In the PDG, we first locate the statements which assign a value to the anomalous variables we obtain above. Then, we identify a candidate statement p as the potentially faulty statement if p can "explain" the statements containing

Algorithm 2 Fault Localization in Embedded Control Systems

Require: The software code of control system, simulator, trajectories with both normal and abnormal behavior

Ensure: Ranked list of statements

$D \leftarrow$ learned DBN from normal trajectories obtained using simulator (Section 3.2)
$T \leftarrow$ time index of start of anomalous event
$t = T$; $BadVars \leftarrow \phi$
while $BadVars == \phi$ **do**
    $BadVars \leftarrow$ list of low-likelihood variables at t according to D
    $t \leftarrow t + 1$
end while
$P \leftarrow$ dynamic PDG of controller at T
$O \leftarrow$ statements outputting $BadVars$ in P
$Rank \leftarrow$ zero-element array of size number of nodes in P
for p a node in P **do**
    for o a statement in O **do**
        if p is an ancestor of o in P **then**
            $Rank(p) = Rank(p)+$ distance in P from p to o
        else
            $Rank(p) = Rank(p) + 50$ {Here we assume 50 is longer than all paths in P.}
        end if
    end for
end for
return statements in P sorted by $Rank$

the variables in the anomalous variable set. In this setting, a statement p can "explain" another statement o if p is the ancestor of o in the PDG. In order to locate the faults in the controller code precisely, we assume that the closer a statement is in the PDG to a statement with an anomalous variable, the better it "explains" that anomalous variable. Thus the closer a statement is to *all* anomalous variables, the more likely it is a faulty statement. Under this assumption, for each statement p in the controller PDG, we sum the distance between p and all the suspicious statements in that set. In this setting, if statement p directly influence statement o, the distance between p and o is 1. If p is the parent i and i is the parent of o, then the distance between p and o is 2. If p is not an ancestor of o, we add a constant to its sum that is larger than the largest path length in PDG. This is because when we compare the distances of the statements, a longer distance in the PDG means that the parent statement is likely to affect many other variables, which were not reported to be anomalous. In this sense, the faultiness of the parent statement is more weakly determined by the child. If p is not the ancestor of o, obviously the distance between o and p should be larger than any distances between o and o's ancestors. Following this rule, we can calculate distance between the statement that contains the anomalous variable and all the statements in the controller code. For each statement, we sum the distances from the all statements containing the variables in the anomalous variable set as a total distance. Then we rank the total distance and return a ranked list. If the assumption we made above is right, a statement p which is the closest common ancestor of all the statements o will in the top of the ranked list, especially when the bugs are rare.

Note that because the controllers are written in MATLAB, a statement could return a matrix or a vector of multiple variables as an output. In our implementation, some of those statements are represented by multiple variables in PDG, for example, the statement $Front = T\_BackFront\_mount \times pf\_front$. Statement variable $Front$ is consists of three variables—$actual\ position\ x$, $actual\ position\ y$ and $actual\ position\ z$. Therefore, the total number of the nodes in PDG may be larger than the total number of the lines of code in controller. The advantage of this implementation is that if later statements use different elements of this matrix in their computations, they could have different dependencies according to which values they

use in that matrix. For instances, in the computation of a vector a, if $a(1)$ is computed using variable b and $a(2)$ is calculated from c, and a statement o is using the $a(2)$ as an input to compute variable x. Then o will have direct dependency on c instead of depend on both c and b.

With the ranked list, we assume that a developer will start debugging by tracing the statements from the top of the list. Thus the earlier in the ranked list faulty statements are, the easier it is to debug the controller and the better our approach is. In the next section we perform experiments to determine how highly faulty statements are ranked by our approach in the SABiR and BHR controllers.

Chapter 4

Empirical Evaluation and Discussion

In this section, we first show that the augmented DBN works well in both of the robots. Then we evaluate the algorithm we described previously on both of the robotic systems we have. We also compare our approach with PFiC and Ochiai, which are elaborated in the related work section. We demonstrate that our method can localize the faults much more precisely than the baselines on our testbed systems.

4.1 Instrumenting the Controllers

In our case, the controller accepts the reference data and calculates the specific hardware and software variables for actual movement. In the end, it passes all the variables to the environment model and robot model to move the robot. In our approach, we record the final values of variables after the controller finishes processing in each time step. Due to the complexity of our systems, it is not possible to record all the variables' values in the controller program. One obvious reason is that many variables in the program are large matrices or vectors. In those matrices, we did not record all the elements in each matrix, otherwise the data size becomes very large. Therefore, to keep the data collected reasonable and keep our DBNs' training time tractable, 100 or fewer variables from each controller are selected as the software variables which represent the characteristics of robots' software. In this work, we use two heuristics to select the variables to monitor.

First, if an output variable of a statement is a matrix (e.g. $X = A * B$), a candidate for monitoring is a single, randomly chosen element from the output variable. If this statement

($X = A * B$) contains a bug, most of the elements in X will be affected and will themselves affect the variables that use X to do calculation as a matrix. However, if one of the descendants of the statement uses a specific element from the matrix (e.g., $X(2,2) = A(2,3) * B(3,2)$), this element will be a candidate for us to monitor.

Secondly, if a variable appears as a common ancestor of other variables, which means that this variable is used by many other variables, we select it as a candidate to monitor. In the controller's PDG, they are the "hub" variables and influence many other variables' values. We select those variables that influence at least five other variables in the program.

Following the heuristics above, we instrument 87 software variables from the controller of SABiR and 59 variables from the controller of BHR. The values of these variables are collected at every time step before the controller starts the next time step, and are used to generate training and testing data.

For the coverage-based techniques, we also instrument the controller to obtain coverage data. Since there are few branches, we simply track which branches are taken on any given run, and reconstruct the coverage matrix from this information.

4.2 Mutation-based Generation of Faulty Controllers

To evaluate the techniques, we need buggy versions of the controllers. During testing, we discovered that one bug occurs during the SABiR simulator's normal operation. This bug causes a subsequent square root computation to return a complex number and only the real part is stored and processed. This problem causes the robot to move the needle out of the workspace in the simulator. Figure 4.1 shows the deviation between actual trajectory and reference trajectory in this bug. In order to generate additional bugs, we apply mutation testing. While the mutation program is running, it keeps generating faulty versions of the controllers (mutants) through the randomized modification of elements of the code. Although this method has been widely used in prior work [37], a big challenge in our case is that our controllers are in MATLAB. Some limited prior work has used Simulink. Zhan et al. mutated the input and output parameters of Simulink blocks [38]. Brillout et al. generated faulty versions of

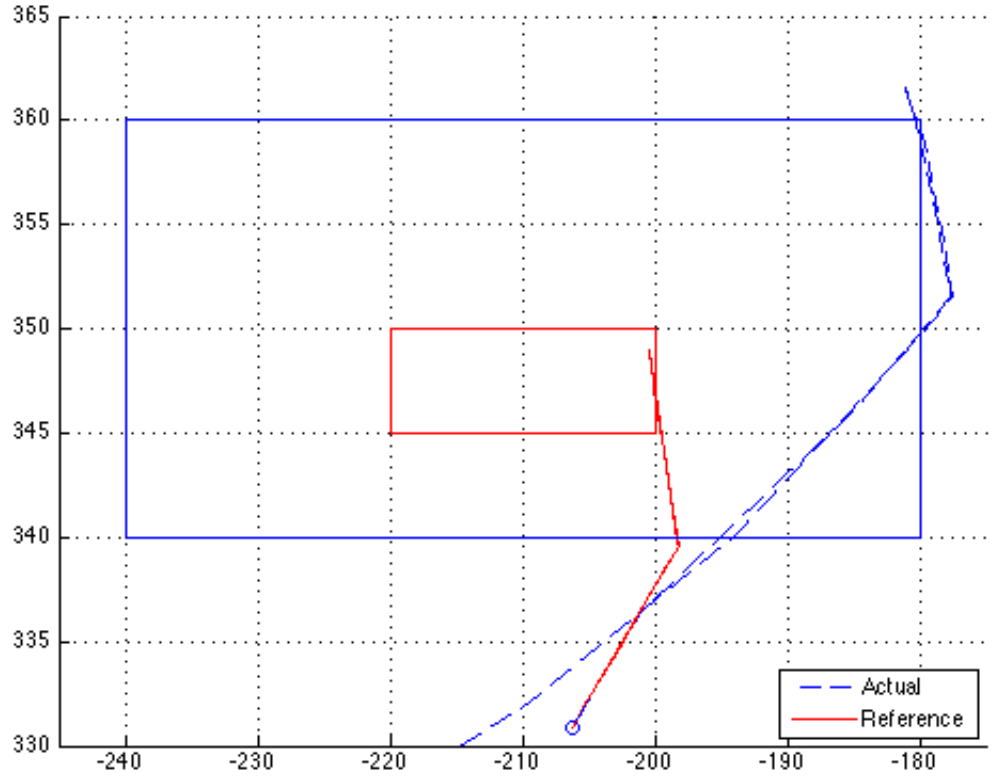


Figure 4.1 Out-of-Workspace Event

Simulink diagrams by altering blocks or removing constraints [39]. However, in this case, our controllers are functions within Simulink in MATLAB, and no prior work has injected faults into MATLAB functions.

To solve this, we converted the MATLAB code into Java and used a tool[1] to automatically generate faults from Java code. Then we recorded the types and locations of the faults and manually injected the same faults into the corresponding MATLAB function in the Simulink model. In the Java mutation tool, we first create four mutation operators: replace numerical constant, negate jump condition, change arithmetic operator, and add/subtract a small numeric value. The procedure of mutation has three steps: first, a Java parser [40] visited the source

[1]This part of the work was done by my colleague Zhuofu Bai.

code of the controller program and recorded all expressions that could apply those four operations such as condition expressions, assignment expressions and binary expressions. After randomly selecting a mutation operator, program will randomly chose an expression that could fit the operator selected previously and generate a fault for the chosen expression. Then we manually inject this fault into the MATLAB controller code to check whether it causes the simulator to produce an abnormal behavior in the hardware without crashing. If so, we store this fault and generate trajectories. Through this process, we generate nine mutants for SABiR and ten mutants for BHR. In the following section, we evaluate our technique on these buggy versions of controllers.

4.3 Modeling Normal Behavior

Prior work has shown that unimproved DBNs work can work well in modeling SABiR system dynamics. Because the augmented DBNs' training procedure includes feature selection and prior structure, the parent size of each variable has been restricted to be at most 5. Comparing to the unimproved DBNs, whose variables' parent set's size is 59, our DBNs are sparse. We hypothesize that the sparse DBNs can also model the systems' behavior well. In order to prove that, we did the experiment below.

In this experiment, we train the augmented DBN and evaluate them on both robots. We build DBN models with normal trajectories. We use 400 normal trajectories to train our DBNs for SABiR and 10 normal trajectories for BHR. Each SABiR trajectory has about 25,000 data points and each BHR trajectory has 110,000 data points. Then we randomly sample $20,000$ (s_t, s_{t+1}) test pairs from another 400 normal trajectories. Using the s_t values, we then estimate predicted state s_{t+1} and compute an r^2 value, which is described in Section 3.2.2, to measure the accuracy of these predictions.

Results in Table 4.1 shows the average r^2 value over either all the software or hardware variables in SABiR and BHR. From these results, we observe that the augmented DBNs are quite good at modeling the time-evolution of the both hardware and software variables. These results show that feature selection can successfully produce DBNs which are sparse but still accurately

Table 4.1 Average r^2 value over all variables

SABiR	Hardware	0.99842
	Software	0.96054
BHR	Hardware	0.99995
	Software	0.84491
	Software excluding heart variables	0.99390

model the normal behavior of our testbed robots. We present the r^2 value of both software and hardware variables but our fault localization technique only use software variables. As we can see, it is much easier to model the hardware variables than software variables. Both robot's hardware variables' average r^2 values are better than those of software variables. In the BHR robot, we found that three variables are not predicted well. That is because they are the real heart variables and heart sometimes moves unexpectedly. However, the results demonstrate that our DBN can predict accurately on other variables. Generally, these results indicate that the augmented DBN models are good models of normal behavior for our robots.

4.4 Fault Localization

We test the hypothesis that the FLECS algorithm will be better at localizing faults in the robot controllers than state-of-the-art coverage-based baselines. We use two coverage-based techniques as baselines in these experiments: PFiC and Ochiai. Both of them are described in section 2.3. Since the two robots have different characteristics and their state variables have different physical meaning, we use different criteria to label the starting point of each abnormal trajectory. For SABiR, we label the starting point manually by plotting the needle's trajectory in MATLAB and comparing to the reference data. For BHR, we label points as anomalous when the "end effector error" variable goes beyond a predetermined normal threshold, which is the maximum value of this variable observed in the normal trajectories. Since we could not find any tools that automatically constructed the PDGs for MATLAB code, we manually constructed the PDGs and dynamic PDGs for both controllers.

Table 4.2 Rank of the first faulty statement for the SABiR controller. Lower is better.

Bug Index	PFIC	Ochiai	FLECS
1	163	163	47, 39
2	166	163	3
3	166	163	3
4	162	163	2
5	162	163	28
6	162	163	27
7	162	163	65
8	162	163	23
9	162	163	10
10	162	163	3

For the two coverage-based techniques, we constructed the coverage matrices using five points from each trajectory which is generated by the buggy version of controller. In this setting, each point on a trajectory can be viewed as a "test case" for the controller. Since there is a need to use same set of data in both baselines and our method, those five points are selected from the region where the anomalous event was triggered. Then we took 985 points from normal trajectories, resulting in a "test suite" of 1000 cases of which 15 were failures.

In the end, all three methods return a ranked list of statements in the controller code. The results of our experiments are shown in Table 4.2 for the SABiR controller and in Table 4.3 for the BHR controller. In those results, we report the rank of the first faulty statement in the list. If the rank of the faulty statement is tied with that of other statements, we place the faulty statement in the middle of the set of statements with the same score because without any other prior information, the developer would consider half of the statements to find the bug on average.

Table 4.3 Rank of the first faulty statement for the BHR controller. Lower is better.

Bug Index	PFIC	Ochiai	FLECS
1	84	84	1
2	84	84	138/1
3	84	84	2
4	84	84	29
5	84	84	18
6	84	84	75/18
7	84	84	2
8	84	84	9
9	84	84	2
10	84	84	10
Multi-bug	84	84	5

The results show the rank of the buggy statement given by FLECS, Ochiai and PFiC. The coverage based baselines, PFiC and Ochiai, do not work at all in this setting. Most of their results are not helpful. This is not surprising because the way they locate the faults is by comparing the difference in coverage between passing and failing runs of the code. However, there are very few branches in our controller code especially in BHR which has no branches. Since most embedded systems have few branches in the controller code to be efficient, there is very little signal in the coverage matrix to support PFiC and Ochiai to locate the faulty statements, which means they fail to work completely. In comparison to these two methods, FLECS consistently outperforms them by a wide margin. We discuss some interesting points in the following paragraphs.

Out of 20 bugs, the bug shown in first row of Table 4.2 is a real bug we experienced when we were testing the controller. It is caused by a missing check that a square root computation gives a complex number as a result but the result is an angle and controller cannot move the

robot to a complex angle. It is triggered in two places depending on the input parameters of the trajectory. As we mentioned in the last chapter, we do not monitor all the variables in the controller. In this bug, the variables that were being computed by the faulty statements were not selected in the list of 87 variables of which constitute the software state in our DBNs. However, the data dependencies of that statement were selected and since the statements containing the square root function "explained" many of these dependencies, it was ranked highly.

Since Bug 1 in SABiR has two buggy statements, we have reported the ranks of the two statements in that row. The first number is the rank of the first faulty statement that we found and the second number is the rank after we fix the fault of the first statement by adding a check there.

In Bug 7, the rank is 65. The reason why FLECS did not return a higher rank is that we did not monitor all the variables. In fact, our DBN did indicate the variable in the faulty statement as an anomalous variables. However, because a lot of abnormal variables are not monitored in our DBN, the faulty statements' ancestors and descendants are ranked higher than themselves.

In the BHR results (Table 4.3), we observe that the results for FLECS are occasionally weak for some bugs. As we can see in the table, Bug 2 and Bug 6 have two ranks and the first ranks are 138 and 75. Interestingly, this is because of the criterion we used to determine the starting point of the anomalous event. Since we choose different criteria to label the different robotic systems' states, it is possible that the criterion we used is not the best one. In BHR, we use an automatic criterion for labeling the starting point rather than a manual labeling approach like SABiR. For Bug 2 and Bug 6, it turns out that the automatic procedure returns a later starting point compared with the true starting point. Since the controller uses the output in the last time step as an input or feedback in current time step, the errors may accumulate over time. When we arrive at the automatically chosen labeled starting point, a number of additional variables beyond those causing the fault are identified as being anomalous by our approach. This results in additional ties of suspicious statements and a weaker rank for the faulty statement. To illustrate this effect, we manually label the starting point for these cases and record the rank

returning by FLECS. The result this gives us is shown in the second rank of Bug 2 and Bug 6. As we can see, the new rank is much better than the old one.

We manually create a faulty event containing multiple faults in BHR and we show its rank in eleventh row of Table 4.3. We combine Bug 1 and Bug 7 together to create a buggy controller version which has multiple faults. In this case, the first bug to be "found" is Bug 1 with rank 5. After we fix the Bug 1, we then evaluate our method again and found another Bug 7 in rank 2. In fact, when Bug 1 is alone, the method gives a rank 1 and it drops to rank 5 when one trajectory has two bugs. This is because the additional bug affects the abnormal variables we found. This experiment is an indication that the technique can work well when the controller has multiple faults.

4.4.1 Sensitivity

In the last experiment, we observed that in some cases FLECS works better after we correct the starting point. Then we decide to explore this interesting direction. We carry out a set of experiments to test the sensitivity of our method to incorrect start point identification for anomalous events. For each bug in BHR, we evaluate our technique on each point before and after the identified starting point until the rank of the faulty statements are stabilized (subject to a maximum range). Because we do not know how many points we need to find the correct starting point, we test each point before and after the starting point and pick 10 points which are most representative of the slope, to draw a figure as a result.

The results are shown in Figure 4.2 and Figure 4.3. Figure 4.2 shows the experiment results from Bug 1 to Bug 6 and Figure 4.3 shows Bug 7 to Bug 10. The red circles in the figures are identified starting point. As we can see in the table, the sensitivity of our technique is not equal over all the bugs. In some bugs like Bug 1, most of the points before and after the original starting point give a close rank of the faulty statement. However, in other bugs like Bug 2, there is a jump between point 3 and point 4, which means that after delaying the starting point, our approach could not assign the faulty statement a high rank anymore. Point 3 gives us a much better rank than point 4. An interesting outcome happens in Bug 5. The rank goes down

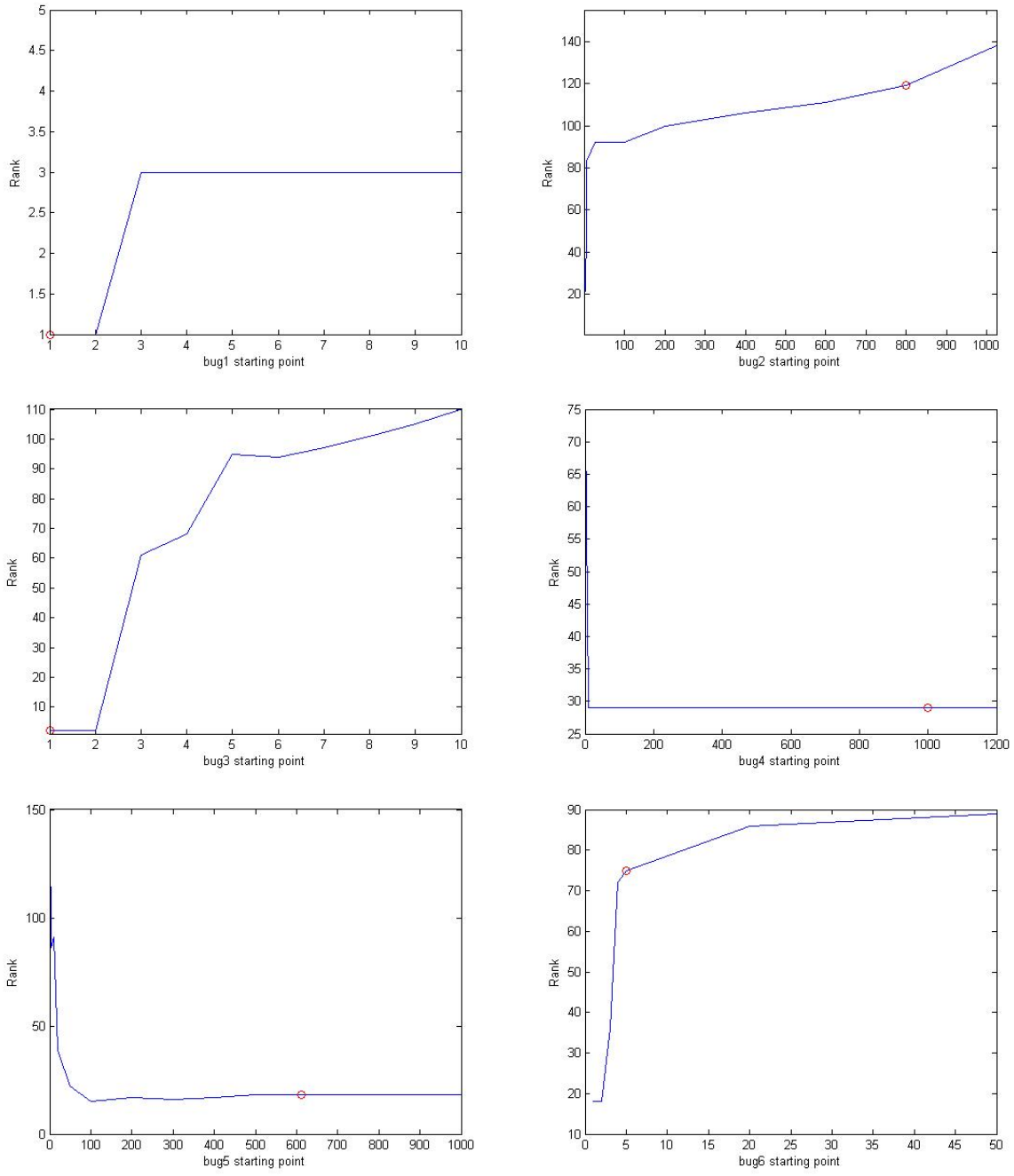


Figure 4.2 Sensitivity results. Bug1 to Bug6 for BHR.

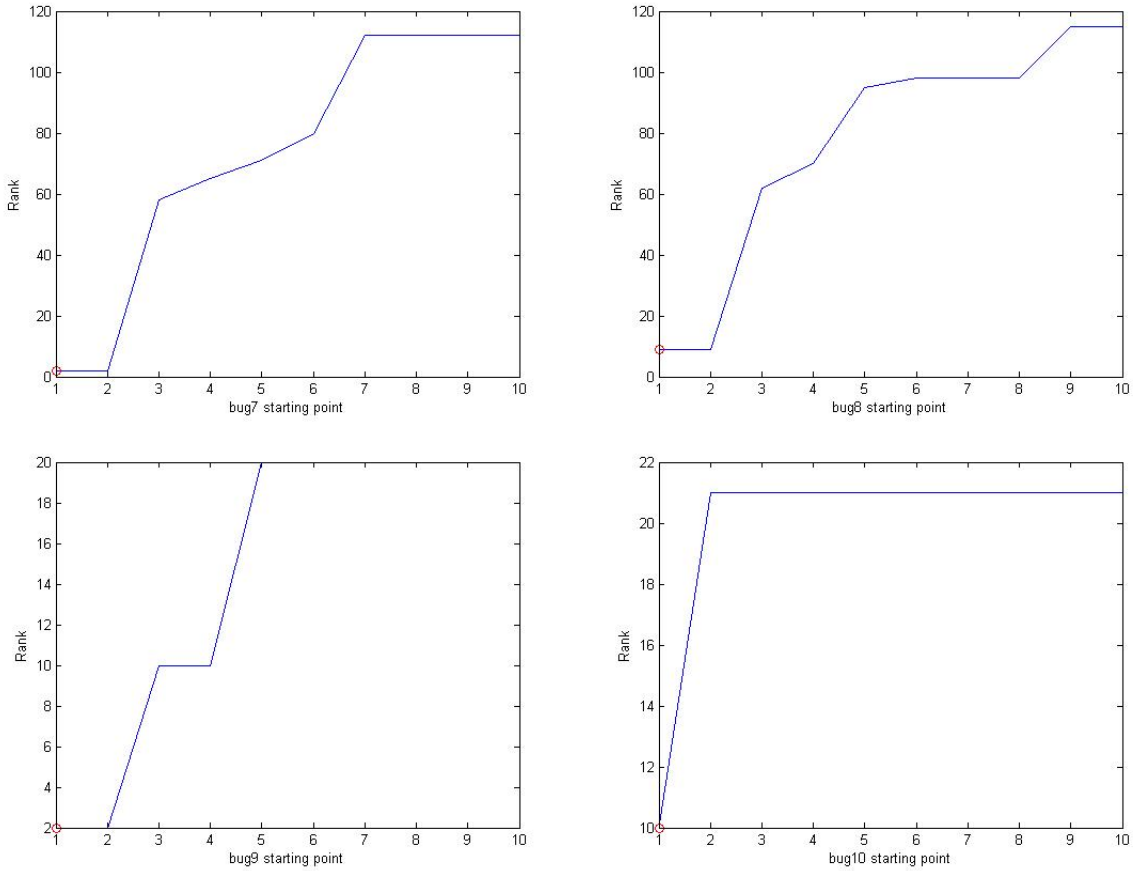


Figure 4.3 Sensitivity results. Bug7 to Bug10 for BHR.

while delaying the starting point. This is because in some specific bugs, the model might flag some variables as abnormal even if they are not actually behaving very badly before the bug happens.

4.5 Limitations

We see that FLECS may not always isolate the faulty statements at the top of the ranked list. This may be because we are not able to monitor all the variables that are incorrectly computed and our anomalous variable selection criterion may return too many variables if the starting point is identified incorrectly. Both of those can influence the precision of our

technique. Nonetheless, the technique seems generally promising and certainly performs significantly better than coverage-based techniques in our tasks.

Another problem we need to consider for FLECS is a criterion to filter out anomalous variables and the approach to measure the dependencies between each statement. Currently we use the "distances" between two statements to measure their dependency. As we described in Section 3.4, the distance between a statement and its parent statement is 1. However, the dependencies between two statements can be influenced by the content of the statements. For example, suppose we have a statement X: $x = 100 * a + 0.001 * b$. a and b are calculated from other statements A and B. In this case, a obviously influences the value of x more than b does, which means the distance between X and A should be larger than the distance between X and B even though A and B are both directly influence statement X. Fortunately, this particular scenario does not happen in the controllers we have tested so far. It may be perhaps avoided using more advanced causal inference techniques.

In conclusion, in some cases, the location of the starting point of the A&A event matters significantly. However, in many cases the technique appears to be robust to the precise location of the starting point. It is certain that a better labeling approach could help our method to achieve a consistently good result.

Chapter 5

Conclusion

In conclusion, the work we have done describes an algorithm to locate faulty statements in embedded control software systems, which is widely used all over the world. Due to the characteristics of those systems, coverage-based fault localization techniques do not work. However, after modeling the system dynamics, it is possible to locate the faults by indicating the abnormal variables in the probabilistic model. Our method first constructs a probabilistic model for the system. Then we use the model to find the anomalous variables in the start point of the anomalous event. Given the anomalous variable set, our approach traces through the PDG to determine the candidate statements that cause the anomalous behavior. Our experiments use two medical robot prototypes developed in our lab. In the fault localization experiments, the results demonstrate that the approach consistently outperforms coverage-based techniques in our setting. In addition, the sensitivity experiment shows that a strong label criterion could improve our approach. In the future work, if more embedded systems are available, we could test our technique on them and improve our DBNs by collecting more empirical data. Also, as we discuss in the Section 4.5 improving the distance formula would help our algorithm to localize faults more precisely. In current work, we are designing a graphical user interface for labeling traces and an improved automated labeling procedure for BHR, which should help mitigate one source of error in our approach.

LIST OF REFERENCES

[1] "Introduction to robots & robotics." http://www.robotplatform.com/knowledge/Introduction/Introduction_to_Robots.html.

[2] "Ward unveils legislation for future highway safety and energy conservation: driverless cars." http://houseminority.wordpress.com/2012/01/19/.

[3] E. E. Tuna, T. J. Franke, O. Bebek, A. Shiose, K. Fukamachi, and M. C. Cavusoglu, "Heart motion prediction based on adaptive estimation algorithms for robotic-assisted beating heart surgery," *Robotics, Transactions on Institute of Electrical and Electronics Engineers*, vol. 29, no. 1, pp. 261–276, 2013.

[4] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th Institute of Electrical and Electronics Engineers/Association for Computing Machinery international Conference on Automated software engineering*, pp. 273–282, Association for Computing Machinery, 2005.

[5] B. SARAH, "Driverless vehicles: liability and new automotive technologies," June 2013.

[6] "FDA launches investigation into da vinci complaints." http://www.advisory.com/daily-briefing/2013/04/12/fda-launches-investigation-into-da-vinci-complaints, 2013.

[7] R. H. Taylor and D. Stoianovici, "Medical robotics in computer integrated surgery," *Transactions on Institute of Electrical and Electronics Engineers Robotics and Automation*, vol. 19, no. 5, pp. 765–781, 2003.

[8] R. H. Taylor, *Computer-integrated surgery: technology and clinical applications*. MIT Press, 1996.

[9] E. Dombre, P. Poignet, F. Pierrot, G. Duchemin, and L. Urbain, "Intrinsically safe active robotic systems for medical applications," in *Proc. 1st IARP/IEEE-RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environment, Seoul*, pp. 21–22, 2001.

[10] G. Duchemin, P. Poignet, E. Dombre, and F. Peirrot, "Medically safe and sound [human-friendly robot dependability]," *Robotics & Automation Magazine, Institute of Electrical and Electronics Engineers*, vol. 11, no. 2, pp. 46–55, 2004.

[11] S. B. Ellenby, "Safety issues concerning medical robotics," in *Institute of Electrical and Electronics Engineers Colloquium On Safety and Reliability of Complex Robotic Systems*, pp. 3–10, IET, 1994.

[12] M. W. Whalen and M. P. Heimdahl, "An approach to automatic code generation for safety-critical systems," in *Proceedings of the 14th Institute of Electrical and Electronics Engineers International Conference on Automated Software Engineering*, 1999. Orlando, FL, USA.

[13] M. C. Çavuşoğlu, *Wiley Encyclopedia of Biomedical Engineering*, ch. Medical Robotics in Surgery. John Wiley and Sons, Inc, 2006. M. Akay, editor.

[14] FDA, "Adverse event report 2955842-2008-01144: Intuitive surgical inc., Da Vinci S Surgical System endoscopic instrument control system." `www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfMAUDE/Detail.CFM\?MDRFOI\_\_ID=1077464`), month = July, year = 2008,.

[15] Intuitive Surgical Inc., "Da Vinci S Surgical System." www.intuitivesurgical.com/products/ davincissurgicalsystem/index.aspx, 2009.

[16] O. Bebek, M. J. Hwang, B. Fei, and M. C. Çavuşoğlu, "Design of a small animal biopsy robot," in *30th Annual International Conference of the Institute of Electrical and Electronics Engineers Engineering in Medicine and Biology Society*, pp. 5601–5604, Institute of Electrical and Electronics Engineers, 2008.

[17] T.Liu, "Design and prototyping of a three degrees of freedom robotic wrist mechanism for a robitic surgery system," 2010.

[18] M. C. Cavusoglu and D. Feygin, "Kinematics and dynamics of phantom (tm) model 1.5 haptic interface," *University of California at Berkeley, Electronics Research Laboratory memo M*, vol. 1, 2001.

[19] T. H. Massie and J. K. Salisbury, "The phantom haptic interface: A device for probing virtual objects," in *Proceedings of the ASME winter annual meeting, symposium on haptic interfaces for virtual environment and teleoperator systems*, vol. 55, pp. 295–300, Chicago, IL, 1994.

[20] R. C. Jackson and M. C. Çavuşoğlu, "Modeling of needle-tissue interaction forces during surgical suturing," in *Proceedings of the Institute of Electrical and Electronics Engineers International Conference on Robotics and Automation (ICRA), St. Paul, MN, USA*, May 14-18 2012.

[21] O. Bebek and M. C. Cavusoglu, "Intelligent control algorithms for robotic-assisted beating heart surgery," *Robotics, Transactions on Institute of Electrical and Electronics Engineers*, vol. 23, no. 3, pp. 468–480, 2007.

[22] M. C. Çavuşoğlu, D. Feygin, and F. Tendick, "A critical study of the mechanical and electrical properties of the phantom haptic interface and improvements for highperformance control," *Presence: Teleoperators and Virtual Environments*, vol. 11, no. 6, pp. 555–568, 2002.

[23] G. F. Franklin, M. L. Workman, and D. Powell, *Digital control of dynamic systems*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[24] D. M. Auslander, *Real-time software for control: program examples in C*. Prentice-Hall, Inc., 1990.

[25] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pp. 73–84, ACM, 2010.

[26] G. Shu, B. Sun, A. Podgurski, and F. Cao, "MFL: Method-level fault localization with causal inference," in *Software Testing, Verification and Validation (ICST), 2013 Institute of Electrical and Electronics Engineers Sixth International Conference on*, pp. 124–133, Institute of Electrical and Electronics Engineers, 2013.

[27] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pp. 49–60, ACM, 2010.

[28] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *Transactions on Institute of Electrical and Electronics Engineers, Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[29] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *Transactions on Institute of Electrical and Electronics Engineers, Software Engineering*, vol. 36, no. 4, pp. 528–545, 2010.

[30] B. Halder and N. Sarkar, "Robust fault detection of a robotic manipulator," *International Journal of Robotics Research*, vol. 26, no. 3, pp. 273–285, 2007.

[31] T. Dean and K. Kanazawa, "A model for reasoning about persistence and causation," *Computational intelligence*, vol. 5, no. 2, pp. 142–150, 1989.

[32] L. B. J. F. R. Olshen and C. J. Stone, "Classification and regression trees," *Wadsworth International Group*, 1984.

[33] K. Liang, F. Cao, Z. Bai, M. Renfrew, M. C. Cavusoglu, A. Podgurski, and S. Ray, "Detection and prediction of adverse and anomalous events in medical robots," in *Twenty-Fifth Innovative Applications of Artificial Intelligence Conference*, 2013.

[34] N. Friedman, I. Nachman, and D. Peér, "Learning Bayesian network structure from massive datasets: the sparse candidate algorithm," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 206–215, Morgan Kaufmann Publishers Inc., 1999.

[35] F. Fleuret, "Fast binary feature selection with conditional mutual information," *The Journal of Machine Learning Research*, vol. 5, pp. 1531–1555, 2004.

[36] D. Posada and T. R. Buckley, "Model selection and model averaging in phylogenetics: advantages of Akaike information criterion and Bayesian approaches over likelihood ratio tests," *Systematic biology*, vol. 53, no. 5, pp. 793–808, 2004.

[37] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *Proceedings of the 23rd international conference on Software engineering*, pp. 339–348, Institute of Electrical and Electronics Engineers Computer Society, 2001.

[38] Y. Zhan and J. A. Clark, "Search-based mutation testing for Simulink models," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pp. 1061–1068, ACM, 2005.

[39] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher, "Mutation-based test case generation for simulink models," in *Formal Methods for Components and Objects*, pp. 208–227, Springer, 2010.

[40] Javaparser, "Java 1.5 parser with AST generation and visitor Support." `http://code.google.com/p/javaparser`.