Fault Localization in Embedded Control System Software

Kai Liang, Zhuofu Bai, M. Cenk Çavuşoğlu, Andy Podgurski and Soumya Ray
Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, OH, USA
Email: {kxl307, zxb31, mcc14, podgurski, sray}@case.edu

***Abstract*—We describe an approach to automatically locating faulty statements in control code in embedded systems. Our approach uses the controller structure and examples of normal behavior in simulation to build structured probabilistic models that compactly encode the dynamic behavior of the system. Given an anomalous behavior sequence, we analyze the values of system state variables to determine which variables are responsible for the behavior. 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 causing adverse and anomalous events during the systems' operation.**

I. Introduction

Embedded systems are cyberphysical systems that use computing elements within more complex devices to fulfill specific functions. These systems are ubiquitous; devices ranging from washing machines to cars to medical robots contain many computing elements embedded within them. The variety of applications makes it likely that the number of microprocessors used in embedded systems is several times larger than that used in general purpose computing machines.

Controllers are the software part of some embedded systems that control the behavior of the system. For example, consider a robot that has a high level goal of moving from one location to another. Typically, a solution to this goal will be computed using a path planner. The onboard controller will then execute the solution through some variant of a sense-plan-act loop where at every point, it senses the current state of the system, reasons about the state and (estimates of) the environment and determines and carries out an action. Controllers of this sort are also ubiquitous. Besides robots, an obvious example is a car; many modern cars have onboard controllers that perform a variety of assistive tasks such as stability and traction control, dynamic braking, adaptive cruising and so forth.

Many embedded control systems are used in safety-critical scenarios where faults can have serious consequences. Thus there is a need for techniques that can aid programmers in building bug-free embedded system code. While a large literature exists on building reliable control systems, most of this work is primarily focused on deriving mathematical models of the controlled system with the goal of finding control laws that enable the system to operate in stable regions of its configuration space. In some scenarios, autocoding [1] can be used together with this process to ensure that the resulting controller code is free of faults. This is not always possible, however, 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 the system due to software bugs is a possible concern. Our focus is on algorithms that can help locate bugs in control code, therebey reducing development time and improving the safety of the system.

Prior work on statistical fault localization has generally attempted to develop code-coverage-based techniques to locate faults in arbitrary code. This is an extremely difficult problem. We focus on a subset of programs, namely embedded control systems. As we argue above, this is still a very large subset given the ubiquity of these systems. Nonetheless, this focus differentiates us from prior work by giving us additional leverage on the localization problem. In particular, it is typically the case that for such systems, a *simulation* of the system and an environment is built on which the controller is tested before deploying on the physical hardware. In our work, we take advantage of this by using the simulator to build a compact *statistical model* of the system under normal operation, which we use to guide the fault localization process.

The focus on control systems, however, also leads to an interesting challenge. Most low-level controllers, including the ones we evaluate, contain very few branch points. Generally, they are sequences of numerical computations that estimate the state of the system, compare to a reference state, solve an optimal control problem and execute the solution. A bug in this context could be the result of an incorrect mathematical operation in the control code. This means that standard coverage based fault localization techniques cannot in general be used because essentially the same sequence of statements is executed whether the output is faulty or not. Rather than just considering if a statement was covered, a key contribution of our work is the consideration of the *values* of the variables in the program by the localization algorithm. It is these values, and not coverage, that determine whether the output of the controller is incorrect.

In the following, we first provide additional background for our problem domain. Next, we describe our approach in detail,

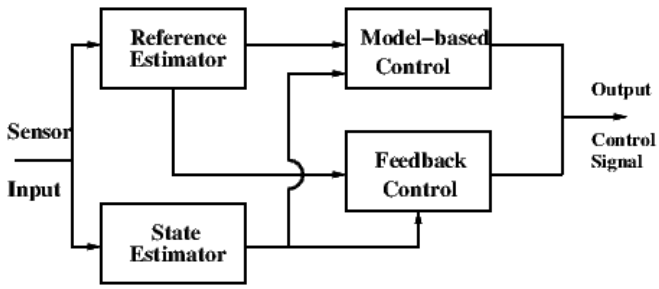

Fig. 1. Block Diagram of a Typical Controller.

followed by empirical evaluation. We then discuss lessons learned and current limitations. Finally we give an overview of related work and conclude.

II. Motivation and Background

While embedded control systems are ubiquitous, the specific systems that we focus on in our work are *robotic surgery systems*. These are modern medical devices that are used in surgery to enhance surgical precision, to allow access to locations where manual instruments have difficulty reaching and to allow image guided surgery [2], [3]. These systems are increasingly deployed in hospitals because they promise to decrease time spent in the operating theater, speed patient recovery time and decrease the chance of side effects. While promising, these systems are also complex, both in terms of their hardware and software, and safety-critical. The complexity raises the possibility of accidents due to hardware malfunctions, software malfunctions (which we focus on in this work), observability limitations, or human-machine interface problems. Indeed such events have already occurred, as evidenced by a number of *adverse and anomalous* (A&A) event reports filed by manufacturers with the Food and Drug Administration (FDA). While it unknown whether these events were caused by software issues, that possibility cannot be discounted. As such, we feel it is important to assist developers with automated tools that can help debug the control systems of such devices. We will use the controllers of two such systems developed in our labs in our empirical evaluation.

Many embedded control systems, including robotic surgery systems such as the above, generally have an associated simulation of their hardware and environment. 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.

In Figure 1 we show a general block diagram of control software. The controller typically will first estimate the current state of the system using the available sensor inputs. In some cases, a reference trajectory that the system should follow may be available, in other cases it may be estimated. Then the controller 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. Next 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. As an example, this is 2kHz in our case. This means that the entire control code has to execute in less than $500\mu s$. 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. 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. Often languages such as MATLAB are used, where single lines of code operate on entire matrices, to take advantage of the speedup provided by vectorization. Finally, the code has very few branch points. This is also quite common in control system software [4], [5] because the core algorithm is about the same (as described above) in most cases, and as can be seen from Figure 1, 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.

III. Fault Localization Algorithm

We now describe our approach to locating faults in control system software. We are interested in faults that cause anomalous behavior in the system. We call the approach Fault Localization in Embedded Control Software (FLECS).

Our approach has three parts: (i) constructing a statistical model of the system, based on the controller's behavior in simulation, (ii) determining the variables causing anomalous behavior using this model and (iii) using these variables to rank statements in the controller in order of suspiciousness. As is common in fault localization, we assume that the developer will use this ranking to guide their debugging effort, and (assuming our approach works) the faulty statement(s) will appear near the top of the ranked list, thus reducing the effort needed to debug the program. This is not a perfect way to measure effectiveness (e.g. it ignores other sources of information a developer may have), but it is commonly used in fault localization and at least offers an objective comparison between techniques, so we adopt it. In the following, we describe each part of our approach in more detail.

A. Statistical Model of Normal Behavior

As the system being controlled operates, it traces out a *trajectory* in its configuration space. Because the system is an engineered system, this trajectory (under normal conditions) is designed to be "smooth" in a certain sense; if it were

not, the system would be very hard to control. We encode this "smoothness" property using a dynamic probabilistic first order Markov model that represents how the *values* of the state variables evolve. Specifically, we use dynamic Bayesian networks (DBNs) [6], [7]. These models represent the probability of the next state given the current state, i.e. $\Pr(S_{t+1}|S_t)$, where each S_t is described by a vector of variables, as follows:

$$\Pr(S_{t+1}|S_t) = \Pr(\mathbf{V}_{t+1}|\mathbf{V}_t) = \prod_{i=1}^{n} \Pr(V_{t+1}^i|\mathbf{V}_t),$$

where $\mathbf{V}_t = \{V_t^i\}$ denotes all of the state variables in the tth time step, including both hardware and software variables. We specify the conditional probability distributions (CPDs) using regression tree models [8]. Each internal node of the regression tree is a test on some variable at the previous time step. Each leaf node is a linear Gaussian model. Regression tree models such as these are a very general, nonparametric representation of nonlinear dynamics; they can be expected to work well for a variety of control systems.

We learn the structure and parameters of our DBNs from data. To do this we generate sequences of normal trajectories from our simulations and estimate the CPDs for the variables from them. CPD parameters are estimated using maximum likelihood; for linear Gaussian models, this is equivalent to linear regression and yields closed form solutions. For regression tree models, we use a greedy top down recursive decomposition approach [8], where the goodness of a split is computed by the improvement in the r^2 measure. In addition to the basic procedure, we also perform feature selection using a variant of the Sparse Candidate [9] feature selection algorithm to limit the number of parents for each variable when learning the CPDs. We do this because we know that engineered embedded systems are designed to be sparse, i.e. most state variables tend to depend on few other variables.

In our experiments on our robots, the resulting learned DBNs are extremely accurate (test-set r^2 values on normal trajectories generally exceed 0.99) [7]. Because these models are capable of general representations of nonlinear dynamics, we argue that (given enough data), they should be equally suitable for application to other control systems as well.

B. Identifying Anomalous Variables

Now consider a scenario where we have learned the DBN and observe a new trajectory of the system which contains an anomalous event. Here we describe a procedure to identify which variables are "responsible" for the event.

Suppose the time index of the point at which the event starts to occur is T (we discuss how to find this point in more detail below). Using the values of the variables in the state at $T-1$, s_{T-1}, and our DBN, we can calculate the likelihood of s_T, the state at time T, given s_{T-1}, i.e. $\Pr(s_T|s_{T-1})$ (Note $\Pr(S_T|s_{T-1})$ will be a Gaussian distribution according to our model choices). Since this point is an anomalous point, we expect this likelihood to be low. But since this likelihood is composed of products over the individual state variables, in order for it to be low, some variable(s) must have low likelihood, which will happen if these variables' values are too different from their "expected" value (the mean of the Gaussian). In other words, using the DBNs, we can identify the potential causes of the anomalous event as low-likelihood variables in the DBN. Thus if we sort the variables in s_T by likelihood and pick the last few, those are likely to be good candidates for causing the anomalous behavior.

Require: Control System Software and Simulator, trajectory with anomalous event
Ensure: Ranked list of statements in order of suspiciousness.

```
D ← learned DBN from normal trajectories obtained using simulator (Section III-A)
T ← corrected time index of start of anomalous event
t = T; BadVars ← φ
while BadVars == φ do
    BadVars ← list of low-likelihood variables at t according to D (Section III-B)
    t ← t + 1
end while
P ← dynamic PDG of controller at T
O ← lines computing final values of BadVars in P
Rank ← zero 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) + 20 {Here we assume 20 is longer than all paths in P.}
        end if
    end for
end for
return statements in P sorted by Rank
```

Fig. 2. The FLECS Algorithm.

How many variables should we pick? For each variable, we maintain a range of "normal likelihoods" obtained from the data used to train the DBNs. In other words, for each variable v, we compute $\Pr(v_{t+1}|\mathbf{V}_t)$ for every point in our training sample and store the minimum and maximum values. We pick all the variables for which their likelihood in s_T lies outside the range of their normal likelihoods.

It is possible in some cases that the criterion above will result in nothing being selected. This can happen when a bug results in a small error accumulating over time, so that at the start point, the likelihood is still not *too* small. In such a case, we consider the states $s_{T+1}, s_{T+2}, ...$ in turn until we find a state where our criterion returns a nonempty set.

Now we discuss how to find the time index of the point at which the anomalous behavior starts (T above). One way to identify this point is through human inspection. This is analogous to the pass/fail labeling that must be done for test cases in software testing; programs producing complex outputs may require manual inspection. Alternatively, an automatic

criterion chosen by the embedded system designer can be used to identify the starting point in some cases. Here, typically, some variable's value is monitored; once it goes beyond a certain value an anomalous event is declared to have started. In our experiments, we consider both cases.

If an automatic approach is used, however, one concern is that it might not be accurate. In particular, it may be the case that when the variable monitored goes outside acceptable range, the anomalous event may already have been underway for a while. This can happen because of the way controller code runs: at every time point, the code is essentially being re-run on its previous output. Thus errors can slowly accumulate over time, until the automatic label is triggered. However, from the perspective of accurately identifying the starting point, this is problematic, because the automatically found start point is in fact in the middle of the event.

Since we have accurate DBN models, however, we remedy the situation above as follows. From the automatically identified starting point, we work our way *back* along the trajectory, considering points at $T-1$, $T-2$ etc. If the identified point was in the middle of an anomalous event rather than the actual starting point, we will observe that the number of anomalous variables identified by the procedure above will *decrease* as we work backwards, because we are reducing the error accumulation in variables not directly responsible for the event, but which are simply being affected during subsequent iterations. Eventually, this decrease in the number of anomalous variables will stabilize (before it shrinks to zero as we move past the true start point). We treat the time index at which the number stabilizes as the true start point of the anomalous event in our approach.

How important is it to be accurate about identifying the start point? Because our samples are relatively fine grained at 2kHz, we find in practice that our approach is robust to small misidentifications of the start point. However, in some cases we have observed errors of around 1000 steps with an automatic criterion, which can lead to poor performance unless the correction described above is used.

C. Identifying Suspicious Statements

Once we have a set of anomalous variables identified as above, we need to translate this to possibly faulty statements in the controller code (line 8-19 in Figure 2). We describe a heuristic technique to do this by analyzing the program dependence graph (PDG) of the controller. Specifically, we consider the dynamic PDG that was induced when traversing the PDG with s_{T-1} as the input (T being the starting index identified in the previous step), which produces the anomalous output s_T. We first augment this PDG with data dependences that reflect dependences from a previous invocation to the next invocation; these are not reflected in the standard PDG, but are implicit in the way the controller code is executed. In this augmented dynamic PDG, we first locate the nodes corresponding to the statements O that compute the anomalous values of the variables we have identified. Now the intuition is that if we find a statement in the PDG that is a data or control ancestor of *and* close to everything in O, that statement should be highly suspicious and should be examined first for a fault, because it is likely to be a "common cause" for all of the observed anomalies. However, a single such statement may not exist, or, if it does, may be close to the root of the PDG, which may not be helpful. Then expanding on our intuition, suppose we consider an arbitrary statement p in the controller. The more statements in O that p is an ancestor of, and the closer it is to those statements, the more suspicious we will treat p to be. Thus for each statement p in the program, we sum their distances in the PDG to every statement in O (lines 11-18 in Figure 2). For a p that is not an ancestor of some o in O, we add a constant to its sum that is larger than the largest path length in the PDG. We then rank statements by this sum and return the ranked list. It is clear that in this ranking, a statement p that is the "nearest" common ancestor to all the o's will have the best rank, if it exists.

An implicit assumption in this procedure is that faults are rare in the code, so that a single "common cause" (or an approximation of it) is more suspicious than many separate causes. In general this is a reasonable approximation for mature code, because it would be difficult to generate normal trajectories with controller code containing a large number of bugs. As well, sophisticated techniques are typically not needed to find faults when they are common; they are typically most valuable to hunt down the rare faults and failures.

We note here that since our controllers are written in MATLAB, a statement can have a matrix (or a vector of multiple variables) as an output. In our instrumentation scheme, described in the next section, we represent some such structures using multiple state variables. Thus if later statements use different elements of this matrix in their computations, a single statement can have data dependences indexed by different state variables in our representation.

IV. Empirical Evaluation and Discussion

We use two prototype robotic surgery systems developed in our lab to evaluate our approach. The first robot is a Small Animal Biopsy Robot (SABiR) [10], and the second is a manipulator that will allow minimally invasive beating heart surgery (BHR) [11].

The SABiR Robot. SABiR 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 and to achieve accuracy better than 250μm (Figure 3 (left)). The controller is written in MATLAB and has 330 lines of code in 6 functions, and contains 11 branches. We use models for the kinematics and inverse kinematics developed in prior work [12] to create a simulation of the robot. The environment of the simulated robot consists of a gel block (to simulate "tissue") placed in the workspace (Figure 3 (center)).

The BHR Robot. BHR uses a three degrees-of-freedom (DOF) robotic platform employing a PHANToM Premium 1.5A haptic interface [13] as the robotic mechanism (Figure 3 (right)). This system's characteristics make it suitable for eventual use in minimally invasive beating heart surgery [11],

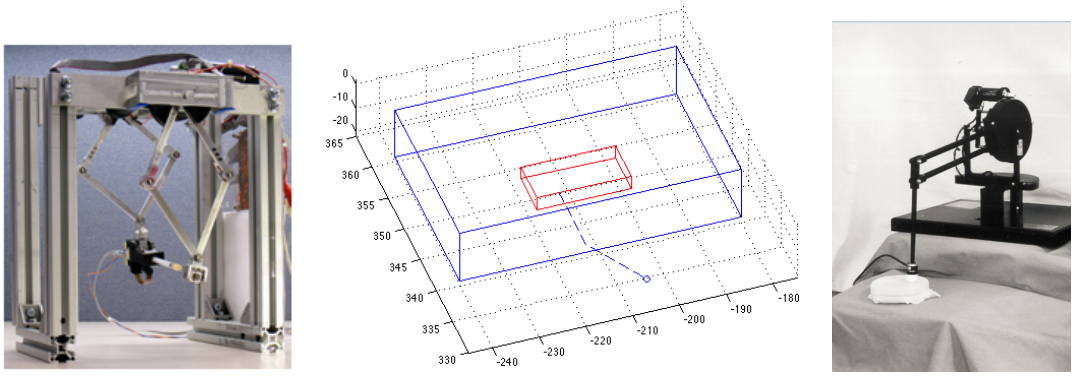

Fig. 3. The SABiR robot (left), a normal needle-tip trajectory in the simulation environment consisting of two "tissue" blocks (center), and the BHR robot (right) showing a tissue phantom "heart."

with sufficient motion and speed for heart motion tracking. The controller is written in MATLAB and has 167 lines of code in 9 functions, and has no branches. We use models for the kinematics and inverse kinematics developed in our prior work [14] to create a simulation of the robot The environment of this robot consists of simulated heart and breathing motions, which the robot arm needs to follow.

Instrumenting the Controllers. To collect data for our approach, we need to record the final values of variables after the controller finishes processing one time step. To keep the data collected reasonable, we chose to monitor 100 or fewer variables from each controller. We used two heuristics to determine which variables to monitor. First, for large matrices that are used in statements as a single entity (e.g. $X = A*B$), a candidate for monitoring is a single, randomly chosen element (from X in this case). The rationale here is that if this statement is faulty, *all* the matrix elements are affected by it. However, if a statement uses a specific element from a matrix (e.g. $X(2,2) = A(2,3) * B(3,2)$) then that element is a candidate for us to monitor. For matrices of size ten or less, however, every entry was a possible candidate for monitoring.

We use a second rule to prune the list of candidates determined above. Here, we select those variables that are "hub" variables in the controller's PDG in that they influence many other variables' values. In our controllers, selecting variables that influence at least five other variables in the program gives us a reasonably sized set of variables to monitor. As a result of this selection process, 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, after the controller finishes running, and used to generate training and testing data.

Faulty Controllers. To evaluate the techniques, we need buggy versions of the controllers. Through randomized testing we discovered one bug in the SABiR controller that triggers because of a missing check. This causes a subsequent square root computation to return a complex number of which only the real part is stored and processed, leading to anomalous behavior in the simulator. To generate additional bugs, we used mutation testing. Here we create a set of faulty versions of the controllers (mutants) through the randomized modification of elements of the code. This approach has been used in prior work [15]. We use four mutation operators: replace numerical constant, negate jump condition, change arithmetic operator, and add/subtract a small numeric value. We repeatedly randomly mutate an expression in the code and retain those which produce anomalous behavior. Through this process, we generated nine faulty controllers for SABiR and ten for BHR.

Baselines. We use three statistical fault localization baselines. Two are coverage based techniques: PFiC and Ochiai [16]. PFiC simply estimates the suspiciousness of a statement s in a program using the probability that the program will fail if the statement is covered ($\Pr(Fails|s\ covered)$), which can be empirically estimated by simply counting the number of failing runs where the statement is covered and dividing by the number of runs that cover the statement. The Ochiai metric estimates the suspiciousness of a statement s as a product:

$$Ochiai(s) = \sqrt{\Pr(Fails|s\ covered)\Pr(s\ covered|Fails)}$$

Our third baseline is Exploratory Software Predictor (ESP) [17], a predicate-level statistical debugger which employs "elastic predicates" to partition the value space of each instrumented variable x. Elastic predicates use execution profiles to compute the mean μ_x and standard deviation σ_x. These statistics are then used to compute predicates that relate the value of x on any run to the statistics, such as $x > (\mu_x + 3\sigma_x)$. ESP then computes an Importance score [18] for each predicate. The suspiciousness of each instrumented variable x is represented by the highest Importance score among the corresponding elastic predicates.

In our work, we map the variables' suspiciousness scores in ESP to statements so we can return a ranked list of statements using this technique as well, for uniform comparison to other techniques. For a statement that contains an instrumented variable, we use the highest suspiciousness over all such variables as the suspiciousness of the statement. For statements not containing any instrumented variable, we find the nearest ancestors and descendants in the PDG that do have an

instrumented variable (and so have a suspiciousness score). We then set their suspiciousness scores to be the highest over all the scores of these other statements.

Methodology. We use 400 normal trajectories obtained through our simulator to train our DBN for SABiR and 10 trajectories to train a DBN for BHR. Each trajectory has 25,000 datapoints for SABiR and 100,000 datapoints for BHR. For each trajectory generated with buggy versions of the SABiR controller that contained an anomalous event, we label the starting point manually by plotting the needle's trajectory in MATLAB and comparing to the reference data (Figure 3 center). For BHR, we label points as anomalous using an automatic criterion, when the "end effector error" variable goes beyond a predetermined normal threshold.

To construct the coverage matrices for the baseline techniques, we generated three trajectories for each bug where the anomalous event was triggered. Since each point on a trajectory is a "test case" for the controller, we took five points from the anomalous event region of each trajectory. We then also took 985 points from normal trajectories, resulting in a "test suite" of 1000 cases of which 15 were failures.

Each technique we use returns a ranked list of statements in the controller code. We report the rank of the first faulty statement in this list. In the case of ties, we place the faulty statement in the middle of the set of statements with the same score, which simulates a developer having to consider half of this list before finding the fault. The results of our experiments are shown in Table IV.

Discussion. Many interesting observations arise from these results. We first observe that the coverage based baselines, PFiC and Ochiai, fail completely in this setting. This is not surprising, as we have discussed before. Since these programs have few branches, there is almost no difference in coverage between passing and failing runs of the code (in fact, no difference at all for BHR, which has no branches). Thus there is very little signal in the coverage matrix to help determine faulty statements, and the techniques fail to work. FLECS consistently outperforms these techniques by a wide margin.

We observe that the ESP approach is significantly better than the other coverage based baselines, and occasionally outperforms FLECS (3 bugs for SABiR and 1 for BHR). We believe ESP's good performance can be directly attributed to its consideration of variable values. In the cases where it did well, the values of variables in the failing test points were significantly different from those in the passing cases, so that the faulty statements could be easily identified. However, it occasionally also fails to find the faulty statement, indicating that for this domain, an analysis of a single time index (test case) as done by ESP has limitations. In particular, variable values may look normal relative to their entire range but *may still be abnormal in the context of the previous state*, which is information leveraged by FLECS that generally leads to more accurate localization.

One could ask whether alternative baselines could be used in our experiments. From our results, it is clear that coverage based localization approaches are unsuitable for these tasks, no matter how sophisticated they may be. This is because of the nature of the controller code, which is essentially a linear sequence of mathematical operations. Usually, the coverage matrix simply has no information about faulty statements. This rules out the vast majority of techniques proposed for fault localization in the literature. Thus we must compare to value-based techniques, of which ESP is a representative. Other state-altering approaches such as Delta Debugging [19] and value replacement [20] are among the few techniques based on values of variables. These techniques attempt to isolate the cause of program failure by altering program states and re-running the program. Though powerful, these approaches require an oracle to determine program success or failure of each alteration, unlike FLECS or ESP. As well, adapting these approaches to scenarios such as ours, where the fault may not be immediate but manifest over time as the program is repeatedly called, seems a nontrivial problem.

V. Related Work

Some prior work has used probabilistic models for fault localization, such as SOBER [21] and the probabilistic program dependence graph (PPDG) [22]. Other approaches have used causal models in this context [23], [24]. These approaches generally do not use variable values for localization and do not model their evolution over program states.

Model-based Software Debugging (MBSD) [25] is a related technique that uses a purely logical component-connection model to reflect the behavior of the program. By comparing the expected behavior and observed behavior of the model, MBSD can compute the possible explanations of which component cause the misbehavior of the program. MBSD focuses on modeling the (static, logical) structure of the program, but does not consider the dynamic, probabilistic relationship between values of variables that we learn using DBNs.

Numerous fault localization techniques have been designed that use coverage information only to compute suspiciousness scores of statements, e.g. [26], [18], [27] among others. As mentioned before, these are generally not suitable for this problem domain.

Most related research on the safety of medical robotic systems in the literature primarily focus on design of intrinsically safe systems, e.g. [2], [28], [29], [30]. While formally proving safe behavior is desirable, it may not always be possible given the complexity of these systems and the stochastic nature of the environment in which they may be deployed. Thus we advocate approaches such as ours that work alongside formal methods to aid the software development process in cases where such guarantees cannot be given.

VI. Conclusion

Control software is widely used in many embedded systems including robots and vehicles. Given the often safety critical nature of such systems, it is valuable to design automated aids to developing them. The characteristics of such systems often make coverage-based fault localization techniques unsuitable. We have presented an alternative value-based approach that

TABLE I
RANK OF THE FIRST FAULTY STATEMENT FOR SABIR AND BHR. LOWER IS BETTER.

		Bug Index									
		1	2	3	4	5	6	7	8	9	10
SABiR	FLECS	39	3	3	2	28	27	65	23	10	3
	ESP	145	6	253	29	29	24	29	21	273	24
	PFIC	166	166	162	162	162	163	166	162	163	166
	Ochiai	163	163	163	163	162	163	166	162	163	166
BHR	FLECS	1	18	2	2	18	18	1	9	2	10
	ESP	25	47	4	51	54	27	4	3	114	29
	PFIC	84	84	84	84	84	84	84	84	84	84
	Ochiai	84	84	84	84	84	84	84	84	84	84

also attempts to exploit the availability of simulators for control systems. Experiments using two medical robot prototypes developed in our labs indicate the approach is promising. In future work, we plan to collect more empirical data on the performance of our technique, as well as explore causal inference techniques to improve the effectiveness of the technique.

VII. ACKNOWLEDGMENTS

This work is supported in part by NSF CNS-1035602. Ray is supported in part by CWRU award OSA110264.

REFERENCES

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

[2] R. H. Taylor and D. Stoianovici, "Medical robotics in computer integrated surgery," *IEEE Transactions on Robotics and Automation*, vol. 19, no. 5, pp. 765–781, 2003.

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

[4] G. F. Franklin, J. D. Powell, and M. Workman, *Digital Control of Dynamic Systems*. Ellis-Kagle Press, 2006.

[5] D. M. Auslander and C. H. Tham, *Real Time Software for Control*. Prentice Hall, 1990.

[6] T. Dean and K. Kanazawa, "A model for reasoning about persistence and causation," *Computational Intelligence*, vol. 5, no. 3, pp. 142–150, 1990.

[7] 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 *Proceedings of the 25th Intl. Conf. on Innovative Applications of Artificial Intelligence*, 2013, seattle, WA, USA.

[8] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[9] N. Friedman, I. Nachman, and D. Pe'er, "Learning of bayesian network structure from massive datasets: The "sparse candidate" algorithm," in *Fifteenth Conference on Uncertainty in Artificial Intelligence*. Stockholm, Sweden: Morgan Kaufmann, 1999.

[10] 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 IEEE Engineering in Medicine and Biology Society*. IEEE, 2008, pp. 5601–5604.

[11] T. Liu, "Design and prototyping of a three degrees of freedom robotic wrist mechanism for a robitic surgery system," Master's thesis, Case Western Reserve University, 2010.

[12] M. J. Hwang, O. Bebek, F. Liang, B. Fei, and M. C. Çavuşoğlu, "Kinematic calibration of a parallel robot for small animal biopsies," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2009, pp. 4104–4109.

[13] 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, no. 1. IOS Press, 1994, pp. 295–300.

[14] M. C. Cavusoglu and D. Feygin, "Kinematics and dynamics of phantom (tm) model 1.5 haptic interface," *University of Califbrnia at Berkeley, Electronic Research Laboratory Memo M01*, vol. 15, 2001.

[15] 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*. IEEE Computer Society, 2001, pp. 339–348.

[16] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 89–98.

[17] R. Gore, P. F. R. Jr, and D. Kamensky, "Statistical debugging with elastic predicates," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011, pp. 492–495.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[19] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 342–351.

[20] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 167–178.

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

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

[23] ——, "Causal inference for statistical fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 73–84.

[24] G. Shu, B. Sun, A. Podgurski, and F. Cao, "Mfl: Method-level fault localization with causal inference," in *the 6th Intl. Conf. on Software Testing, Verification and Validation (ICST), Luxembourg*, 2013.

[25] W. Mayer and M. Stumptner, "Model-based debugging: state of the art and future challenges," *Electronic Notes in Theoretical Computer Science*, vol. 74, no. 4, pp. 61–82, 2007.

[26] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 273–282.

[27] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems and Software*, vol. 89, pp. 51–62, 2014.

[28] E. Dombre, P. Poignet, F. Pierrot, G. Duchemin, and L. Urbain, "Intrinsically safe active robotic systems for medical applications," in *Proceedings of the 1st IARP/IEEE-RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environment*, 2001.

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

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