Abstract—Software can contain faults that remain undetected prior to its release. Such faults may be detected during their on-line use using monitors (refer to our prior work [19]). It is then important to determine the plausible root-cause, namely, the faulty lines of code, or indicators for any missing lines of code. To localize a software fault to its “root-cause”, we introduce the notion of a fault-seed, a fragment of a faulty-run, and propose a model-based automated approach that analyzes the observed faulty-run of the software, recorded during its runtime operation, to determine the fault-seed. We use input-output extended finite automata (I/O-EFA) for modeling a software, whose computations are modeled as the runs of the I/O-EFA. Owing to resource constraints in certain system such as embedded system, the run-time data-logging can be incomplete, resulting in partial observation of software runs—otherwise the data-logging would incur unacceptable overhead. A feature of our analysis is to localize the possible root cause in presence of such partial observability of data variables. Another feature is that the approach is helpful in localizing faulty lines of code or an indicator for missing lines of code (as the case may be).

I. INTRODUCTION

In spite of the tremendous progress made in software testing and verification, a software can contain faults. Such faults, that remain undetected prior to the release of a software, are thus present before the software is used and hence unanticipated. Presence of software faults can result in the failure of a system, where a failure is considered the violation of one of the specifications. One approach to detect unpredicted exceptions/faults is to instrument the code by inserting assertions, which if violated, indicate the presence of a software fault [13].

In [19], we proposed a two-tiered hierarchical approach for detecting faults in embedded control software during their runtime operation: Monitoring at the control software level as well as at the controlled-system level. A software fault is immediately detected when an observed behavior is rejected by a software level monitor. In contrast, when a system level monitor rejects an observed behavior, it indicates a system level failure, and an additional isolation step is required to conclude whether a software fault occurred. The additional system level monitoring safeguards against any possible incompleteness of the software level properties (and so incompleteness of software level monitoring).

When a software fault is detected, there remains the difficult task of localizing it. Fault localization is the process of guiding and narrowing the search for identifying the faulty lines of code, or indicators for missing lines of code. Thus the fault-localization problem studied here is distinct from fault-detection (where the goal is to detect whether a failure occurred) or fault-identification (where the goal is to identify which component(s) failed that caused a failure), and we only refer to works dealing with fault-localization as defined above.

[15] presented a nearest neighbor method for fault localization under the assumption that there exist a faulty-run and a large number of nonfaulty-runs. Statistical techniques such as statement ranking based fault localization are discussed in [12]. [2], [17], [8] presented methods that exploit the nonfaulty-runs to identify the statements of a faulty-run. [5], [7] introduced a technique for localizing faults based on finding a “closest” nonfaulty execution to a given counterexample. A limitation of the above approaches is they require either having or finding nonfaulty-runs (for comparing with the available faulty-run). In contrast our approach is based on the analysis of a single faulty-run observed during run-time monitoring. Also, even when non-faulty runs are available, a comparison of a faulty run with one or more non-faulty runs can only identify the set of states unique to the faulty-run, and finding the root-cause of failure still requires further analysis.

[1], [18] discussed a dynamic slicing technique for identifying a subset of statements that are likely responsible for producing an incorrect computation. [3], [9] proposed an approach for localizing program errors based on variable dependency analysis, also called path-slicing, that removes from a counterexample-path those statements which do not affect the variables present in the specification. [16] provided an automated procedure to zoom into potential software defects by analyzing a single concrete counterexample. Path-slicing is only a way to reduce the search space for the root-cause of fault; it doesn’t necessarily identify the root-cause.

In this paper we present a way of analyzing an observed faulty-run to further narrow down the scope of possibilities for the root-cause. One of our main contributions is to formalize the concept of a root-cause by introducing the notion of a fault-seed, a fragment of a faulty-run, which can be algorithmically computed: Checking whether a chosen fragment (of the executed sequence of statements) is a fault-seed is formulated as an instance of a model-checking problem. A subset of statements included in a faulty-run is called a fault-seed if their influence execution in any run of the software causes a failure (a certain specification violation) to occur. Note a fault-seed can itself contain faults in which case itself a root-cause and referred as fault-fragment or, if not itself faulty, its execution is essential for the manifestation of failure caused by some missing code and in this case it is an indicator of the root-cause (see Section V for an example of a fault-indicator). Thus another contribution is that the approach is helpful in localizing faulty lines of code or an indicator for missing lines of code (as the case may be). Since it is expensive to record the variable values after each statement execution, another contribution of our approach is that it works with a partial data-log of a run. More details

This work was supported in part by the National Science Foundation under the grants NSF-ECS-0424048, NSF-ECS-0601570, NSF-ECCS-0801763, NSF-CCF-0811541, and NSF-ECCS-0926029. C. Zhou is with Magnatech LLC, East Granby, CT. changyan.zhou@gmail.com R. Kumar is with the Department of Elec. & Comp. Eng., Iowa State Univ., Ames, IA. rkumar@iastate.edu S. Jiang is with General Motors R&D, Warren, MI. shengbing.jiang@gm.com
are provided in Sections III and IV.

Consider for example a simple program for computing minimum (in form of output \( y_1 \)) and maximum (in form of output \( y_2 \)) for three inputs \( \{u_1, u_2, u_3\} \):

1. \( \min = u_1; \max = u_1; y_1 = y_2 = 0; \)
2. if (\( \max < u_2 \)) \( \max = u_2; \)
3. if (\( \max < u_3 \)) \( \max = u_3; \)
4. if (\( \min > u_2 \)) \( \max = u_2; \)
5. if (\( \min > u_3 \)) \( \min = u_3; \)
6. \( y_1 = \min; y_2 = \max; \)

Initially, \( y_1 = y_2 = 0 \), and it is desired that as the program evolves, the following property must be maintained: \( y_1 \leq y_2 \).

For the input \( u_1 = 5, u_2 = 1, u_3 = 6 \), the following sequence of \( (y_1, y_2) \) values is computed as the program evolves: \( (0, 0) \rightarrow (0, 0) \rightarrow (0, 0) \rightarrow (0, 0) \rightarrow (5, 1) \). Clearly the property \( y_1 \leq y_2 \) is violated in the last step. The fault lies in the 4th statement, where instead of “max” it should have been “min”.

As is illustrated later in the paper, our approach (that analyzes the program, its specification, and the above faulty-run) identifies the exact faulty statement as a fault-fragment. Note that a path-slicing based approach will present the entire faulty-run as a fault-fragment since the sequence of statements executed in the above faulty-run form a chain of causally-dependent statements. On the other hand, the approaches based on comparing a faulty-run versus a nonfaulty-run are not applicable here since a nonfaulty-run is not even available.

As remarked later (see Remark 1) our approach can also be used in conjunction with the path-slicing and comparative approaches to achieve a finer resolution in fault-localization. In this sense, our approach complements and supplements the existing approaches.

II. SOFTWARE MODEL

We use input-output extended finite automata (I/O-EFA) for modeling software. An I/O-EFA consists of locations \((L)\), data \((D)\), continuous (numeric) inputs \((U)\), continuous (numeric) outputs \((Y)\), discrete (symbolic) inputs \((\Sigma)\), discrete (symbolic) outputs \((\Delta)\), transitions \((E)\), initial locations \((L_0)\), and initial data values \((D_0)\). The locations together with the data form the state-space of an I/O-EFA. The locations are finite and form the vertices of the automaton graph. (For a software model, locations correspond to the values of the program counter.) The edges of the graph represent transitions between the locations and are guarded by constraints over the data and the inputs. (For a software model, the statements are captured as part of the edges.)

The occurrence of a transition triggers a data update and an output assignment. I/O-EFA is formally defined as follows.

Definition 1: An input/output extended finite automaton (I/O-EFA) is a nine-tuple \( P = (L, D, U, Y, \Sigma, \Delta, E, L_0, D_0) \), where

- \( L \) is the set of locations,
- \( D = D_1 \times \ldots \times D_p \) is the set of \( p \)-dimensional data,
- \( U = U_1 \times \ldots \times U_q \) is the set of \( q \)-dimensional input,
- \( Y = Y_1 \times \ldots \times Y_r \) is the set of \( r \)-dimensional output,
- \( \Sigma \) is the set of discrete (symbolic) inputs,
- \( \Delta \) is the set of discrete (symbolic) outputs,
- \( E \) is the set of edges, and each \( e \in E \) is a 7-tuple, \( e = (\sigma_e, e_1, \sigma_e, \delta_e, G_e, f_e, h_e) \), where
  - \( \sigma_e \in L \) is the origin location,
  - \( e_1 \in L \) is the terminal location,
  - \( \sigma_e \in \Sigma \cup \{e\} \) is the discrete input,
  - \( \delta_e \in \Delta \cup \{e\} \) is the discrete output,
  - \( G_e \subseteq D \times U \) is the enabling guard (a predicate),
  - \( f_e : D \times U \rightarrow D \) is the data update function,
  - \( h_e : D \times U \rightarrow Y \) is the output assignment function,
- \( L_0 \subseteq L \) is the set of initial locations, and
- \( D_0 = D_{10} \times \ldots \times D_{p0} \subseteq D \) is the set of initial data values.

We use \( \vec{u}, \sigma, \vec{y}, \delta, \vec{d}, \) and \( l \) to denote a numeric input, a symbolic input, a numeric output, a symbolic output, a data (i.e., a numeric state), and a discrete state, respectively. The numeric variables range over countable sets and can be taken to be the set of integers.

For the min/max computation program introduced earlier, the I/O-EFA models \( P \) of the program and \( R \) of the specification monitor are shown in Figure 1. Each location and each edge has been given a name, which will be used later when discussing fault localization.

The specification monitor \( R \) captures all runs that satisfy the desired invariant property, \( y_1 \leq y_2 \), written in temporal logic as \( G[y_1 \leq y_2] \) (i.e., for all paths, always \( y_1 \leq y_2 \)). Reaching the FAULT location implies the violation of the specification. (The introduction of temporal logic and the construction of a monitor from a 1st-order linear-time temporal logic are omitted. Refer to [19] for related details.)

![Fig. 1. I/O-EFA model P of min/max program (left) and specification monitor R (right)](image-url)
may not manifest as a specification violation immediately after the execution of a fault-seed, extensions are considered). We next formulate the notion of a faulty-step.

We start by defining a step, which is simply a single state transition. A step of an I/O-EFA $P$ is given by,

$$(l, d) \rightarrow (l', d')$$

where $l, l' \in L, d, d' \in D, u, u' \in U, y, y' \in Y,$ and exists $e = (o_e, t_e, \sigma_e, \delta_e, G_e, f_e, h_e) \in E$ such that

$$[\alpha_e = l, t_e = l', \sigma_e = \sigma, \delta_e = \delta] \land$$

$$G_e(d, u) \land [f_e(d', u') = d'] \land [h_e(d', u) = y']$$

Using the notion of a step, we can define the notion of run, which is simply a sequence of steps starting from an initial state. A run $r$ of an I/O-EFA $P$ is a finite sequence of steps starting from an initial state:

$$r := (l_0, d_0) \rightarrow \ldots \rightarrow (l_n, d_n)$$

where $l_0 \in L_0, d_0 \in D_0$ and for $j \in \{0, \ldots, n\}$,

$$(l_j, d_j) \rightarrow (l_{j+1}, d_{j+1})$$

is a step. A run $r$ is said to be a faulty-run of a certain specification if $r$ violates that specification.

Given a run $r$, we say that a path $\pi^r = e_0 \ldots e_n$ (a sequence of edges of $P$) is associated with the run $r$ if

$$\forall 0 \leq i \leq n : [\alpha_e = l_i, t_e = l_{i+1}, \sigma_e = \sigma, \delta_e = \delta] \land G_e(d_i, u_i) \land [f_e(d_{i+1}, u_i) = d'] \land [h_e(d_i, u_i) = y']$$

Note the length of $\pi^r$ is the same as that of $\pi$. Dually, we say that a run $r$ is associated with a path $\pi$ if $\pi^r = \pi$.

A path is said to be a faulty-path if it is associated with a faulty-run.

We say a sequence of edges $e_0 \times e_1 \times \ldots \times e_m$ is a fragment of $\pi^r$ if $0 \leq i_0 \leq \cdots \leq i_m \leq n$.

Edges in a path can influence one another. Given a path $\pi = e_0 \ldots e_n$, we say that an edge $e_i$ influences another edge $e_j$ ($i, j \in \{0, \ldots, n\}$), denoted $e_i \sim e_j$, if $j > i$ and the output of $e_j$ (as determined by $G_{e_j}$, $f_{e_j}$, $h_{e_j}$, or $\delta_{e_j}$) depends on the output of $e_i$ (as determined via $G_{e_i}$, $f_{e_i}$, $h_{e_i}$, or $\delta_{e_i}$). For a fragment to serve as a plausible root-cause of a specification violation along a path, none of the intermediate edges of the path should influence the edges of the fragment. Accordingly, a fragment $\pi^f = e_{i_0} \times \ldots \times e_{i_m}$, of a path $\pi^r = e_0 \ldots e_n$ is said to be an influence-fragment if for all $k \in [i_0, i_m] \setminus \{i_0, \ldots, i_m\}$, there does not exist $l \in \{i_0, \ldots, i_m\}$ such that $e_k \sim e_l$.

Finally, a fragment $\pi^f$ of a path $\pi^r$ associated with a faulty-run $r$ is said to be a fault-seed if (i) exists a run possessing $\pi^f$ as an influence-fragment (otherwise $\pi^f$ cannot be regarded as a candidate for a fault-seed), and (ii) for any path $\pi$, with $\pi^f$ as an influence-fragment, all subsequent paths eventually have an associated faulty-run. (Note a fault may not manifest as a specification violation immediately after the execution of $\pi$, but it must eventually manifest along each subsequent path for some run in order for $\pi^f$ to be regarded as a fault-seed.

A fault-seed can either itself be faulty (and hence itself is a root-cause) and in which case we refer to it as a fault-fragment, or otherwise (if the fault-seed itself not faulty) it indicates the presence of certain missing lines of code that manifest as failure whenever the fault-seed gets executed in any run, and in this case we refer the fault-seed as fault-indicator.

Example 1: Consider the min/max computation program with inputs chosen as: $(u_1, u_2, u_3) = (5, 1, 6)$. Then the run is given by:

$$r = (l_0 = 1, min = -, max = -) \rightarrow (5,1,6)/(0,0) \rightarrow (2,5,5)/(0,0) \rightarrow (5,1,6)/(0,0) \rightarrow (4,5,6)/(5,5,1) \rightarrow (6,5,1)/(7,5,1)$$

Its associated path is $\pi^r = e_1 e_2 e_3 l e_4 e_5 e_6$, whereas $\pi^f = e_2 e_4 e_1$ is a fragment of $\pi^r$ that can also be seen to be an influence-fragment since it computes, the value of max, independently of the statements $e_1$ or $e_3$ or $e_2$ preceding it. It so happens that the $e_2 e_4 e_1$ is also a fault-seed (and also a fault-fragment) since as is shown in the next section $e_4$ by itself is a fault-fragment.

As we will see below, checking whether a selected fragment is a fault-seed can be formulated as a model-checking problem for the software model $P$ refined with respect to the specification model $R$. The refinement is obtained via synchronous composition of two I/O-EFAs defined as follows. For notational convenience, let $\bar{r}$ denote the variables $(\bar{d}, \bar{d}, \bar{u}, \bar{y}, \sigma, \delta) \in D \times D \times U \times Y \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\})$, and define

$$p_{edb}(\bar{r}) \equiv$$

$$\sigma_e(\bar{d}, \bar{d}, \bar{y}, \sigma, \delta) \equiv G_e(\bar{d}, \bar{u}) \land [\bar{d} = f_e(\bar{d}, \bar{y})] \land$$

$$[\bar{y} = h_e(\bar{d}, \bar{u}) \land [\sigma = \sigma_e] \land [\delta = \delta_e].$$

The above predicate captures the guard-condition, the data-update function, the output-assignment function and the symbolic inputs and outputs of an edge $e \in \bar{E}$ as a single predicate. Then each edge $e$ of an I/O-EFA can be succinctly represented as a tuple $e = (o_e, t_e, p_{edb})$.

**Definition 2:** The synchronous composition of $P_1 = (L_1, D_1, U_1, Y_1, \Sigma_1, \Delta_1, E_1, L_01, D_01)$, $i = 1, 2$ is given by:

$$P_1 \| P_2 := (L_1 \times L_2, D_1 \times D_2, U_1 \times U_2, Y_1 \times Y_2, \Sigma_1 \times \Sigma_2, \Delta_1 \times \Delta_2, \bar{E}, L_01 \times L_02, D_01 \times D_02),$$

where each edge $e \in \bar{E}$ is a tuple $e = (o_e, t_e, p_{edb})$ such that the following holds:

$$\exists e_i = (o_e, t_e, p_{edb}) \in E_i(i = 1, 2) :$$

$$\sigma_e = (o_e, \bar{c}_e, p_{edb}), \ t_e = (t_{o_e}, t_{e_2}), \ p_{edb} = p_{edb}(e_1) \land p_{edb}(e_2) \land [\bar{v}_1 = \bar{v}_2].$$

According to the above definition, two edges of $P_1$ and $P_2$ synchronize if and only if both are enabled and agree on the values of all the variables.

**IV. FAULT LOCALIZATION**

Once a specification violation is detected by on-line monitoring, analysis must be performed to localize the root-cause of the failure—either faulty lines of code or indicators for missing lines of code. For certain applications such as embedded applications, resources are limited and so logging every piece of data may not be viable. Yet enough data must be recorded so that it is useful for fault localization. Thus there exists trade-off between on-line resources required...
for data-logging versus the off-line computation required for fault localization. As a general guideline, one should record sufficient data to ensure a faulty-run is not masked as a non-faulty one, and, to ensure no unbounded sequence of unobserved steps, data values for at least one edge in any cycle in the I/O-EFA model of the software should be recorded.

For illustration, consider the min/max program extended to include data-logging commands.

1. \( \min = u_1; \max = u_1; y_1 = y_2 = 0; \)
2. if \( (\max < u_2) \) max = \( u_2; \)
3. else \( \text{record} (\delta_1; y_1; y_2); \)
4. if \( (\min > u_2) \) \{ max = \( u_2; \) \text{record} (\delta_2; y_1, y_2); \}
5. \} if \( (\min > u_3) \) min = \( u_3; \)
6. \} y_1 = \min; y_2 = \max.

In the above example, the variables \( y_1 \) and \( y_2 \) are recorded, along with the labels \( \delta_1 \) and \( \delta_2 \) as identifiers. The labels are unique to statements (equivalently to the edges of the I/O-EFA model), and so can be used to identify the edges. For example \( \delta_1 \) and \( \delta_2 \) correspond to execution of edges \( e_{22} \) and \( e_{41} \) respectively.

It is evident that data-logging introduces a partial observation of variables, and further the partial observation is state or edge-dependent (since recordings at each edge may be different). We use edge-dependent observation functions \( \{ M_e | e \in E \} \) to capture the partial observability introduced by a certain data-logging scheme. We use \( M \) to denote the above collection of observation functions. For a run

\[
\begin{align*}
 r &= (l_0, d_0) \xrightarrow{\tilde{a}_0/\tilde{y}_0; \tilde{\sigma}_0/\delta_0} \cdots \xrightarrow{\tilde{a}_n/\tilde{y}_n; \tilde{\sigma}_n/\delta_n} (l_{n+1}, d_n),
\end{align*}
\]

let \( \pi^r = e_0 \cdots e_n \) be the path associated with \( r \). Then the run \( r \) is as follows:

\[
M(r) = M_{e_0}(e_0; d_0; \tilde{u}_0/\tilde{y}_0; \tilde{\sigma}_0/\delta_0) \cdots M_{e_n}(e_n; d_n; \tilde{u}_n/\tilde{y}_n; \tilde{\sigma}_n/\delta_n).
\]

The partial observation due to data-logging introduces an indistinguishability among the runs: Two runs \( r_1 \) and \( r_2 \) are indistinguishable if their recorded values are the same: \( M(r_1) = M(r_2) \). The set of all runs indistinguishable from a run \( r \) are given by:

\[
M^{-1}M(r) = \{ r' | M(r') = M(r) \}.
\]

Thus when a software executes a faulty-run \( r \) (that violates a certain specification), set of all run-indistinguishable runs in \( M^{-1}M(r) \) must be examined to identify a fault-seed. To obtain the set of run-indistinguishable runs in \( M^{-1}M(r) \), all one needs is first build a I/O-EFA model for \( r \) (which is simply a chain of edges representing the run) and next “\( M \)-synchronize” the model with \( P \). Then the set of all runs of the resulting model is same as \( M^{-1}M(r) \).

Definition 3: The \( M \)-synchronous composition of \( P_i = (L_i, D_i, U_i, Y_i, \Sigma_i, \Delta_i, E_i, L_0, D_0) \), \( i = 1, 2 \) is given by,

\[
P_1 || M P_2 := (L_1 \times L_2, D_1 \times D_2, U_1 \times U_2, Y_1 \times Y_2, \Sigma_1 \times \Sigma_2, \Delta_1 \times \Delta_2, E, L_0 \times L_0, D_0 \times D_0),
\]

where each \( e \in E \) is a tuple \( e = (\sigma_e, t_e, \tilde{d}_e) \) such that each of the following three cases hold:

\[
\begin{align*}
\exists e_i &= (\sigma_i, t_e, \tilde{d}_e) \in E_i (i = 1, 2) : \\
o_e &= (\sigma_e, \tilde{d}_e), \ t_e = (t_e_1, t_e_2),
\end{align*}
\]

\[
\text{pred}_e(\tilde{v}) = \text{pred}_{e_1}(\tilde{v}_1) \land \text{pred}_{e_2}(\tilde{v}_2) \land [M_{e_1}(v_1) = M_{e_2}(v_2) \neq e],
\]

\[
\exists e_{x_1} \in L_1, e_1 = (\sigma_1, t_e, \tilde{d}_e_1) \in E_1 : \\
o_e = (\sigma_1, \tilde{d}_e), \ t_e = (t_e_1, t_e_2),
\]
Propositionally-label the next edge of the fragment. The counter-augmented 
fragment) or reset (if the next edge taken influences the fragment).
The synchronous composition computes \( \pi \) fragment in as:
atomic proposition shown in Figure 3.
location of the type (m, k), \( e \neq e_{ik} \wedge k = m \):
if \( e = e_{ik} \wedge k < m \),
if \( (o_{l}, e_{l}, k + 1), \sigma_{e}, \delta_{e}, \sigma_{e} \wedge \delta_{e}, f_{e}, h_{e}) \in E^{f} \)
if \( (o_{l}, e_{l}, k), \sigma_{e}, \delta_{e}, f_{e}, h_{e}) \in E^{f} \)
if \( e = e_{ik} \wedge |l| \leq k \): \( e \neq e_{il} \) or \( k = m \),
if \( e = e_{ik} \wedge k \leq k \): \( e \neq e_{il} \) or \( k = m \).
\( \exists \pi \in \Pi^{m} \) of the path \( \pi^{f} = e_{1}e_{2}e_{3}e_{41}e_{51}e_{6} \in \Pi^{m} \). The I/O-EFA models for \( \pi^{f} \) and \( P^{f} \) are as shown in Figure 2.
5) **Refine counter-augment** \( \bar{P} \) wrt specification \( R \), i.e., compute \( P^{f} \parallel R \).  
Example 4: The synchronous composition \( P^{f} \parallel R \) is as shown in Figure 3.
6) **Propositionally-label** \( P^{f} \parallel R \) and **model-check** \( \pi^{f} \) for fault-seed:
Label a location in \( P^{f} \parallel R \) by an atomic proposition \( m \), if the counter reads \( m \) (this is a location of the type \((-, m, -) \in P^{f} \parallel R \)); and by an atomic proposition \( f \), if the monitor is in the FAULT location (this is a location of the type \((-, -, \text{FAULT}) \) in \( P^{f} \parallel R \)). Use model-checking to verify the following CTL formula:
\[
P^{f} \parallel R \models [EF m \wedge AG(m \Rightarrow AF f)].
\]
In other words, “Exists a run that fully executes the fragment \( \pi^{f} \) (as captured by \( EF m \))” and “For all runs always if \( \pi^{f} \) is a fragment, then for all subsequent runs eventually fault occurs (as captured by \( AG(m \Rightarrow AF f) \)).” \( \pi^{f} \) is a fault-seed if and only if the answer to the model-checking problem is affirmative.) Stop if the answer is yes, else remove \( \pi^{f} \) from \( \Pi^{m} \) and go back to step 2.

**Example 5**: It is immediate to verify that
\[
P^{f} \parallel R \models [EF 1 \wedge AG(1 \Rightarrow AF 1)]
\]
holds, concluding that \( \pi^{f} \) is a fault-seed. Since \( e_{41} \), the 4th statement of the min/max program, is itself faulty where “max” has been replaced with “min”, our localization approach has been able to identify the exact faulty statement. In contrast, an approach based on path-slicing will present the entire faulty-run as a fault-seed since the sequence of statements executed in the above faulty-run form a chain of causally-dependent statements. On the other hand, the approaches based on comparing a faulty-run versus a nonfaulty-run are not applicable here since a nonfaulty-run is not available.

The following theorem follows from the definition of fault-seed and the above algorithm.

**Theorem 1**: Consider a software model \( P \), a specification \( R \), a faulty-run \( r \), and an observation function \( M \) (as induced by data-logging). A fragment \( \pi^{f} \) of a path in \( \Pi^{m} \) (the set of all paths associated with the \( r \)-indistinguishable runs) is a fault-seed if and only if
\[
P^{f} \parallel R \models [EF m \wedge AG(m \Rightarrow AF f)].
\]

**Proof**: The proof argument is based on the correctness of each step. Step 1 correctly constructs the set of all \( r \)-indistinguishable runs, all of whose fragments must be examined for a fault-seed, and steps 2 and 3 are just initialization and continuation/termination steps. Step 4 correctly tracks the length of a fragment \( \pi^{f} \in \Pi^{m} \subset \Pi^{m} \) executed so far: In the first clause, the counter is incremented by one if the executed edge of \( P \) is the same as the next edge of the fragment that is pending to be executed, and the counter hasn’t reached its maximum. In the second clause, the counter does not change if the executed edge of \( P \) differs from the next edge of the fragment that is pending.
to be executed and does not influence any of the edges of the fragment pending to be executed, or if the counter has reached its maximum. In the third clause, the counter resets if the executed edge of $P$ differs from the next edge of the fragment that is pending to be executed and influences an edge of the fragment pending to be executed, and the counter has not reached its maximum.

The refinement step 5 is correct (the refined model $P' \parallel R$ has the same set of runs as the model $P'$ since by construction, at every location of a specification monitor $R$, the disjunct over all outgoing edges of the guard conditions is true. Thus, in step 5, $P' \parallel R$ evolves whenever $P'$ evolves, and $m$ is reached in $P' \parallel R$ if and only if $m$ is reached in $P'$.

The model-checking formula in step 6 requires that there exists at least one run in which the fragment $\pi'$ is fully executed as an influence-fragment (otherwise $\pi'$ is not to be regarded as a candidate for a fault-seed) as captured for all runs in $P'. Therefore, every location of this specification monitor is reached in all subsequent runs, eventually $\text{FAULT}$ is reached in the monitor (i.e., the specification is eventually violated). Note from the construction of $P' \parallel R$, if the specification is violated along one run, say $r$, then it is also violated along all runs associated with $\pi'$ (since if $r$ reaches a location $(\neg, \neg$, FAULT), then any other run associated with $\pi'$ also reaches the same location $(\neg, \neg$, FAULT)). Thus requiring “in all subsequent runs, eventually the FAULT location is reached in the monitor” is equivalent to “all subsequent paths eventually have an associated run such that the location FAULT is reached in the monitor” (which agrees with the definition of a fault-seed, as desired).

Remark 1: Note the verification of the CTL formula is polynomial in the size of the model $P' \parallel R$ as well as the length of the CTL formal (namely, the number of operators in the formula). Since the above CTL formula is short in length, the complexity is mainly determined by the size of the model. Further a root-cause is typically limited to a few lines of code, and so in practical setting it is expected that only small length fragments need to be examined to identify a fault-seed. Standard approaches such as symbolic model-checking [4] or bounded model-checking [6] or ranking abstraction [14] can be used to further reduce the complexity of model-checking. Also note that those portions of the fragment that do not affect the variables present in the specification being violated, can be pruned out to reduce the search space. I.e., our approach allows incorporation of the path-slicing technique for further complexity improvement. Further if a comparative study between faulty versus non-faulty runs is available, then the fragments unique to faulty-runs (that aren’t present in the non-faulty runs) can be treated as the candidates for the fault-seeds, and our model-checking approach can be utilized to confirm whether the chosen candidates are indeed the fault-seeds. Thus our approach can be utilized in conjunction with the path-slicing or comparative approaches to achieve finer resolution in fault-identification.

V. ILLUSTRATIVE EXAMPLE

In this section we present an example in which the fault is due to a missing line of code. A missing line of code cannot be directly identified but as the example shows, in this case, our approach identifies a fault-seed whose execution is essential for the manifestation of a failure, caused by the missing line of code. Thus when a fault is due to missing lines of code, a fault-seed identified by our approach serves as an indicator for the missing lines of code, and hence we refer to it as a fault-indicator. In this manner our approach helps with fault-localization even when the root-cause is missing lines of code.

Consider an embedded software that allows users to monitor whether the temperature of a device exceeds its critical value. A user can enable and disable the monitoring process by pressing a key on a keyboard. It is desired that if the monitoring is enabled, then if the temperature is higher (lower) than a critical value $T_{c}$ (a critical value minus a hysteretic value $Hys$), the alarm is On (Off); and if the monitoring is disabled, then the alarm is Off. A temperature sensor is connected to the analog channel ‘Channel’ of an embedded controller and its value is read by the function $\text{analogIn}$. The function $\text{Kelvin}$ converts the reading into the scale of Kelvin. An alarm is connected to the pin $\text{Bit}$ of the port $\text{Port}$ of the embedded controller and set On/Off by the function $\text{BitWrPort}$. The key pressing is detected by the function $\text{kbhit}$. If key “1” (“2”) is pressed, then the monitoring is enabled (or disabled). The functions $\text{analogIn}$, $\text{BitWrPort}$, $\text{Kelvin}$, $\text{kbhit}$ are provided within a software-library and we assume those are correct. A variable $\text{SensorArmed}$ is used so that the alarm is set On for only once when the temperature is above $T_{c}$. The function $\text{Record}$ is included for data-logging purpose.

```c
1 brdInit(); key = '0';
   EnableC = False; SensorArmed = True;
   BitWrPort(Port,Bit,0); // AlarmStatus = Off;
2 while (1) {
   if (kbhit()) { key = getchar();
      Record(\delta_1,key); }
3   if (key == '1' || key == '2') {
4      if (key == '1') EnableC = True;
5      if (key == '2') EnableC = False;
6      rawTemp = analogIn(Channel);
7      if (Tc >= Thc) {
8         if (EnableC && SensorArmed) {
            BitWrPort(Port,Bit,1); // AlarmStatus = On;
            SensorArmed = False;
            Record(\delta_2); }
9        else Record(\delta_3); }
10       if (Tc < Thc-Hys) {
11          BitWrPort(Port,Bit,0); // AlarmStatus = Off;
12          SensorArmed = True;
13          Record(\delta_4); }
    }
```

Assume the line of code, “$\text{SensorArmed} = \text{True}$;”, written in bold, is missing. For simplicity, we replace the code $\text{BitWrPort(Port,Bit,1)}$ (resp., $\text{BitWrPort(Port,Bit,0)}$) by $\text{AlarmStatus} = \text{On}$ (resp., $\text{AlarmStatus} = \text{Off}$), and let the specification be $G[\text{key} = '1' \land Tc > Thc] \Rightarrow X^{\leq 2}(\text{AlarmStatus} = \text{On})$, which states that, “whenever monitor is enabled (key='1') and temperature is above
threshold, in next two steps alarm should be set On". The I/O-EFA models of the program and its specification monitor are shown in Figure 4.

When key ‘1’ is pressed, a violation scenario of the specification is shown in Figure 5, where the violation occurs at the sample time t(4). Owing to the missing line of code, the alarm is Off when Tc>ThC at t(4).

The following faulty run is recorded:
\[ M(r) = (\delta_1 \| key = '1' \| Tc > ThC - Hys) \] where the labels \( \delta_1, \delta_2, \delta_3, \delta_4 \) correspond to the execution of edges \( e_{21}, e_{81}, e_{72}, e_{91} \) respectively.

One of the faulty-paths associated with the runs in \( M^{-1} M(r) \) is given by:
\[ \pi = e_{1} e_{21} e_{31} e_{41} e_{52} e_{6} e_{72} e_{91} e_{22} e_{31} e_{41} e_{52} e_{6} e_{71} e_{81} \]
\[ e_{92} e_{22} e_{31} e_{41} e_{52} e_{6} e_{72} e_{91} e_{22} e_{31} e_{41} e_{52} e_{6} e_{71} e_{82} \]
\[ e_{92} e_{22} \in \Pi^r. \]

Consider \( m = 2 \) and select the fragment \( \pi^f = e_{41} e_{81} \in \Pi^r. \) of path \( \pi^r. \) The I/O-EFA model of \( \pi^f \) and a portion of \( P^f \) are shown in Figure 6. To simplify the figure, we only show the edge names as the corresponding guard, data update and output assignment are known to us and can be omitted. A portion of the synchronous composition \( P^f \| R \) is shown in Figure 7. It can be verified that
\[ P^f \| R \models [EF2 \land AG(2 \Rightarrow AF f)] \]
holds, concluding that \( \pi^f \) is a fault-seed. An examination of the lines of code corresponding to \( \pi^f \) indicates no fault. This implies \( \pi^f \) is an indicator for certain missing lines of code. Such information about indication of missing lines of code is as useful to a debugger as a report that locates those lines of code that are themselves faulty.

VI. Conclusion

We proposed an approach for localizing a fault detected during the runtime operation of software (refer to our prior work on fault detection [19]). We presented a model-based approach for fault localization that is based on the notion of a fault-seed, generalizing the approach proposed in [10]. Checking whether a fragment, of a partially observed faulty-run recorded during runtime operation of the software, is a fault-seed (i.e., a plausible root-cause of the observed failure) was formulated as a model-checking problem. It is desirable to find a minimal length fault-seed, and this is guaranteed by our approach since only the fragments smaller or equal in length to a shortest fault-seed are examined. The complexity of checking whether a candidate fragment is a fault-seed is polynomial in the size of the software and the specification. Also since in practice a root-cause is confined to a few lines of code, only a small number of fragments, not exceeding in length of a root-cause, are checked for candidacy to a fault-seed. As noted at the end of introduction as well as in Remark 1, our approach can be used to complement and supplement the existing approaches (path-slicing or comparison of faulty versus non-faulty runs).
to achieve a finer resolution in fault-localization. Also the proposed approach is helpful in localizing faulty lines of code or an indicator for missing lines of code (as the case may be), and works with partial observations of a run. In the future work we will pursue implementation and empirical studies of the approach presented in the paper. Another research topic of interest is deciding what data values to record and at which program locations so that fault-detection as well as fault-localization can be performed—Recording less data is helpful as it minimizes the data-logging overhead, but it results in a larger search space for the faulty-seeds, and worse, it can even mask a faulty run from a non-faulty one. Deciding which data variables to record is like the sensor selection problem [11], and a similar approach can be developed.

REFERENCES


