2011 American Control Conference
on O'Farrell Street, San Francisco, CA, USA
June 29 - July 01, 2011

Analysis of Runtime Data-Log for Software Fault Localization

Changyan Zhou, Ratnesh Kumar, and Shengbing Jiang

Abstract— **Software can contain faults that remain undetected prior to its release. It is then important to determine the plausible root-cause of fault, 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. 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. A feature of our analysis is to localize the possible root cause in presence of such partial observability of data variables.**

I. Introduction

In spite of the tremendous progress made in software testing and verification, a software can contain faults. 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 [10]. In [15], 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. 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.*

The approaches proposed in prior works such as [11], [9], [2], [13], [6], [4], [5] have the limitation that they require having nonfaulty-runs for comparing with the available faulty-run. Also such comparisons do not necessarily yield the root cause. Slicing based approaches are investigated in [1], [14], [3], [7], [12] that slice-out non relevant portions (such as those lacking variable dependencies).

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: A subset of statements included in a faulty-run is called a fault-seed if their influencive execution in *any* run of the software causes a failure (a certain specification violation) to occur. 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.

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=u1; max=u1; y1=y2=0;
2   if (max < u2) max = u2;
3   if (max < u3) max= u3;
4   if (min > u2) max= u2;
5   if (min > u3) min= u3;
6   y1= min; y2= 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: $(0,0) \rightarrow (0,0) \rightarrow (0,0) \rightarrow (0,0) \rightarrow (0,0) \rightarrow (5,1)$. 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.*

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 (Σ), discrete (symbolic) outputs (Δ), 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 edges.) The occurrence of a transition triggers a data update and an output assignment. An 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,
- Σ is the set of discrete (symbolic) inputs,
- Δ is the set of discrete (symbolic) outputs,
- E is the set of edges, and each $e \in E$ is a 7-tuple, $e = (o_e, t_e, \sigma_e, \delta_e, G_e, f_e, h_e)$, where
  - $o_e \in L$ is the origin location,
  - $t_e \in L$ is the terminal location,
  - $\sigma_e \in \Sigma \cup \{\epsilon\}$ is the discrete input,
  - $\delta_e \in \Delta \cup \{\epsilon\}$ is the discrete output,

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

  - $G_e \subseteq D \times U$ is the enabling guard (a predicate),
  - $f_e : D \times U \to D$ is the data update function,
  - $h_e : D \times U \to Y$ 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. ■

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, $y_1 \le y_2$, written in temporal logic as $G[y_1 \le y_2]$ (i.e, for all paths, always $y_1 \le y_2$). Reaching the FAULT location in R implies the violation of the specification.

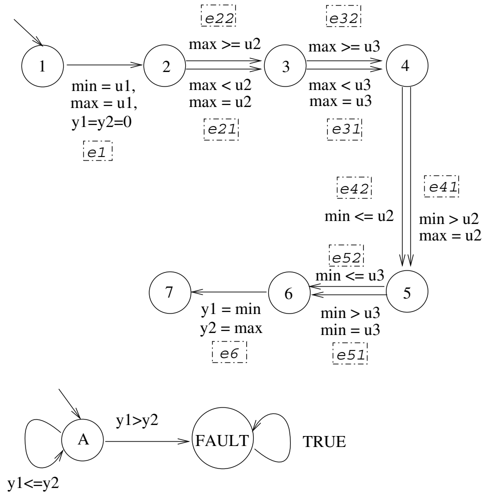

Fig. 1. I/O-EFA model P of min/max program (top) and specification monitor R (bottom)

III. Fault-Seed and Fault-Fragment

A *fault-seed* is used to characterize a plausible root-cause of a specification violation, namely, either faulty or missing code. A fault-seed is an ordered set of statements (i.e., an ordered set of edges of I/O-EFA model) such that any sequence of statements of the software, containing the fault-seed in an influencive manner, can eventually perform a violating computation through the additional execution of the statements. (Note since the fault may not manifest immediately after the execution of a fault-seed, extensions are considered). We start by defining a *step*, which is simply a single state transition.

Definition 2: A *step* of an I/O-EFA P is given by, $(l, \vec{d}) \xrightarrow{\vec{u}/\vec{y};\sigma/\delta} (l', \vec{d}')$, where $l, l' \in L, \vec{d}, \vec{d}' \in D, \vec{u} \in U, \vec{y} \in Y$, and exists $e = (o_e, t_e, \sigma_e, \delta_e, G_e, f_e, h_e) \in E$ such that

$$[o_e = l, t_e = l', \sigma_e = \sigma, \delta_e = \delta] \wedge$$
$$G_e(\vec{d}, \vec{u}) \wedge [f_e(\vec{d}, \vec{u}) = \vec{d}'] \wedge [h_e(\vec{d}, \vec{u}) = \vec{y}].$$

■

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.

Definition 3: A *run* r of an I/O-EFA P is a finite sequence of steps starting from an initial state:

$$r := (l_0, \vec{d}_0) \xrightarrow{\vec{u}_0/\vec{y}_0;\sigma_0/\delta_0} \ldots \xrightarrow{\vec{u}_n/\vec{y}_n;\ \sigma_n/\delta_n} (l_{n+1}, \vec{d}_{n+1}),$$

where $l_0 \in L_0$, $\vec{d}_0 \in D_0$ and for $j \in \{0, \ldots, n\}$, $(l_j, \vec{d}_j) \xrightarrow{\vec{u}_j/\vec{y}_j;\sigma_j/\delta_j} (l_{j+1}, \vec{d}_{j+1})$ is a step. ■

Definition 4: For a run r, we say a *path* $\pi^r = e_0 \cdots e_n$ (a sequence of edges of P) is *associated with the run* r if

$$\forall 0 \le i \le n : [o_{e_i} = l_i, t_{e_i} = l_{i+1}, \sigma_{e_i} = \sigma_i, \delta_{e_i} = \delta_i] \wedge$$
$$G_{e_i}(\vec{d}_i, \vec{u}_i) \wedge [\vec{d}_{i+1} = f_{e_i}(\vec{d}_i, \vec{u}_i)] \wedge [\vec{y}_i = h_{e_i}(\vec{d}_i, \vec{u}_i)].$$

Dually, we say that a run r is *associated with a path* π of P if $\pi^r = \pi$. We say a sequence of edges $\pi^f = e_{i_0} \cdots e_{i_m}$ is a *fragment* of π^r if $0 \le i_0 \le \cdots \le i_m \le n$. ■

Edges in a path can influence one another. 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.

Definition 5: Given a path $\pi = e_0 \ldots e_n$, we say that an edge e_i *influences* another edge e_j ($i, j \in [0, n]$), denoted $e_i \rightsquigarrow e_j$, if $j > i$ and the output of e_j (as determined by G_{e_j}, f_{e_j}, h_{e_j}, or δ_{e_j}) depends on the output of e_i (as determined via G_{e_i}, f_{e_i}, h_{e_i}, or δ_{e_i}).

A fragment $\pi^f = e_{i_0} \ldots e_{i_m}$ of a path $\pi^r = e_0 \ldots e_n$ is said to be a *influencive-fragment* if for all $k \in [i_0, i_m] - \{i_0, \ldots, i_m\}$, there does not exist $l \in \{i_0, \ldots, i_m\}$ such that $e_k \rightsquigarrow e_l$. ■

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:

$$\begin{aligned} r \quad = \quad & (l_0 = 1, min = -, max = -) \xrightarrow{(5,1,6)/(0,0)} \\ & (2, 5, 5) \xrightarrow{(5,1,6)/(0,0)} (3, 5, 5) \xrightarrow{(5,1,6)/(0,0)} \\ & (4, 5, 6) \xrightarrow{(5,1,6)/(0,0)} (5, 5, 1) \xrightarrow{(5,1,6)/(0,0)} \\ & (6, 5, 1) \xrightarrow{(5,1,6)/(5,1)} (7, 5, 1). \end{aligned}$$

Its associated path is $\pi^r = e_1 e_{22} e_{31} e_{41} e_{51} e_6$, whereas $\pi^f = e_{22} e_{41}$ is a fragment of π^r that can also be seen to be an influencive-fragment since it computes, the value of max, independently of the statements e_1 or e_{31} or e_{32} preceding it. ■

A run r is said to be a *faulty-run* of a certain specification if r violates that specification. A path is said to be a *faulty-path* if it is associated with a faulty-run. A fragment of a faulty-path can be a fault-seed:

Definition 6: A fragment π^f of a faulty-path π^r associated with a faulty-run r is said to be a *fault-seed* if (i) exists a run possessing π^f as an influencive-fragment, and (ii) for any path π, with π^f as an influencive-fragment, all subsequent paths eventually have an associated faulty-run. ■

Note a fault must eventually manifest along each subsequent path for some run in order for π^f to be regarded as a fault-seed.

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 $\vec{v}$ denote the variables

$(\vec{d}, \vec{d}', \vec{u}, \vec{y}, \sigma, \delta) \in D \times D \times U \times Y \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\})$, and define

$$\begin{aligned} pred_e(\vec{v}) &\equiv pred_e(\vec{d}, \vec{d}', \vec{u}, \vec{y}, \sigma, \delta) \\ &:= G_e(\vec{d}, \vec{u}) \wedge [\vec{d}' = f_e(\vec{d}, \vec{u})] \wedge \\ &\quad\ [\vec{y} = h_e(\vec{d}, \vec{u})] \wedge [\sigma = \sigma_e] \wedge [\delta = \delta_e]. \end{aligned}$$

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

Definition 7: The *synchronous composition* of $P_i = (L_i, D_i, U_i, Y_i, \Sigma_i, \Delta_i, E_i, L_{0i}, D_{0i})$, $i = 1, 2$ is given by, $P_1 \| P_2 := (L_1 \times L_2, D_1 \times D_2, U_1 \times U_2, Y_1 \times Y_2, \Sigma_1 \times \Sigma_2, \Delta_1 \times \Delta_2, E, L_{01} \times L_{02}, D_{01} \times D_{02})$, where each edge $e \in E$ is a tuple $e = (o_e, t_e, pred_e)$ such that the following holds:

$$\begin{aligned} &\exists e_i = (o_{e_i}, t_{e_i}, pred_{e_i}) \in E_i (i = 1, 2) : \\ &o_e = (o_{e_1}, o_{e_2}),\ t_e = (t_{e_1}, t_{e_2}), \\ &pred_e(\vec{v}) = pred_{e_1}(\vec{v}_1) \wedge pred_{e_2}(\vec{v}_2) \wedge [v_1 = v_2]. \end{aligned}$$

■

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

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.

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

```
1   min=u1; max=u1; y1=y2=0;
2   if (max < u2) max = u2;
    else record($\delta_1$;y1,y2);
3   if (max < u3) max= u3;
4   if (min > u2) {
      max= u2; record($\delta_2$;y1,y2);
    }
5   if (min > u3) min= u3;
6   y1= min; y2= max;
```

In the above example, the variables y_1 and y_2 are recorded, along with the labels δ_1 and δ_2 as identifiers: The labels are unique to statements, and so can be used to identify the edges. For example δ_1 and δ_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 edge-dependent (since recordings at each edge may be different). We use edge-dependent observation functions $\{M_e \mid e \in E\}$ to capture the partial observability introduced by a certain data-logging scheme. We use $\mathcal{M}$ to denote the above collection of observation functions. For a run

$$r = (l_0, \vec{d}_0) \xrightarrow{\vec{u}_0/\vec{y}_0; \sigma_0/\delta_0} \cdots \xrightarrow{\vec{u}_n/\vec{y}_n;\ \sigma_n/\delta_n} (l_{n+1}, \vec{d}_{n+1}),$$

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

$$\begin{aligned} \mathcal{M}(r) \ =\ & M_{e_0}(e_0;\ d_0;\ \vec{u}_0/\vec{y}_0;\ \sigma_0/\delta_0) \\ & \ldots \quad M_{e_n}(e_n;\ d_n;\ \vec{u}_n/\vec{y}_n;\ \sigma_n/\delta_n). \end{aligned}$$

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: $\mathcal{M}(r_1) = \mathcal{M}(r_2)$. The set of all runs indistinguishable from a run r are given by, $\mathcal{M}^{-1}\mathcal{M}(r) := \{r' \mid \mathcal{M}(r') = \mathcal{M}(r)\}$. Thus when a software executes a faulty-run r (that violates a certain specification), set of all r-indistinguishable runs in $\mathcal{M}^{-1}\mathcal{M}(r)$ must be examined to identify a fault-seed. To obtain the set of r-indistinguishable runs in $\mathcal{M}^{-1}\mathcal{M}(r)$, 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 "$\mathcal{M}$-synchronize" the model with P.

Definition 8: The $\mathcal{M}$-synchronous composition of $P_i = (L_i, D_i, U_i, Y_i, \Sigma_i, \Delta_i, E_i, L_{0i}, D_{0i})$, $i = 1, 2$ is given by, $P_1 \|_{\mathcal{M}} P_2 := (L_1 \times L_2, D_1 \times D_2, U_1 \times U_2, Y_1 \times Y_2, \Sigma_1 \times \Sigma_2, \Delta_1 \times \Delta_2, E, L_{01} \times L_{02}, D_{01} \times D_{02})$, where each $e \in E$ is a tuple $e = (o_e, t_e, pred_e)$ such that either of the following three cases hold:

- $\exists e_i = (o_{e_i}, t_{e_i}, pred_{e_i}) \in E_i (i = 1, 2)$: $o_e = (o_{e_1}, o_{e_2})$, $t_e = (t_{e_1}, t_{e_2})$, $pred_e(\vec{v}) = pred_{e_1}(\vec{v}_1) \wedge pred_{e_2}(\vec{v}_2) \wedge [M_{e_1}(v_1) = M_{e_2}(v_2) \neq \epsilon]$;
- $\exists o_{e_2} \in L_2, e_1 = (o_{e_1}, t_{e_1}, pred_{e_1}) \in E_1$: $o_e = (o_{e_1}, o_{e_2})$, $t_e = (t_{e_1}, o_{e_2})$, $pred_e(\vec{v}) = pred_{e_1}(\vec{v}_1) \wedge [M_{e_1}(v_1) = \epsilon]$;
- $\exists o_{e_1} \in L_1, e_2 = (o_{e_2}, t_{e_2}, pred_{e_2}) \in E_2$: $o_e = (o_{e_1}, o_{e_2})$, $t_e = (o_{e_1}, t_{e_2})$, $pred_e(\vec{v}) = pred_{e_2}(\vec{v}_2) \wedge [M_{e_2}(v_2) = \epsilon]$. ■

According to the definition, two edges of P_1 and P_2 synchronize if and only if both are enabled and agree on the observed values of all the variables. An edge of P_1 (resp. of P_2) is executed asynchronously without the participation of P_2 (resp., P_1) if the edge is enabled in P_1 (resp., P_2) and all the variables of P_1 (resp., P_2) are not observed (so the observed value equals ϵ).

Next we describe a model-based approach for fault localization as an instance of a model-checking problem.

Algorithm 1: Given a software modeled as P, its specification modeled as R, a faulty-run r (recorded as $\mathcal{M}(r)$), our algorithm for identifying a fault-seed is given as follows.

1) **Compute r-indistinguishable paths:** Identify the set of all paths Π^r in P associated with r-indistinguishable runs: $\Pi^r := \{\pi \mid \exists$ faulty $r' \in \mathcal{M}^{-1}\mathcal{M}(r)$ s.t. $\pi^{r'} = \pi\}$. Let P^r denote an I/O-EFA model of r, then Π^r is simply the set of paths in $(P\|R)\|_{\mathcal{M}} P^r$ that reach a location $(-, \text{FAULT}, -)$. Since there are no unbounded sequence of unobserved steps possible, Π^r is finite.

   Example 2: To illustrate our fault localization approach outlined above, we revisit the Example 1. As discussed in the introduction, the specification $G[y_1 \leq y_2]$ is violated for this input $(u_1, u_2, u_3) = (5, 1, 6)$, and the faulty-run is also mentioned in the introduction and formalized in Example 1.

   For the data-logging mentioned above that records the variables y_1 and y_2 following the execution of the edges e_{22} (recorded as δ_1 label) and e_{41} (recorded as δ_2 label), the following faulty-run is recorded: $\mathcal{M}(r) = (\delta_1;\ y_1 = 0;\ y_2 = 0)(\delta_2;\ y_1 = 0;\ y_2 = 0)$.

   The set of all faulty-paths Π^r associated with the r-indistinguishable runs are given by:

$$\begin{aligned} \Pi^r &= \{e_1 e_{22}(e_{31} + e_{32}) e_{41} (e_{51} + e_{52}) e_6\} \\ &= \{e_1 e_{22} e_{31} e_{41} e_{51} e_6, e_1 e_{22} e_{31} e_{41} e_{52} e_6, \end{aligned}$$

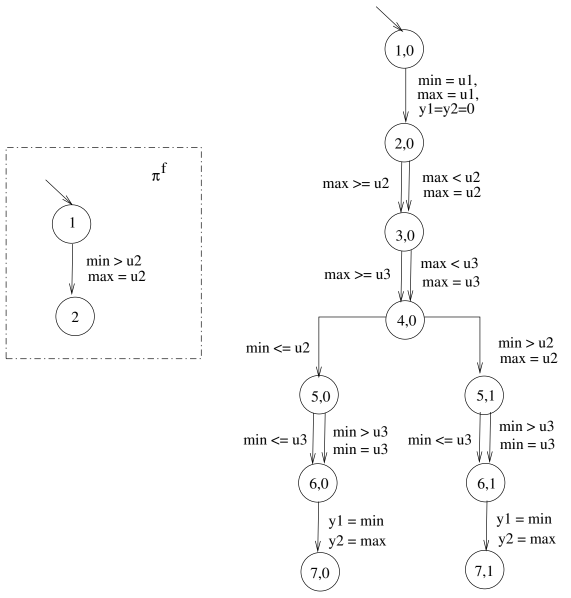

Fig. 2. π^f (left) and P^f (right)

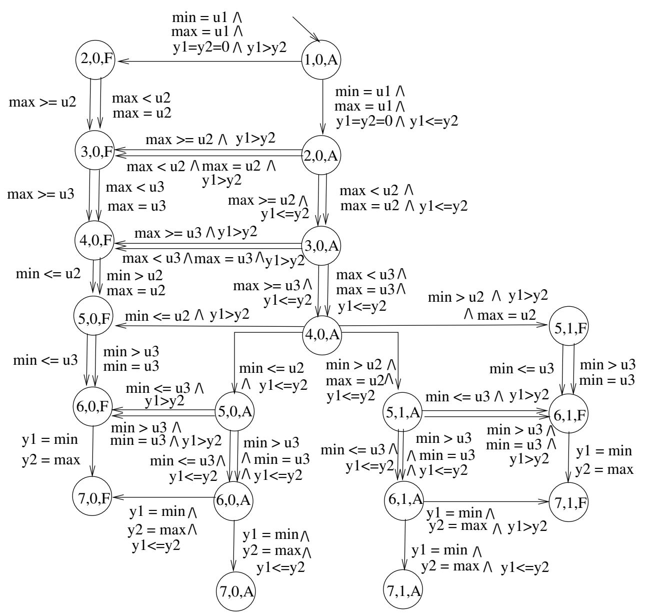

Fig. 3. $P^f \| R$

$$e_1 e_{22} e_{32} e_{41} e_{51} e_6, e_1 e_{22} e_{32} e_{41} e_{52} e_6\}.$$

■

2) **Initialize length for fragments to be checked:** Set a count value $m = 1$, and for all paths in Π^r, obtain the set of all fragments of length m, denoted $\Pi^{r,m}$.
3) **Check non-emptiness of set of fragments of a chosen length:** Proceed to next step if $\Pi^{r,m}$ is nonempty, else set $m := m + 1$ and if m exceeds the length of the faulty-run, then stop, else obtain $\Pi^{r,m}$ and repeat step 3.
4) **Augment P with a counter to track length of fragment-prefix executed so far:** Pick a length-m fragment $\pi^f \in \Pi^{r,m}$, and augment P with a counter of maximum count value m to keep track of the length of the fragment-prefix executed, in an influencive manner, so far: Whenever a next edge taken of P is also a next edge of the fragment, the counter is incremented by one, and otherwise, the counter is either unchanged (if the next edge taken of P does not influence the fragment) or reset (if the next edge taken influences the fragment).
   The counter-augmented P is denoted P^f and is defined as: $P^f = (L \times \{0, \cdots, m\}, D, U, Y, \Sigma, \Delta, E^f, L_0 \times \{0\}, D_0)$, where $\forall e \in E, \forall k \in \{0, \ldots, m\}$:
   $$\begin{aligned} &((o_e, k), (t_e, k+1), \sigma_e, \delta_e, G_e, f_e, h_e) \in E^f \\ &\text{if} \quad [e = e_{i_k}] \wedge [k < m], \\ &((o_e, k), (t_e, k), \sigma_e, \delta_e, G_e, f_e, h_e) \in E^f \\ &\text{if} \quad [e \neq e_{i_k} \wedge \forall l \in [k, m] : e \not\rightsquigarrow e_{i_l}] \vee [k = m], \\ &((o_e, k), (t_e, 0), \sigma_e, \delta_e, G_e, f_e, h_e) \in E^f \\ &\text{if} \quad [e \neq e_{i_k} \wedge \exists l \in [k, m] : e \rightsquigarrow e_{i_l}] \wedge [k < m]. \end{aligned}$$
   Example 3: For $m = 1$, suppose we select the fragment $\pi^f = e_{41} \in \Pi^{r,1}$ of the path $\pi^r = e_1 e_{22} e_{31} e_{41} e_{51} e_6 \in \Pi^r$. The I/O-EFA models for π^f and P^f are as shown in Figure 2. ■
5) **Refine counter-augment P wrt specification R,** i.e., compute $P^f \| R$.
   Example 4: The synchronous composition $P^f \| R$ is as shown in Figure 3. ■
6) **Propositionally-label $P^f \| R$ and model-check π^f for fault-seed:** Label a location in $P^f \| R$ by an atomic proposition m, if the counter reads m (this is a location of the type $(-, m, -)$ in $P^f \| R$); and by an atomic proposition f, if the monitor is in the FAULT location (this is a location of the type $(-, -,$FAULT$)$ in $P^f \| R$). Use model-checking to verify the following CTL formula:
   $$P^f \| R \models [EFm \wedge AG(m \Rightarrow AFf)].$$
   In other words, "Exists a run that fully executes the fragment π^f (as captured by EFm)" and "For all runs always if π^f is a fragment, then for all subsequent runs eventually fault occurs (as captured by $AG(m \Rightarrow AFf)$)". (π^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 π^f from $\Pi^{r,m}$ and go back to step 3.
   Example 5: It is immediate to verify that
   $$P^f \| R \models [EF1 \wedge AG(1 \Rightarrow AFf)]$$
   holds, concluding that π^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. ■

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 $\mathcal{M}$ (as induced by data-logging). A fragment π^f of a path in Π^r (the set of all paths associated with the r-indistinguishable runs) is a fault-seed if and only if

$P^f \| R \models [EFm \wedge AG(m \Rightarrow AFf)]$. ■

Remark 1: Note the verification of the CTL formula is polynomial in the size of the model $P^f \| R$ as well as the

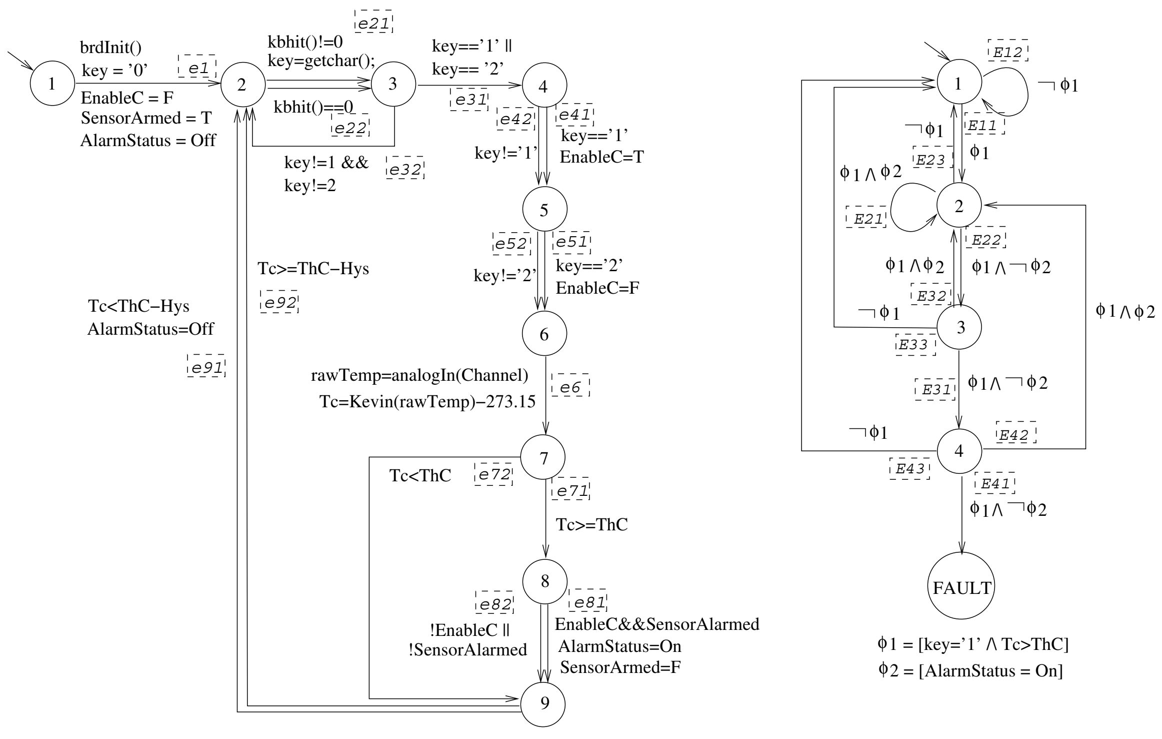

Fig. 4. P (left) and R (right)

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. 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. 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

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 `ThC` (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 `analogIn`. The function `Kelvin` converts the reading into the scale of Kelvin. An alarm is connected to the pin `Bit` of the port `Port` of the embedded controller and set On/Off by the function `BitWrPort`. The key pressing is detected by the function `kbhit`. If key "1" ("2") is pressed, then the monitoring is enabled (or disabled). The functions `analogIn,BitWrPort,Kelvin,kbhit` are provided within a software-library and we assume those are correct. A variable `SensorArmed` is used so that the alarm is set On for only once when the temperature is above `ThC`. The function `Record` is included for data-logging purpose.

```
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);
         Tc = Kelvin(rawTemp) − 273.15;
7         if (Tc $\geq$ ThC) {
8            if (EnableC && SensorArmed) {
                 BitWrPort(Port,Bit,1);// AlarmStatus = On;
             SensorArmed = False;
             Record($\delta_2$);}
         }
         else Record($\delta_3$);
9         if (Tc < ThC-Hys) {
             BitWrPort(Port,Bit,0);   // AlarmStatus = Off;
           **SensorArmed = True;**
           Record($\delta_4$);
         }
       }
   }
```

Assume the line of code, "**SensorArmed = True;**", written in bold, is missing. For simplicity, we replace the code BitWrPort(Port,Bit,1) (resp., BitWrPort(Port,Bit,0)) by AlarmStatus = On (resp., AlarmStatus = Off), and let the specification be $G[(\text{key} = \text{`1'} \wedge \text{Tc} \geq \text{ThC}) \Rightarrow X^{\leq 2}(\text{AlarmStatus = On})]$, 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 $t(4)$.

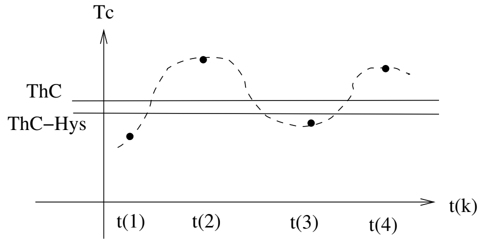

Fig. 5. A violation scenario of R

The following faulty-run is recorded:
$\mathcal{M}(r) = (\delta_1; key = \text{`1'})(\delta_3)(\delta_4)(\delta_2)(\delta_3)(\delta_4)$, 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 $\mathcal{M}^{-1}\mathcal{M}(r)$ is given by:

$$\begin{aligned}\pi \quad = \quad & e_1 e_{21} e_{31} e_{41} e_{52} e_6 e_{72} e_{91} e_{22} e_{31} e_{41} e_{52} e_6 e_{71} e_{81} \\ & e_{92} e_{22} e_{31} e_{41} e_{52} e_6 e_{72} e_{91} e_{22} e_{31} e_{41} e_{52} e_6 e_{71} e_{82} \\ & e_{92} e_{22} \in \Pi^r.\end{aligned}$$

Consider $m = 2$ and select the fragment $\pi^f = e_{41} e_{81} \in \Pi^{r,2}$ of path π^r. A portion of the synchronous composition $P^f \| R$ is shown in Figure 6. It can be verified that

$$P^f \| R \models [EF2 \wedge AG(2 \Rightarrow AFf)]$$

holds, concluding that π^f is a fault-seed. An examination of the lines of code corresponding to π^f indicates no fault. This implies π^f is an *indicator* for certain missing lines of code.

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 [15]). We presented a model-based approach for fault localization that is based on the notion of a fault-seed, generalizing the approach proposed in [8]. 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. 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.

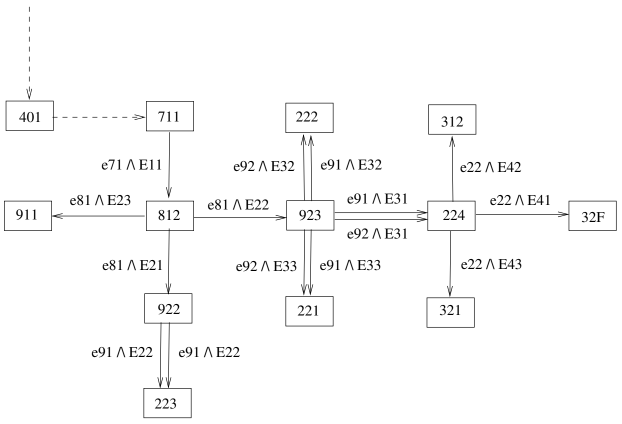

Fig. 6. A portion of $P^f \| R$

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990.

[2] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–105, New York, NY, USA, 2003. ACM.

[3] Samik Basu, Diptikalyan Saha, and Scott A. Smolka. Localizing programs errors for cimple debugging. In *International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 3235, pages 79–96, Madrid, Spain, September 2004. Springer-Verlag.

[4] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. *SIGSOFT Softw. Eng. Notes*, 29(6):73–82, 2004.

[5] A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2004.

[6] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *CC*, pages 80–95, 2006.

[7] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the 27th Annual ACM Conference on Programming Language Design and Implementation*, pages 38–47, 2005.

[8] S. Jiang, T.E. Fuhrman, and S.K. Jha. Model checking for fault explanation. In *Decision and Control, 2006 45th IEEE Conference on*, pages 404–409, San Diego, CA, Dec 2006.

[9] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, NY, USA, 2005. ACM.

[10] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of jml accommodates both runtime assertion checking and formal verification. *LNCS: Formal Methods for Components and Objects*, 2852:262–284, 2003.

[11] M. Renieres and S.P. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. 18th IEEE International Conference on*, pages 30– 39, Oct 2003.

[12] C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? causal analysis for counterexamples. In *ATVA*, pages 82–95, 2006.

[13] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 347–351, New York, NY, USA, 2005. ACM.

[14] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[15] C. Zhou, R. Kumar, and S. Jiang. Hierarchical fault detection in embedded control software. *compsac:2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 816–823, 2008.