Hierarchical Fault Detection in Embedded Control Software

C. Zhou and R. Kumar
Dept. of Elec. & Comp. Eng.,
Iowa State Univ., Ames, IA 50011
czhou, rkumar@iastate.edu

S. Jiang
General Motors R&D,
Warren, MI 48090
shengbing.jiang@gm.com

Abstract

We propose a two-tiered hierarchical approach for detecting faults in embedded control software during their runtime operation: The observed behavior is monitored against the appropriate specifications at two different levels, namely, the software level and the controlled-system level. (The additional controlled-system level monitoring safeguards against any possible incompleteness at the software level monitoring.) 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. This is done by tracking the executed behavior in the system model comprising of the models for the software and those for the nonfaulty hardware components: An acceptance by such a model indicates the presence of a software fault. The design of both the software-level and system-level monitors is modular and hence scalable (there exists one monitor for each property), and further the monitors are constructed directly from the property specifications and do not require any software or system model. Such models are required only for the fault isolation step when the detection occurs at the system level. We use input-output extended finite automata (I/O-EFA) for software as well as system level modeling, and also for modeling the property monitors. Note since the control changes only at the discrete times when the system/environment states are sampled, the controlled-system has a discrete-time hybrid dynamics which can be modeled as an I/O-EFA.

1. Introduction

Software deployed in embedded control systems is used for performing computations that are based on the inputs from the underlying “plant” (system under control) and the environment, and the results of such computations are used to affect the plant. Such software are called embedded since they are embedded within a larger system, and are reactive in nature as they react to the plant and the environmental states. Further the embedded control systems exhibit hybrid dynamics, i.e., the continuous dynamics of the underlying plant mixed with the switching logics of the control software.

Both the hardware and the software components of an embedded control system can contain faults. In the hardware components, the faults can develop during their on-line use (due to wear and tear) whereas, any fault in the software components is present even prior to their on-line use. Another difference between hardware and software faults is that while hardware faults can be anticipated and hence modeled, the software faults are unanticipated. Presence of faults in the hardware or the software components can result in the failure of the system. So for safety-critical systems and infrastructure it is important that they have tolerance against such failures. An important first step in tolerance against faults is an on-line monitoring for fault detection and isolation, followed by recovery/reconfiguration procedures for fault mitigation.

There are existing approaches to deal with software faults. One is to attempt to eliminate software bugs to the greatest extent possible by the improvements of the software development processes. Even the most mature software development process however cannot guarantee a zero defect software. Formal verification can prove the correctness of a software with respect to certain formal specifications/properties. There are many limitations though such as, difficulty of the detailed modeling of the software (not just of the functional code but also of the infrastructural code), the complexity (infinite state space leading to undecidability of verification), and the lack of complete set of property specifications. Another approach is to use techniques based on design diversity (such as N-version programming) that utilizes redundancy and majority voting to detect and mitigate software bugs. Such techniques require independent development of design replicas, and are not cost-effective for many of the embedded control applications. Another scheme for software fault tolerance is exception handling [2], in which only a single version of software is used. This approach however does not handle unpredicted exceptions/faults. [1] pre-

*The research was supported in part by the National Science Foundation under the grants NSF-ECCS-0424048, NSF-ECCS-0601570, NSF-ECCS-0801763, and NSF-CCF-0811541.
presented an approach that treats software fault tolerance problem as a robust supervisory control problem. The hypothesized faults are fixed apriori, and any of the unforeseen faults may not be tolerated.

A commonly used approach to detect software faults is to instrument the code by inserting assertions which if violated indicate the presence of a software fault [5]. It is not always clear however, how to instrument a code to check for general safety and liveness properties which depend on the history of evolution and not just the current state (program counter and data values).

We propose a two-tiered hierarchical approach for detecting faults in embedded control software during their run-time operation: The observed behavior is monitored against the appropriate specifications at two different levels, namely, the software level and the controlled-system level. (The additional controlled-system level monitoring safeguards against any possible incompleteness at the software level monitoring.) 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. This is done by tracking the executed behavior in the system model comprising of the models for the software and those for the nonfaulty hardware components: An acceptance by such a model indicates the presence of a software fault. We use input-output extended finite automata (I/O-EFA) for software as well as system level modeling, and also for modeling the property monitors. Note since the control changes only at the discrete times when the system/environment states are sampled, the controlled-system has a discrete-time hybrid dynamics which can also be modeled as an I/O-EFA.

Some of the features of our approach are:

- The design of both the software-level and system-level monitors is modular (there will be a monitor for each property) and hence scalable.
- Monitors can be constructed over atomic propositions appearing in the property specifications, and the details of asserting the atomic propositions becomes an independent concern and can be handled separately.
- The design of monitors does not require the software or the system model (monitors are constructed directly from the property specifications).
- The two-tiered hierarchical monitoring approach safeguards against any possible incompleteness of the software level properties.
- The isolation step requires tracking an observed input-output behavior in the system model without the hardware faults. This can be done without having to compose the individual component models thereby making the approach scalable and suitable for on-line implementation.

2. An Embedded System Example

Consider an embedded control system shown in Figure 1 in which a microcontroller is used to control the velocity profile of a motor to follow a piecewise constant trajectory specified as a sequence of velocity commands and their durations: \( \text{VelCmd}[\text{SeqNo}], \text{Duration}[\text{SeqNo}] \). Here for each sequence number SeqNo, a velocity command VelCmd is specified for a duration Duration.

![Figure 1. A simple embedded control system](image)

An operator can also press a Start (resp., Stop) button (the corresponding event is represented by a binary signal Start (resp., Stop) to start (resp., stop) the motor movement. The motor is used to operate a machine tool which requires the presence of certain supply (of coolant etc.) for a normal operation. A sensor (Supply Detector) is used to detect the presence of the supply, and the output of the sensor is a binary signal HasSupply. The angular position of the motor’s shaft is measured by an Encoder (that generates a quadrature pair of square wave voltage outputs), which together with a Decoder serves as an Analog to Digital (A/D) converter whose output is the quantized position of the motor’s shaft (QuantizedPos). The motor is driven by a Pulse Width Modulation (PWM)-controlled power supply whose output signal (Voltage) is proportional to the pulse on time (PWMOnTime). The PWMOnTime is determined by a Propositional-Integral-Differential (PID) control algorithm given in a Servo Library bundled with the microcontroller. It takes the QuantizedPos and VelCmd[SeqNo] as inputs and computes the output PWMOnTime. Also, the motor’s velocity ServoVel is calculated in the control algorithm as a by-product and is accessible for monitoring.
The control software is meant to ensure the following functionality: For each sequence number SeqNo, when the Start and the HasSupply conditions hold, the motor’s velocity should be set to VelCmd[SeqNo] for the duration Duration[SeqNo]. Whenever the Stop condition holds or the HasSupply condition does not hold, the motor motion should be halted. Also, the motor motion should be halted after the execution of the very last sequence number. The control software program is given below. (MS_TIMER is a global timer variable updated every millisecond.)

```c
#include "servo.lib"
int MaxValSeqNo, VelCmd[100], Duration[100];
void LoadParameters () {
  VelCmd[SeqNo] = ...;
  Duration[SeqNo] = ...;
}

void main () {
  int StartIn, StopIn, HasSupplyIn;
  int TravelTimer;

  void ServoInit() {
    ServoEnable();
    ServoClosedloop(1); //initialize
    StopIn = 1;
    StartIn = 0;
  }

  void ServoDisable() {
    HasSupplyIn = 0;
  }

  if (MS_button pressed or no Supply, stop
  else HasSupplyIn = 0;
}

if (PortIn & 128 == 128) //if has Supply
  TravelTimer = MS_TIMER + TravelTime[0];

if (PortIn & 2 == 2) //if stop button is pressed
  StopIn = 1;

while (1) {
  WpPortI(PADDR, &PADDRShadow, 0x80);
  ServoInit();
  ServoClosedloop(1); //initialize
  StopIn = 1;
  StartIn = 0;
}

while (TravelTimer < MS_TIMER + TravelTime[0]) {
  if (MS_TIMER > TravelTimer) { //if current SeqNo
    TravelTimer = MS_TIMER + Dura-
    StopIn = 1;
  }
}

if (MS_TIMER > TravelTimer) { //if current SeqNo
  TravelTimer = MS_TIMER + Dura-
}

if (SeqNo == MaxSeqNo) StopIn = 1;
else if (StepIn2 == 0) {
  ServoDisable();
  HasSupplyIn = 0;
}
```

Suppose the control software, the A/D converter, and the PWM-controlled power supply can have faults, whereas the other components (such as the Servo Library, the motor, the microcontroller, the supply detector, and the start/stop buttons) do not witness faults. Our goal is to design online monitors for detecting the execution of any unanticipated software fault that manifests during the on-line use. The monitoring system for detecting the software and the system level faults is drawn in Figure 1, where the signals observable to the monitoring system are indicated.

3. Notation and Preliminaries

An embedded control software can be viewed as a dynamical system with continuous (numeric) as well as discrete (symbolic) inputs, states, and outputs. An example of a continuous input is a desired velocity, whereas a discrete input can be a start/stop command. An example of a continuous output is acceleration setting, whereas a discrete output can be an alert signal. Similarly the states of a software are of two types: The ones representing the values of the program control-flow which are finite-valued (represented as nodes in an I/O-EFA graph called locations), and the ones representing the values of the data which can be infinite-valued.

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 (L0), and initial data values (D0). 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. The edges of the graph represent transitions between the locations and are guarded by constraints over the data and the inputs. 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,
- \( \Sigma \) is the set of discrete inputs,
- \( \Delta \) is the set of discrete outputs,
- \( E \) is the set of edges, and each \( e \in E \) is a 7-tuple, \( e = (a_c, t_e, \sigma_e, s_e, G_e, f_e, b_e) \), where
can be expressed as safety properties in the linear-time temporal logic (LTL). LTL formulas are built using atomic propositions, Boolean operators, and temporal operators. In our context, atomic propositions will correspond to predicates over continuous inputs/outputs and data values. Letting $p$ denote an atomic proposition and $\phi$ denote a LTL formula, the following grammar generates the set of all LTL formulas:

$$
\phi \rightarrow \text{True} \mid p \mid \neg \phi \mid \phi \land \phi \mid X\phi \mid U\phi \mid P\phi \mid S\phi,
$$

where $X$ (next-time) and $U$ (until) denote the future-tense operators, and $P$ (previous-time) and $S$ (since) denote the past-tense operators [7]. These operators can be used to derive other temporal operators such as $F$ (in-future), $G$ (globally, i.e., always) etc. A LTL formula is called a past-tense formula if it does not contain any future-tense operators. A property that can be represented as a LTL formula $G\phi$ (always $\phi$), where $\phi$ is a past-tense formula, is called a safety property. Since the atomic propositions correspond to predicates over system variables, the properties will be formulas in “1st-order LTL”. Their satisfiability will be decidable if and only if the satisfiability of the predicates encoded as atomic propositions is decidable.

4. Hierarchical Software Fault Detection

Software can contain unanticipated faults introduced during either design or implementation. We propose a two-tiered hierarchical approach for detecting faults in embedded control software during their runtime operation: The observed behavior is monitored against the appropriate specifications at two different levels, namely, the software level and the controlled-system level.

- At the software level, the behavior of the control software is monitored against its property specifications. A software fault is immediately detected when an observed behavior is rejected by a software level monitor.

- At the system level, the behavior of the controlled-system (that includes the control software together with the underlying hardware that it controls) is monitored against the system level property specifications. 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. This is done by tracking the executed behavior in the system model comprising of the models for the software and those for the nonfaulty hardware components: An acceptance by such a model indicates the presence of a software fault.

The properties can be the same as those used for capturing the functional and safety require-
4.1. Fault Detection at Software Level

For fault detection at the control software level, the input-output behavior of a control software is compared against its properties expressed in the 1st-order LTL. For this purpose, a monitor that accepts the set of all input-output behaviors satisfying a specification formula is constructed as follows:

- Each predicate appearing in the 1st-order LTL specification is treated as an atomic proposition. The 1st-order LTL formula can then be viewed as a proposition LTL (PLTL) with the predicates replaced by the corresponding atomic propositions.

- An acceptor automaton for the resulting PLTL formula is obtained to serve as a monitor. The monitor automaton can be chosen to be deterministic since the properties being monitored are safety properties. Each transition of the monitor automaton is labeled with a predicate over inputs and outputs that is a subformula of the specification formula. Thus, the knowledge of the current input-output values lets us determine the enabled transition in a current monitor state (and accordingly the subsequent monitor state).

- A faulty state is added to the monitor automaton to represent the violation of the specification. A software fault is reported when the monitor reaches the faulty state.

Separate monitors can be constructed to monitor the violations of the separate properties. Thus the approach is modular and hence scalable (this is useful for a real-time implementation). Each monitor is constructed directly from a given property specification, and so a knowledge of the software model is not required.

Example 1 Consider the embedded control software of Section 2. Its discrete inputs are StartIn, StopIn, HasSupplyIn (read via RdPortI(PADR)), continuous input is VelCmd[SeqNo] and continuous outputs are SeqNo,PWMOnTime (written via ServoSetVel(VelCmd[SeqNo]), ServoDisable()). Its data variables are TravelTimer, StopIn, StartIn, PortIn. The control software properties are given by:

1. \( G[RdPortI(PADR)\&2 \Rightarrow \phi_2] \Rightarrow X^{\leq 4}(ServoDisable()\text{ called}) \), and
2. \( G[RdPortI(PADR)\&128 \Rightarrow 128 \&\& P[RdPortI(PADR)\&1 \Rightarrow 1] \&\& \text{SeqNo} < \text{MaxSeqNo} \Rightarrow X^{\leq 8} [\text{ServoSetVel(VelCmd[SeqNo] called)} \text{ || ServoDisable() called}]. \)

The first specification states that anytime the Stop signal is high (2nd pin of port PADR is high), then in at most next four steps the control software must execute the ServoDisable() command that stops the motor. The second specification states that anytime the HasSupply signal is high (5th pin of port PADR is high), the Start signal was previously high (1st pin of port PADR was high), and the sequence number has not exhausted (SeqNo is less than the MaxSeqNo), then in the next eight steps the control software must execute either ServoSetVel(VelCmd[SeqNo]) or ServoDisable(). The corresponding monitors for the two specifications, in which certain states for atomic propositions have been merged, are shown in Figure 3.

\[ \begin{align*}
\phi_1 &= \text{[StartIn(PADR) & 2 == 2]}
\phi_2 &= \text{[ServoDisable() called]}
\end{align*} \]
We can relate the above cases to the state-traces in Figure 3 because the propositions are labeled with the transitions in our monitor model. We have the following:

Case 1 corresponds to the state trace $(C11, C11)$; Case 2 corresponds to either the trace $(C11, C12, C12)$ or the trace $(C11, C12, C11)$, where $p_{i+1} \Rightarrow \phi_2$ in Case 2 is divided into two subcases of $p_{i+1} \Rightarrow (\phi_2 \land \neg \phi_1)$ and $p_{i+1} \Rightarrow (\phi_2 \land \phi_1)$ (and this is why we have two corresponding traces) since we also need to consider the satisfaction of $Spec_{p1}$ by $p_{i+1} \Rightarrow (\phi_1 \Rightarrow X \leq 4 \phi_2)$; Case 3 corresponds to either the trace $(C11, C12, C13, C12)$ or the trace $(C11, C12, C13, C11)$, where again we have two traces like in Case 2; Case 4 corresponds to either the trace $(C11, C12, C13, C14, C12)$ or the trace $(C11, C12, C13, C14, C11)$; and Case 5 corresponds to either the trace $(C11, C12, C13, C14, C15, C12)$ or the trace $(C11, C12, C13, C14, C15, C11)$. In order to violate $Spec_{p1}$ at a proposition $p_i$ in the proposition-trace $\pi$, we must have $(p_i \Rightarrow \phi_1) \land (p_{i+1} \Rightarrow \neg \phi_2) \land (p_{i+2} \Rightarrow \neg \phi_2) \land (p_{i+4} \Rightarrow \neg \phi_2)$, which corresponds to the trace $(C11, C12, C13, C14, C15, C1f)$ in Figure 3. From above we know that the monitor accepts any propositions traces satisfying $Spec_{p1}$ and rejects any traces violating $Spec_{p1}$.

### Sample Trace

<table>
<thead>
<tr>
<th>State in Monitor 1</th>
<th>State in Monitor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(2,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(3,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(4,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(5,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(6,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(7,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(8,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(9,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(10,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(11,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(12,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(13,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(14,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(15,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(16,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(17,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(18,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(19,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(20,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(21,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(22,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(23,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(24,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(25,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(26,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(27,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(28,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
<tr>
<td>$(29,(0,1,0,0,0,0))$</td>
<td>$C1$</td>
</tr>
</tbody>
</table>

**State Notation:**
1: StartIn, 2: StopIn, 3: PortIn, 4: HasSupplyIn, 5: SeqNo, 6: MS_TIMER

---

**Figure 4. Control software trace and monitor states visited**

A sample trace of the control software and the corresponding states visited in the two monitors are shown in Figure 4. (Assume MS_TIMER is 1000ms when the control software reaches the program counter location 9.) Note that the 2nd property is violated along this trace.

### 4.2. Fault Detection at System Level

It is possible that the set of software level specifications is not complete and so every violation of the software may not be detected by the software level monitors. To safeguard against the possible incompleteness of the software level properties, we propose monitoring at the level of the controlled-system against its own set of properties. When a system level monitor rejects an observed behavior of the system, it indicates a system level failure, and an additional isolation step is required to conclude whether a software fault occurred. This is done by tracking the executed behavior in the system model comprising of the models for the software and those for the nonfaulty hardware components: An acceptance by such a model indicates the presence of a software fault.

We have used I/O-EF A to model a software component. It turns out that I/O-EF A can also be used to model the hardware components of the controlled-system. This is because, in an embedded control application, the control changes only at discrete times when the system/environment states are sampled, and as a result the controlled-system has a discrete-time hybrid dynamics which can be modeled using I/O-EF A. (By hybrid dynamics we mean continuous dynamics mixed with switching logic. Input-output hybrid automata \cite{4} can be used to model such hybrid systems, and in the discrete-time setting where inputs, outputs and states are examined when sampling occurs, the state and the output equations associated with each discrete mode of an input-output hybrid automaton can be represented as an update and an assignment function respectively of an I/O-EF A.)

Figure 5 shows the I/O-EF A models of the various hardware components, namely, (i) the components between the supply detector and the microcontroller, (ii) the components between the start button and the microcontroller, (iii) the components between the stop button and the microcontroller, (iv) the discrete-time motor dynamics, (v) the PWM-controlled power supply, and (vi) the analog to digital (A/D) converter.

Note that the A/D converter and the PWM-controlled power supply are subject to hardware faults. When the A/D convert fails, it outputs $-1$, 0, or 1 regardless of the input. Similarly when the power supply fails, it outputs $V_{cc}$ or 0 regardless of the pulse-width of the applied square waveform. The I/O-EF A model of the proportional-integral-differential (PID) controller included in the servo library is shown in Figure 6.
Figure 5. I/O-EFA models of hardware components

Discrete input: HasSupply; Discrete Output: RfPort(PADR) & 128 := 128
Model of components between Sensor Detector & Microcontroller
Start: RfPort(PADR) & 1 := 1
Discrete input: Start; Discrete Output: RfPort(PADR) & 1
Model of components between Start Button & Microcontroller

Continuous Input: QuantizedPos
Continuous output: PWMOnTime, k, ServoVel
Data Variables: ServoPosOld, ServoPos, PosCmd, Err, DErr, IErr
Model of PID Control (provided in servo library)

Figure 6. I/O-EFA model of PID control

Note that in order to isolate a software fault when a controlled-system level failure is detected, the corresponding input-output behavior must be tracked in the controlled-system without the hardware faults. Also note the tracking of an observed input-output behavior can be done independently in each of the components of the controlled-system, i.e., the composition of the individual components to form an overall controlled-system model is not required. This feature makes our approach scalable and suitable for an on-line implementation.

Example 2 Consider the embedded control system shown in Figure 1. Its discrete inputs are Start, Stop, HasSupply, continuous inputs are VelCmd[SeqNo], Duration[SeqNo] and continuous outputs are ServoVel, PWMOnTime, HasSupplyIn. The property specifications of the controlled-system are given by:

1. \( G[\text{HasSupply} \Rightarrow \neg X(\text{HasSupplyIn} \geq 1)] \).
2. \( G[\text{Start} \&\& \text{Stop} \&\& \text{HasSupplyIn} \geq 1] \Rightarrow \left\{ \begin{array}{l}
\frac{1}{N} \sum_{k=0}^{N-1} P_k(\text{PWMOnTime}) > T_{h1} \&\& \\
\frac{1}{N} \sum_{k=0}^{N-1} P_k(\text{ServoVel}) < T_{h2}
\end{array} \right. \)

The first specification states that anytime the HasSupply signal is high, in the next step the value of HasSupplyIn must exceed unity. The second specification states that anytime the Start holds, Stop does not hold, and HasSupplyIn exceeds unity, it must not be the case that the N-step average of PWMOnTime is above a threshold \( T_{h1} \) while the same N-step average of ServoVel is below a threshold \( T_{h2} \). (The temporal operator \( P_k \) denotes “kth step in past.”) The corresponding monitors for the two specifications, in which certain states for atomic propositions have been merged, are shown in Figure 7.

Consider VelCmd[0]=4 with Duration[0]=24000ms and VelCmd[1]=2 with Duration[1]=10000ms. Assume SampleRate=256/s and PWMPeriod=512. Figure 8 shows a trace of a portion of the controlled-system data values while control software program counter is
at location 11 (the data values at other locations along the trace are omitted, while certain data variables are completely excluded). The trace violates the first property of the system since HasSupplyIn exceeds its upper bound 32767 at the 8388th sampling instant, and thus any further increment results in a negative value that causes the system to terminate under the normal operating conditions.

![Diagram of a system with transitions and properties]

**Figure 7. Monitors for specifications of controlled-system**

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>HasSupplyIn</th>
<th>HasSupplyIn &gt; 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stop</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Start</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \phi_1 := \text{HasSupplyIn} \]
\[ \phi_2 := \text{HasSupplyIn} > 1 \]

5. Conclusion

Embedded control systems consist of software components (such as the control or monitoring software) and hardware components (such as the system under control, sensors, actuators, etc.) all of which are subject to faults. We proposed a model-based approach to fault detection in embedded control software. The approach is hierarchical involving monitoring at the control software level as well as the controlled-system level against the software level and the system level properties respectively. The approach is modular and hence scalable, and also safeguards against possible incompleteness of the software level properties. All components including software, hardware, and property monitors can be modeled as input-output extended finite automata in our approach, and this is illustrated through a simple motor control example. The approach guarantees no false-alarms, and further no missed-detections is guaranteed when a complete set of properties are monitored, and there is sufficient input-output observability to ensure the monitorability of the properties. Future work will examine conditions that the level of observability must satisfy so as to ensure the monitorability of the desired properties. Future work will also examine the effect of noise that is present in the measurement of the hardware signals and also the effect of the approximations involved in the hardware models. Finally the formulation of isolation of software level faults when system level failures are detected needs further exploration.

![Trace of a portion of controlled-system data values while control software at location 11]

**Figure 8. Trace of a portion of controlled-system data values while control software at location 11**

**References**


