Making real-time systems fault tolerant: a specification-based approach

Seyed Morteza Babamir\textsuperscript{1} and Saeed Jalili\textsuperscript{2}

\textsuperscript{1}University of Kashan, Kashan, Iran
\textsuperscript{2}Tarbiat Modares University, Tehran, Iran

Received 29 May 2008; revised 05 April 2010; accepted 09 April 2010

To make an event-triggered real-time system safe in application layer, this study presents a specification-based run-time verification (RV) and fault tolerance approach in following steps: i) System is isolated from its environment by modeling interaction between them; ii) Considering safety requirements violation, observation-verification-tolerance rules are systematically obtained; and iii) Rules are weaved into control software (called software instrumentation) by an automatic way. For effectiveness, proposed approach is applied to classic and real-time Railroad Crossing Control System (RCCS).

Keywords: Aspect-oriented, Fault tolerance, Instrumentation, Run-time verification, Safety-critical software

Introduction

Fault avoidance methods (formal methods) and fault removal methods (software testing) are used to make software reliable. However, some faults may be left because of: 1) Undecidability and difficulty of proving correctness of software specification\textsuperscript{1}; 2) Unpredictability of run-time environment\textsuperscript{1}; 3) Impracticality of testing software by all of possible input values\textsuperscript{2,3}; and 4) Difficulty of selection of a proper subset of test data. In addition, software testing method just can reduce failure rate up to $10^{-4}$ failures/h, while for safety-critical software\textsuperscript{4,5}, one expects $10^{-9}$ failures/h. Therefore, a practical solution to handle unknown and unpredictable software faults is that software is made fault tolerant. Fault tolerance\textsuperscript{6} is survival attribute of computing systems or software in their ability to deliver continuous service to their users in presence of faults. Fault tolerance techniques are aimed at preventing systems catastrophe, such as aerospace\textsuperscript{7}, nuclear power\textsuperscript{8} and railway\textsuperscript{9} systems. This study presents a specification-based run-time verification (RV) and fault tolerance approach.

Proposed Approach

In a systematic specification-based RV (run-time verification) and fault-tolerance approach, modified control software detects and recovers from faults that are not handled in hardware or operating system layers. Approach is applied in application layer in three steps (Fig. 1), by which a goal-oriented approach (goals are users’ concerns) is followed. A user’s concern is a safety requirement (SR), indicating that nothing bad happens\textsuperscript{10}. In a real-time system, something bad happens when the system responses to environment events incorrectly or late (Relation 1). Therefore, an SR means that Relation 1 should not hold (Relation 2). In a Railroad Crossing Control System (RCCS), for instance, when train arrives at crossing, gate should not be open (Relation 3). In order to elaborate Relation 2, happenings (events and system responses) are defined as Relations 8 and 9 in step 1.

(1) Happens (event) $\Rightarrow$ Responses (inapt-action)

(2) Happens (event) $\Rightarrow$ ~ Responses (inapt-action)

(3) Happens (train-arrive) $\Rightarrow$ ~be open (gate)

In first step, SRs violation (Fig. 1) is obtained as relations including environment events and system responses to events. These events and responses are obtained from the model, in which system has been isolated from its environment. Having determined SRs violation, an OVT rule is determined for each SR violation in second step. These rules are used to verify control software against SRs violation and to make software tolerant. For demonstrating rules, active rule system is used as Event-Condition-Action (ECA) rules\textsuperscript{11}.
In third step, highly reliable automatic software instrumentation is done with OVT rules using aspect-oriented approach\(^12\).

**Step 1: Isolating System Environment and Introducing SRs**

To elaborate Relation 1, environment event, stated on left hand side of relation, and system response, stated on right hand side of relation, are formally defined. For this purpose\(^13\), system is isolated from its environment, specified using state variables, and environment events and system responses are defined using variables. Having isolated system from its environment (Fig. 2), system environment is modeled by some monitored state \((MS)\) variables, referred to as Relation 4 and system behavior by some controlled state \((CS)\) variables, referred to as Relation 5.

(4) MS\(=\{m_1, \ldots, m_i, \ldots, m_h\}\) where \(m_i\) is an MS variable

(5) CS\(=\{c_1, \ldots, c_j, \ldots, c_p\}\) where \(c_j\) is a CS variable

An MS or CS may have different states/values at different time instants (Relations 6 & 7).

(6) \(m_i=\{m_{i1}, \ldots, m_{ik}, \ldots, m_{in}\}\) where \(m_{ik}\) indicates the \(k^{th}\) state/value of \(m_i\).

(7) \(c_j = \{c_{j1}, \ldots, c_{jp}, \ldots, c_{jq}\}\) where \(c_{jp}\) indicates the \(p^{th}\) state/value of \(c_j\).

To show time instant when \(m_i\) or \(c_j\) takes some value/state, temporal function “@” on MS and CS variables is defined as \(\tau= @(k, z)\), where \(z= m_i\) or \(z= c_j\). In fact, this function gives back time instant when variable \(z\) takes state or value \(k\). Now, event is defined. If some \(m_i\) takes two distinct states/values at two consecutive time instants, one has an environment event (Relation 8). Relation 8 states that an event has happened at \(\tau_i\) if value of \(m_i\) changes at \(\tau_j\) where \(\tau_j > \tau_i\).

(8) \((m_i, \tau_i) \equiv [\tau_i= @(k, m_i)\) and \(\tau_j= @(k, m_j)]\) where \(k_i \neq k_j\) and \(\tau_j > \tau_i\).

When value/state of \(m_i\) changes, system should set value/state of some \(c_j\) to a correct value/state at the right time in order to control its environment in time. Such setting is called a system apt-action (an appropriate response) and formally defines it as Relation 9. In Relation 9, apt-action \((c_j, \tau)\) is taken by the system at \(\tau\) if state/value of \(c_j\) is set to \(k\) at \(\tau\).

(9) apt-action \((c_j, \tau) \equiv [\tau = @(k, c_j)]\), where \(k\) is a correct value and \(\tau\) is a proper time instant

A system action (Fig. 2), which is taken by setting some \(c_j\) to value \(k\), is a response to an environment event, which is identified by change of some \(m_i\). In other words, when some event happens, control software receives event from monitor component and deal with computing value \(k\). Afterwards, it sends value to controller component to do an apt-action.

**Safety Requirement (SR)**

SRs are constraints imposed through system environment on system functions. These constraints are shown in the form of “\(\square\) [if event then apt-action]” and defined formally as Relation (10). Notation “\(\square\)” is a temporal logic notation and means that relation always should hold\(^14\). Such constraint indicates a correct and timely \(c_j\) value/state when some \(m_i\) takes a specific value/state.

(10) \(\square\) [if event \((m_i, \tau_i) \Rightarrow \) apt-action \((c_j, \tau_j)\)]\), where event \((m_i, \tau_i)=[\tau_i= @(k, m_i)\) and \(\tau_j= @(k, m_j)]\) and \(\tau_j \neq k_j\), apt-action \((c_j, \tau_j)=[\tau_j = @(k, c_j)]\) and \(\tau_j < \tau_j < \tau_j\).

Violation of an SR is shown by negation of Relation 10 (Relation 11). It means that when an event occurs,
system takes an inapt-action because control software: 1) Computes values/state of $c_i$ incorrectly; 2) Does not compute a necessary value/state for some $c_j$; or 3) Computes value/state of $c_i$ late. Time instant $\tau_i$ in Relation 11 shows deadline of apt-action.

\[ (11) \Diamond \left[ \text{if } \text{event}(m_i, \tau_i) \Rightarrow \text{inapt-action}(c_j, \tau_j) \right] \text{ where } \text{“inapt-action” is the negation of “apt-action”} \]

Notation “$\Diamond$” is a temporal logic notation and means that relation eventually holds\(^{14}\). In RCCS, for instance, when event “approach” occurs, action “gate closing” is an apt-action and its negation, “being gate open” (means not taking a correct action) is an inapt-action.

**Step 2: Determining OVT Rules**

An OVT rule is considered for monitoring each SR violation. In order to specify SR violations, fault tree\(^{15,16}\) is considered. A fault tree is a chain of statuses that causes system fails in response to its environment. In fact, tree is a deductive top-down technique, in which its root is a system failure and its leaves are causes of failure. Parents in tree are conjunctions or disjunctions of their children. Each fault is categorized in four categories (computational, algorithmic, hardware and mechanical), which are put at the first level of tree. Then, each level is extended by related faults. Finally, leaves of tree show account of SRs violation.

When an environment event occurs or state/value of some $m_i$ changes, in response to event, system acts by changing state/value of some $c_j$. By monitoring changes of $m_i$ and $c_j$, violation of SRs can be verified. Having any SR violation, system should be redirected to a correct state to tolerate violation. OVT rules undertake to monitor states/values of $m_i$ and $c_j$ to verify SRs violation. Having generated a violation, OVT rule undertakes to make system fault tolerant through state redirection.

**Constructing OVT Rules**

Given a fault tree, an OVT rule is produced for each leaf of fault tree. Active rules represent OVT rules. Active rules mainly are used for monitoring an active database management system (ADBMS)\(^{11}\), which is an event-based system, in which some system actions (data modification) raise some events. The events are monitored and system reacts to them by triggering corresponding active rules. In addition, active rules have been used for control processing\(^7\), which monitors work progress with some sensors and control a process making with some actuators.

![Fig 3—ECA rule](image)

An active rule appears in the form of event-condition-action (ECA) (Fig. 3). Event part is used for observing events, condition part for verifying violation, and action part for making system fault tolerant. So, RV is achieved by combination of event-condition. Fault tolerance is based on forward or backward recovery concept and appears as exception handling and voting schemes. One state forward (backward) recovery is used to tolerate delay or hurry faults. Such faults denote that system has mistimed a reaction. One state recovery attempts to forward (backward) system from an unsafe state to an immediate successor (predecessor) state, from which system can continue operation.

For a general fault tree, types of fault tolerance rules are represented in Rule set 1, where each rule denotes a leaf. In Rule set 1, Rule 1 makes hardware fault tolerant by replacing faulty hardware with a healthy backup one and Rule 2 makes mechanical component fault tolerant with human intervention. Rules 3 and 4 denote software faults i.e. algorithmic and computational ones. In case of software fault, it is first assumed that system application suffers algorithmic fault and hence another version of application should be selected by voting scheme (Rule 3 in Rule Set 1) such as N-version programming (NVP). In NVP scheme, each version uses a distinct algorithm. Having executed all versions, final decision is taken by the vote of versions in terms of their results. NVP scheme is based on the assumption that software built differently should fail differently and thus if one of the redundant versions fails, at least one of the others should provide an acceptable output.

However, if conditions are still adverse, control software suffers computational fault and hence an exception handler should be invoked. Computational faults usually are on account of: 1) Conversion from one type of variable into another type (assigning to an integer variable a real value or assigning to a logical variable an integer value); and 2) Operations such as divide by zero and computation overflow. Therefore, to make software fault tolerant against computation faults, one can place variable assignments and computation operations in
exceptionable blocks to pass application execution control to an exception handler if some fault occurs. Exception handler recovers system to an immediate successor or predecessor state (rule type 5 in Rule Set 1) by making a proper reaction. Proper actions appear in the action part of rules in the form of mandatory and prohibitive ones. When system is late in reaction to an event, a mandatory action must be taken and when system is early in the reaction, a prohibitive action must be done. Both faults denote that system has mistimed reaction.

By mandatory and prohibitive actions, system can be recovered from an unsafe state to a safe state in order to the system can continue its tasks. When control software faces a fault at run-time, system will make a transition into an unsafe state and an unexpected and erroneous result will be created. If result propagates over the system, eventually the system user observes a failure. However, once an unsafe state has been identified, by redirecting the system to a safe state, recovery can be initiated to prevent from the propagation of erroneous result.

**Step 3: Instrumenting Software**

To instrument control software by OVT rules, determine software locations to insert verification and fault tolerance codes of rules. Codes monitor events and verify conditions to prevent control software from abnormal termination state when control software violates any safety requirements. To instrument control software efficiently, constraints are: 1) How to identify correspondence between low-level software activities and high-level events; and 2) How to find out and localize scattered locations over control software? In cases 1 and 2, modularization of scattered code of an SR over software facilitates automatic instrumentation of control software (i.e. weaving verification code into application). Based on modularization, one is capable of weaving RV and fault tolerance, OVT rules into control software automatically. Generally, manual software instrumentation is subject to time-consuming and over sighting code because one should localize scattering locations in software and insert verification code manually.

In case 1, high-level environment events should be mapped onto low-level software execution activities. To do such mapping, in the first step, SRs are separated from system functions. Then, an OVT rule is determined for each SR violation. Now, run-time verification part of OVT rules is mapped, i.e. event and condition parts of rule into corresponding low-level software activities. A method in control software is considered for handling each event of rule. Therefore, event part of each active rule will have a corresponding method in control software. To observe environment events, an extra code is not needed. However, in order to verify SRs violations (condition part of each rule), method should be instrumented, i.e., some corresponding “if” commands are inserted in the method. Fault tolerance (action part of each rule) is accomplished through invocation of a fault tolerance method (exception handler). Invocation commands are put in the then part of each “if” command.

In case 2, code of each OVT rule is weaved to corresponding locations scattered over control software. For this purpose, localize those locations. Manual localization and code weaving are hard, error-prone and it may cause some locations are ignored. Therefore, for efficiency, apply automatic localization and weaving. For example, in Fig. 4, control software contains m modules and each module contains n methods (functions) on average. Consider a constraint whose code appears in half of the software methods, one should compose a method and insert its invocation in $\frac{m \times n}{2}$ different locations of control software. These application-wide requirements (concerns) that span multiple modules are called crosscutting concerns. An increase in concerns makes manual instrumentation a very difficult task. One solution for managing crosscutting concerns efficiently and automatically is using aspect-oriented approach, which is the program development pattern and modularizes a crosscutting concern by introducing an aspect, consisting of join points, pointcuts, and advice codes.

Fig. 5 shows automatic instrumentation of software shown in Fig. 4 by applying aspect-oriented approach. A join point is an identifiable point in control software
such as call to a method or an assignment to a data member of an object. Code revolving around join points is the place where crosscutting actions are woven in. A pointcut is a program construct that selects join points and collects context at those points. In fact, it describes a localizing method using some join points i.e., it shows how aspect should crosscut control software modules. For example, a pointcut can select a join point that is a call to a method and it could also capture method’s context.

Advice is code to be woven and executed at a join point that has been selected by a pointcut. Advice can be executed before, after or instead of surrounded code of join point. Therefore, a concern (an OVT rule) is implemented in an aspect instead of all core modules. An aspect weaver, which is a compiler-like entity, observes aspect join points and pointcuts and composes instrumented application by combining core software and advice code.

Exploiting Some Similarities

There are some similarities between an ECA rule and an aspect enabling software instrumentation automatically. The event and condition parts of an ECA rule corresponds to pointcut and joint point parts of an aspect. Action part of an ECA rule corresponds to advice part of an aspect. In effect, using these similarities, one can automatically instrument control software with a set of OVT rules. Also, each pointcut of an aspect has modifiers before, after and around, which correspond to before, after and instead modifiers of an ECA rule. Therefore, aspect modifiers can be applied to implement ECA modifiers directly.

Case Study: Applying Proposed Approach

For effectiveness, proposed approach is applied to RCCS system. Leveson and Heitmeyer designed a benchmark for specification and verification of real-time approaches to control traffic at rail and road junction. System comprises an input sensor for monitoring approach of a train to crossing, an output sensor for monitoring exit of a train from crossing, a timer for monitoring passing of time, a gate for closing and opening road and a control unit to control gate by steering software. When input sensor (or output one) senses a train, it notifies control unit to move down (or move up) the gate. Therefore, system contains three monitor components and one controlling component to open/close a gate at a railroad crossing to satisfy SRs.

Modeling RCCS System

According to Fig. 2, RCCS is modeled in Fig. 6, where system has been isolated from its environment. Considering problem specification, system environment consists of train and road. However, since just train is monitored, system environment is specified by an MS variable, $M_{\text{train}}$, which may have four states (distant, approached, on-crossing and passed). In addition, internal MS variable, $M_{\text{timer}}$, whose value is set by system timer, is considered to monitor passing of time. Number $\text{max-count}$ is time span when train makes transition from state approached to state passed.
Table 1—RCCS specifications and safety requirements (SRs)

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Description</th>
<th>Safety requirements (SRs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach to crossing</td>
<td>Distance between input sensor and crossing point is given (after train detected by input sensor, at least it takes t time units until train arrives at crossing point).</td>
<td>SP1: t time units from train detection by input sensor, gate must not be open or about to move down.</td>
</tr>
<tr>
<td>Exit to detect</td>
<td>Interval time between two successive trains is given (there is at least t/3 time units between a train departure from crossing point and next train arrival at input sensor).</td>
<td>LP1: By at least t/3 time units after train leaves crossing point, gate must open the road.</td>
</tr>
<tr>
<td>Pass</td>
<td>Maximum speed of train is given (at least it will take a time unit until train passes crossing point).</td>
<td>-</td>
</tr>
<tr>
<td>Down to close</td>
<td>Moving down speed of gate is given (it takes from t/15 to t/6 time units until gate moves down completely i.e., road closes).</td>
<td>-</td>
</tr>
<tr>
<td>Up to open</td>
<td>Moving up speed of gate is given (it takes from t/15 to t/3 time units until gate moves up completely i.e., road opens).</td>
<td>SP2: before passing t/3 time units from train detection by output sensor, gate must not open the road.</td>
</tr>
</tbody>
</table>

\[ m_{1} = M_{train} = \{ m_{i1} = \text{distant}, m_{i2} = \text{approached}, m_{i3} = \text{on-crossing}, m_{i4} = \text{passed}\} \]

\[ m_{2} = M_{timer} = \{0, 1, \ldots \text{max-count}\} \]

RCCS: Determining Events and Actions

RCCS events are Interrupt, approach, cross and pass. Interrupt indicates passing of one time unit and rises at every time unit. Approach and cross indicate train approach and entry to crossing respectively. Pass indicates train exit from crossing. Approach and exit are sensed by input and output sensors. Cross rises at time instant t, where t is computed as \( x = vt \). Parameter x indicates distance between input sensor and crossing and v indicates train speed. Control software sets \( M_{timer} \) variable to zero when it receives approach or pass and increments \( M_{timer} \) one unit when it receives an Interrupt. So, \( M_{timer} \) shows elapsed time since an event has occurred. Also, control software sets \( M_{train} \) variable to approached and passed states when it receives approach and pass events from input and output sensors respectively. Default value of \( M_{train} \) variable is distant. According to Relation 8, above relations are shown as Relations 12 to 15. There are two other internal events raised by self-test circuits: i) Control unit hardware fault (AlertByControl); and ii) Gate fault (AlertByGate).

\[ \text{(12) interrupt (} M_{timer}, \tau_{i} \text{)} \equiv [ \tau_{i} = @(k_{i}, M_{timer}) \text{ and } \tau_{2} = @(k_{2}, M_{timer}) ] \text{ where } k_{2} = k_{i} + 1 \]

\[ \text{(13) approach (} M_{train}, \tau_{i} \text{)} \equiv [ \tau_{i} = @(k_{i}, M_{train}) \text{ and } \tau_{2} = @(k_{2}, M_{train}) ] \text{ where } k_{2} = \text{on-crossing, } k_{2} = \text{passed} \]

System-controlling component is just the system gate. It is shown by a CS variable (called \( C_{gate} \)), whose state is set by system software as MoveDown and MoveUp, and closes/opens the road. Thus, \( c_{i} = C_{gate} = \{ c_{i1} = \text{MoveDown, } c_{i2} = \text{MoveUp}\} \). Normal events monitored by system timer and sensors, are sent to control software to compute appropriate reaction for controller component gate.

RCCS: Determining Safety Requirements (SRs)

Considering specifications of RCCS system, SRs are determined (Table 1). Requirements (SP1, SP2 and LP1) are extracted from specifications ApproachToCross, UpToOpen and ExitToDetect respectively indicating constraints on system functions and must be satisfied by RCCS control software. SP1 and SP2 are safety requirements and LP1 is a liveness requirement. While an SR denotes a bad event should not happen, a liveness one denotes a good event must eventually occur.

Having determined fault tree of RCCS system (Fig. 7), leaves of tree (faults) are considered as account of violation of SRs, SP1 and SP2. Considering Relation
11. Leaf 1 corresponds to violation Not taking correct action MoveDown and leaf 2 corresponds to violation Taking correct action MoveDown late, i.e., after t time units. According to Relation 11, these two inapt-actions are formally stated as Relations 16 and 17. Similarly, violations of SP2 may be stated.

(16) Description: train approaches to crossing point, but gate is still up.

tolerance: raise MoveDown  //mandatory action

◊ [if approach(M_{train}, \tau_1) \Rightarrow \neg MoveDown(C_{gate}, \tau_3)]

where

approach (M_{train}, \tau_1) \equiv [\tau_1= @(distant,M_{train}) and \tau_2= @(approached,M_{train})] and

MoveDown(C_{gate}, \tau_3) \equiv [\tau_3= @(MoveDown, C_{gate})],

\tau_1<\tau_2<\tau_3

(17) Description: train has already entered crossing, but gate has not closed road yet (delay fault indicated by t \prec \tau_3 - \tau_1)

◊ [if approach(M_{train}, \tau_1) \Rightarrow MoveDown(C_{gate}, \tau_3)]

where

approach (M_{train}, \tau_1) \equiv [\tau_1= @(distant,M_{train}) and \tau_2= @(approached,M_{train})] and

MoveDown(C_{gate}, \tau_3) \equiv [\tau_3= @(MoveDown, C_{gate})],

\tau_1<\tau_2<\tau_3

RCCS: Determining Active Rules

Having specified SRs violation of RCCS system, an OVT rule is defined for each violation in Rule Set 2. Considering Relations 15 and 16, it is thought that: 1) Left hand side of relations as event part of the rules; 2) Right hand side of relations as condition part of the rules; and (3) Fault tolerance of violation as the action part of the rules.

For violations committed by hardware and mechanical components, Rules 1 and 2 are stated in Rule Set 2. Event Alert in Rules 1 and 2 of Rule Set 2 has highest priority over others. For violations committed by control software (i.e. algorithmic and computational faults), Rules 3-6 are stated to tolerate them by software methods. However, Rules 7-9 will be fired if any fault still exists. When a software fault is made, it is first assumed that control software suffers an algorithmic fault and tolerance is achieved by selecting another version of module of control software. Selecting mechanism is performed by some voting method such as N-Version programming (Rules 3-5) where another version is selected and a voted event is raised. If problem still exists, fault is a computational one and Rule 6 is fired. Finally, exception handler forces system recovery by raising a mandatory action (MoveDown in Rule 7 and speed up in Rule 9) or by a prohibitive action (cancel-action in Rule 8). Recovering is accomplished by raising a system action that causes system to pass to a safe state.

RCCS: Instrumenting Control Software

In order to instrument control software of RCCS with OVT rules (Rule Set 2), one aspect including a pointcut (and its join point) is defined for event (i.e., ON part of OVT rule) and an advice code for fault should be tolerated (i.e., Do part of OVT rule). Fig. 8 shows aspect definition of OVT rule 3 in Rule Set 2. Advice code is call voting() weaved by aspect weaver into approach method. By this method, RCCS control software processes event approach. While RCCS sensors monitor event approach or pass, instrumented control software takes control and verifies corresponding SR to make system fault tolerant if system has violated it. Otherwise, control software continues its normal task.

Related Works and Proposed Approach

Basically, techniques for achieving fault tolerance depend upon effective deployment and utilization of redundancy and most related work address fault tolerance problem at code level or at design level and propose classical schemes such as software redundancy and voting mechanism for making software code fault
tolerant. Little work\textsuperscript{21} addresses fault tolerance in real
time systems at specification level. At design level,
design diversity is mainly used for protection against
certainty\textsuperscript{22,23}. Design diversity techniques are applied
to software design to build program versions that fail
independently and with low probability of coincidental
failures. Due to complexity of software, use of design
diversity for software fault tolerance is an art rather
than a science\textsuperscript{22}. Another problem is that additional
redundancy may increase size and complexity and thus
adversely affect software reliability. Using design
patterns is another design level fault tolerance technique\textsuperscript{24}
aiming to present a fault tolerance domain (i.e. hardware,
software and environment) in order to help development
of well-structured fault tolerant systems.

Features of Proposed Approach

Proposed approach is a specification-based run-time
verification and fault tolerance approach to event-
triggered real-time applications. A run-time requirement
verification and fault tolerance approach is useful when
fault tolerance schemes is not capable of detecting faults,
particularly run-time environment ones. Shimeall &
Leveson\textsuperscript{25} stated that simple range checks by run-time
assertions could detect 23 faults that were found in no
other fault tolerance schemes. Also, run-time verification
has emerged as a new subject in software analysis and
has attracted interest of researchers\textsuperscript{26}. This study applied
a RV method to Insulin Pump safety critical system\textsuperscript{27}
and real-time systems\textsuperscript{28}.

Present approach put forward a model-based
approach, wherein high level system specification is
represented with domain vocabulary and decomposed
into system functions and safety constraints. Specification and model based features provided the
highest abstraction of users’ requirements. Decomposition feature helped to deal with weaving fault
verification and tolerant rules into control software automatically and thoroughly. New ideas in run-time
monitoring and verification approach have addressed
model-based approach\textsuperscript{29,30}. Proposed approach also
exploited idea of depending on system specification to
make a system fault tolerant. However, specifications
used active rules. This made current approach susceptible
of using aspects to weave rules into control software
clearly and automatically. While there may not be enough
reliability for completeness of manual weaving, current
approach exploiting aspects presented thorough software
instrumentation.

Conclusions

Proposed approach built a bridge over gap between
high-level environment events and low-level control
software activities. Accordingly, current approach
enabled to make control software fault tolerant against
safety requirements violation in run time environment.
In effect, current approach is a dynamic approach, which
covers fault tolerance more thorough because it monitors
control software in its run-time environment and makes
control software fault tolerant against those faults not
detected in verification and validation phases of system
specification.

References

1 Xu J, On inspection and verification of software with timing
2 Gahl D J, DijKstra E J & Hoare C A R, Notes on Structured
3 Parnas D L, Schouwen A J V & Kwan S P, Evaluation of safety-
4 Butler R W & Finelli G B, The infeasibility of quantifying the
reliability of life-critical real-time software, IEEE Trans Soft-
5 Littlewood B & Stringini L, Validation of ultrahigh dependabil-
69-80.
6 Avizienis A, Fault-tolerance: The survival attribute of digital
7 Guillaume J. J. D, Fault-tolerant Flight Control and Guidance
Systems: Practical Methods for Small Unmanned Aerial Vehicles
(Springer, Zurich, Switzerland) 2009, 3-6.
J, PODS, A project on diverse software, IEEE Trans Software
9 Chakraborty A, Fault tolerant fail safe system for railway signal-
ing, in Proc World Congr on Engg & Comput Sci (IAENG Pub-
llication, Sanfrancisco, USA) 2009, 1177-1183.
10 Handbook of Automated Reasoning, edited by J A Robinson &
A Voronko (Elsevier and MIT Press, USA) 2001, 1637-1640.
11 Dayal U, Ten years of activity in active database systems: what
have we accomplished? in Proc 1st Int Workshop on Active and
Real-Time Database Systems (ARTDB-95), Workshops in Com-
puting (Springer-Verlag, Sweden) 1995, 3-22.
12 Laddad R, Aspectj in Action: Enterprise AOP with Spring Ap-
aplications, 2nd edn (Manning Publication, USA) 2009, 3-8.
13 Parnas D L & Madey J, Functional documents for computer sys-
14 Kroger F & Merz S, Temporal Logic and State Systems (Springer-
Verlag, USA) 2008, 19-63.
15 Leveson N G & Stolzy J L, Safety analysis using petri nets, IEEE
16 Veseley W E et al, Fault Tree Handbook, NUREG-0492 (US
17 Bae J, Bae H, Kang S & Kim Y, Automatic control of workflow
processes using ECA rules, IEEE Trans Knowledge & Data


