

Specifying Fault Tolerance in Mission Critical Systems

Tolety Siva Perraju
Dept. of Computer Science,
Wayne State University
Detroit MI 48202
tolety@cs.wayne.edu

Satyendra Pal Rana*
IBM Corporation
Austin, TX78758
srana@vnet.ibm.com

Susanta P Sarkar
VETRONICS Lab,
US Army TACOM
Warren MI 48397
sarkars@cc.tacom.army.mil

Abstract

Dependability is a central concern in the design of mission critical systems. A major design constraint is that the system cannot be brought down for repair during mission times. A number of alternate designs are possible for a given specification. Alternate designs necessitate evaluation. This requires capturing the system specifications and design alternates in the same formalism. In this paper, we propose an extended I/O automata to specify fault tolerant requirements of dependable mission critical systems. The properties of the behaviors of the extended automaton can capture temporal properties like deadlines. This framework is then used to specify the fire control system of a combat vehicle and demonstrate the usefulness of the proposed framework for capturing fault tolerance aspects in mission critical systems.

1 Introduction

The development of mission critical distributed systems with fault-tolerance requirements is difficult because of the trade-off between reliability and responsiveness. This problem is further exacerbated in embedded systems where minor errors have catastrophic consequences [8]. Mission critical systems are characterized by *the inability of repair during mission time, physical constraints of space and volume, and high probability of transient failures*. A typical example of is a Combat Vehicle Electronic System (CVES) [9]. Constant efforts are directed towards developing more accurate, responsive and dependable mission critical systems. A trade-off between dependability and responsiveness is made in the design of a mission critical system, leading to alternate designs. Selection among alternates, necessitates evaluation. Hence, it is necessary to capture the system requirements specifications

and the designs in a single formal framework. In this paper, we propose to extend the basic I/O automaton [6] model to capture the fault tolerant properties of mission critical systems. Section 2, gives an overview of I/O automaton model. Section 3 motivates the need for extensions and presents an *extended I/O automaton* model. Section 4 shows how the concepts of fault tolerance can be captured using the proposed extended I/O automata. In Section 5, we specify the fire control system of a generic combat vehicle using our formal framework to demonstrate how the proposed framework can capture fault tolerance aspects of mission critical systems. Section 6 concludes with directions for future work.

2 I/O Automata

I/O automata provide an appropriate model for discrete-event systems consisting of concurrently-operating components [6]. These systems continuously receive input from and react to their environment. Each system component is modeled as an automaton, a mathematical object with states and named transitions between them [6]. The actions of an automaton are classified as either *input*, *output* or *internal*. In an I/O automaton the internal and output actions are generated autonomously, and the output actions are transmitted instantaneously to its environment. The input actions are generated by the environment and transmitted instantaneously to the automaton. The important characteristics of I/O automata are *non blocking of input, nondeterminism, composition and externally visible events or behaviors*. An automaton is suitable for a particular purpose, if the behaviors of the automaton have the appropriate characteristics. Thus correctness conditions of an automaton are formulated in terms of properties of the automaton's behavior. One way to specify properties of an automaton's behavior is in terms of another automaton.

*Supported by US Army Contract DAA07-94-C-R010

That is, one can define a specification automaton B, and say that an automaton A correctly implements B, if every finite behavior of A, is also a finite behavior of B [6]. This model permits description of the same system at different levels of abstraction. This powerful abstraction facility in I/O automata led to its use in investigation of various facets of distributed systems [6]. Extensions to the basic I/O automaton model are proposed to capture semantics of particular domains [2, 3, 7, 9]. In the following section, we propose yet another extension of I/O automaton model for specifying the fault tolerance properties.

3 Extended I/O Automata

The specification of fault tolerance in dependable mission critical systems is a major input to the design process. A formal approach to specifying fault tolerance, requires the underlying formalism to represent fault tolerance in an unambiguous and implementation independent manner.

An important property to be captured in these specifications, is the kinds of faults to be tolerated by the system. For recoverable systems, with each type of fault, recovery requirements must also be specified. To capture the *normal*, *fault* and *recovery* aspects of the specification, we classify the automata signature into three kinds of actions. These are *Normal actions*, *Fault actions* and *Recovery actions*.

Normal actions are the set of input and output actions that occur in an execution of the system in a normal mode with no faults. A fault can occur due to an external injection or manifest itself due to some internal actions of the system. A fault action can be either input action or output action depending on whether the fault is injected or generated within the system. When recovery is triggered by an external intervention, it is an input action and when it becomes visible due to some actions within the system, it is an output action. Thus, each of the above three action sets may have both input and output actions.

Input to I/O automaton cannot be blocked and input actions are always enabled. In fault tolerant systems, all inputs are not active all the time though they may be enabled. For example, a recovery input though enabled has no effect in the normal mode of operation of the system. To capture this subtlety, we describe actions to be either *active* or *inactive* in a specification. An active action can further be disabled or enabled. Only enabled active actions can be executed. An inactive action being enabled has no significance. It cannot be executed even if enabled permitting us

to ignore processing of extraneous inputs in a given mode (for example certain inputs can be ignored after occurrence of a fault and before recovery). Thus at any given instant of execution of an automaton, there is a set of active actions and another set of inactive actions. These sets are dynamic and actions move from one set to another during the course of an execution. For example a recovery action may be in the inactive set initially, but can become active upon the occurrence of a fault.

Another facet of fault specification is the extent of visible damage done by the fault to the system's normal functioning. This extent of visible damage is specified through two lists called the **activation** and **deactivation** lists. Each action has an associated set of actions which it turns off (makes inactive) upon being enabled/executed. This set is called the **deactivation** list of the action. Similarly the set of actions turned on (made active) by an action are maintained in its **activation** list. Actions in the **activation** (**deactivation**) list of an *action* are activated (**deactivated**) either immediately after the action is enabled or when the action is executed. Hence a flag which indicates when the particular action is to be turned on (**off**) is associated with each member of the **activation** (**deactivation**) lists. The duration of a fault is another facet that affects the recovery and performance aspects of the systems. Hence it is necessary to capture the duration of presence of faults. This is achieved through specifying *lower* and *upper* bounds for the local actions in the automaton. The *lower* bound of a *fault* action specifies the time for which a fault is latent, while the *upper* bound specifies the maximum time for which the fault is present. The *lower* and *upper* bounds for *normal* actions define the deadlines. For *recovery* actions they specify the time taken for recovery from the fault. These extensions allow us to capture a spectrum of faults and fault tolerance requirements for specifying dependable mission critical systems.

Automaton specifying a system at the highest level of abstraction can be realised through the composition of several subsystem automaton. A suitable *composition operator* is defined for this purpose.

3.1 Formal Definition

The extended I/O automaton A is a tuple consisting of

- an **extended action signature** $sig(A)$
- a set of $states(A)$ of states

- a non empty set $start(A) \subseteq states(A)$ of start states
- a transition relation $steps(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$, with the property that for every state s and input action π , there is a transition (s, π, s) in $steps(A)$.
- a partition $part(A)$ of locally controlled actions into at most countably many equivalence classes. Every action in a class C has lower and upper time bounds $lower(C)$ and $upper(C)$, where $0 \leq lower(C) < \infty$ and $0 < upper(C) \leq \infty$

The **extended action signature** is the set of actions of the automaton A . These actions are classified as *normal*, *fault* or *recovery* actions. Actions are also classified as *input*, *output* and *internal* actions. The set of *input* and *output* actions are called the *external* actions. The set of *internal* and *output* actions are called *locally controlled (local)* actions.

An *action* in the extended signature $sig(A)$ is denoted by a tuple consisting of an *action label*, *default action status value*, *activation list* and *deactivation list*. The action label denotes the name of the *action*. The default action status value denotes the default status (on/off). An on indicates an active action and off an inactive action. The activation list is a list of actions which are activated by the enabling or execution of the action. The deactivation list is the list of actions which are deactivated by the enabling / execution of the action. Each action label in the activation (deactivation) list has a flag associated with it. This flag indicates if the action label is activated by the enabling of the containing action or after its execution. When an action is enabled all actions in its activation (deactivation) list with an *enable (enab)* flag are activated (deactivated). Similarly, when the action is executed all actions in its activation (deactivation) list with an *execution (exec)* flag are activated (deactivated).

All *input* actions of the automata are enabled. However an action can be executed only if it is both active and enabled. Alternately, an inactive action is unavailable in the automaton. An action becomes available when it is active. Thus an extended automaton may dynamically change. Dynamically evolving systems can be specified in a simple manner by extended I/O automata.

All actions of the automaton are executed instantaneously. A timed execution of the automaton is an alternating sequence of actions of the form $s_0, (\pi_1, t_1), s_1, \dots$ where $\pi \in acts(A)$. For each j , it must be that $s_j \xrightarrow{\pi_j} s_{j+1}$. The successive times are non decreasing and are required to satisfy the given lower

and upper bounds [7]. The subsequence of external actions in a timed execution of automaton A is a *timed trace (behaviour)* of the automaton. An automaton A is said to implement a specification automaton B , if all finite behaviours of A are also the finite behaviours of B .

3.2 Properties of behaviours

The properties of behaviours characterise the dynamic aspects of the system being specified by the **extended automaton**. These properties can be classified as either *generic* or *domain specific*. Generic properties include *safety conditions* and *liveness properties*. In mission critical systems some temporal properties like *deadlines* and *schedulability* are also generic.

Assurance in mission critical systems must, in principle, consider all possible behaviours of the systems. This is achieved by specifying a *safety property* that will hold in all possible states of the specification automaton. A *safety property* of a behaviour is an assertion that something may never happen [5]. An assertion that something eventually does happen is a *liveness property*. More formally, a *liveness* condition L for an automaton A is a subset of the executions of A , such that any finite execution of A is a prefix of an execution of L [5]. In [1], it is shown that an augmented definition of *liveness* is required to compose automaton while preserving the *liveness* properties. This is based on the notion of *receptiveness* [1]. A system is said to be receptive if it behaves properly independent of the moves of the environment¹. The behaviour of the system is determined by a *strategy*. A *strategy* is a pair of functions (g, f) . Function ' g ' specifies what state will be reached next in response to an input action; function ' f ' determines the next move of the system, which is a local action or null(\perp)² action. Let A be an automaton, α a finite execution of A , (g, f) a strategy defined on A , and I be an external input sequence for A . Then $O_{(g,f)}(\alpha, I)$ is the outcome of strategy (g, f) given I [1]. A pair (A, L) where A is an automaton, L is a subset of the executions of A , is defined to be environment free if there exists a strategy (g, f) such that for any finite execution α of A , and any input sequence I , the outcome $O_{(g,f)}(\alpha, I) \subseteq L$. An environment free pair (A, L) is a live automaton, where L represents the *liveness* properties of A . An

¹In mission critical systems, this implies that the system behaves correctly in spite of faulty inputs

²A null action by the automaton in response to an input, implies that the automaton does not respond to the input. In mission critical systems, if a null(\perp) action is associated with all undefined inputs, it implies ignoring spurious inputs

environment free pair (A,L) implies that the automaton A behaves properly irrespective of the spurious inputs that can be obtained from the environment. Since, in mission control systems it is necessary that the specified service be provided inspite of spurious and faulty inputs, the specification automaton should be environment free. Hence, the specification for a mission critical system should be an environment free pair (A,L).

3.3 Deadlines and Schedulability

The set of tasks in a mission critical system can be classified as either **periodic** or **sporadic**. **Periodic** tasks have two temporal properties viz. **period** and **execution time**. The **period** specifies the maximum time that can elapse between two consecutive executions of the **periodic** task. **Execution time** specifies the time required for the complete execution of the task. The temporal properties of **sporadic** tasks are **deadline** and **execution time**. The **deadline** specifies the time allowed for the completion of the task, once it enters the system.

These temporal properties can be specified in the automaton formalism. In order to do so, each task T_i is represented by a single input action T_{ii} and an output action T_{io} . The input action T_{ii} abstracts the task's invocation, and the output action T_{io} embodies the task's computation. The input action T_{ii} enables the output action T_{io} . Let T_i be a **periodic** task with a period of t_{ip} and execution time of t_{ie} . In the automaton formalism, lower and upper time bounds can be specified for local actions (i.e internal and output actions). For the periodic task T_i , we specify lower and upper time bounds on the output action T_{io} . The bounds are $[t_{ie}, t_{ip} - t_{ie}]$. The lower bound t_{ie} specifies that the action should be enabled for a time equal to atleast it's execution time. The upper bound specifies that if the output action is not scheduled for execution before $t_{ip} - t_{ie}$, then it cannot be successfully completed. This specification captures the semantics of the time constraints associated with **periodic** tasks. Similarly, if T_j is a **sporadic** task, with a deadline of t_{jd} and an execution time of t_{je} , then the lower and upper bounds on the output action T_{jo} associated with the task T_j are $[t_{je}, t_{jd} - t_{je}]$.

Mission critical systems can be characterised by a set of well defined periodic tasks³. It is important to verify, if the set of tasks in the mission critical system

³Sporadic tasks in a mission critical system can be modelled as periodic tasks with a period of $1/t_f$, where t_f is the frequency of occurrence of the sporadic task.

is schedulable. The automaton specification approach lends itself to the process of verifying schedulability.

Let $T = \{T_1, T_2, \dots, T_i, \dots, T_n\}$ be a given set of periodic tasks. Each task T_i is a quadruple $\langle T_{ii}, T_{io}, t_{ip}, t_{ie} \rangle$, where T_{ii} abstracts the task T_i 's invocation, T_{io} abstracts the computation embodied in task T_i , t_{ip} is the cycle time of task T_i and t_{ie} is the execution time of task T_i .

Let T_A be the *automaton* abstracting the set of tasks T . The set of T_{ii} form the input actions of the automaton T_A . Similarly the set, T_{io} form the output actions of the automaton. Each output action T_{io} has a lower time bound of t_{ie} and an upper time bound of $t_{ip} - t_{ie}$. Every input action T_{ii} is assumed to enable the corresponding output action T_{io} .

Lemma 1: T_A is an extended automaton embodying the set of tasks T . Let α_s be a finite behaviour sequence of the automaton T_A , such that α_s contains all the normal input and output actions of the automaton in some order such that T_{ii} precedes T_{io} . If such a sequence α_s exists for the automaton T_A , then the given set of tasks is schedulable.

Proof: If α_s contains all output actions T_{io} preceded by their corresponding input actions T_{ii} , it implies that all output actions are executed and the timing constraints imposed on the actions are also satisfied. Hence all tasks associated with the actions of the automaton T_A are executed within the time constraints imposed. Hence α_s embodies a set of schedulable tasks.

3.4 Composition of Automaton

A mission critical system is composed of various subsystems. Hence, the automaton specifying the complete mission critical system, can be realised by a composition of automaton specifying different subsystems. Composition helps in modular construction. Let $normal(S_i)$, $fault(S_i)$, $recov(S_i)$ denote the normal, fault and recovery action sets the automaton S_i .

Two automaton S_i and S_j are compatible if

$$\begin{aligned} out(S_i) \cap out(S_j) &= \phi \\ int(S_i) \cap acts(S_j) &= \phi \\ fault(S_i) \cap recov(S_j) &= \phi \\ fault(S_i) \cap normal(S_j) &= \phi \\ \forall f_a \in fault(S_i) : (f_a \in acts(S_j)) \vee ((normal(S_j) \cup \\ recov(S_j)) \cap deactivationlist(f_a) &= \phi) \end{aligned}$$

If an action a is present in two compatible automaton S_i and S_j , the action a will have the following

⁴This implies that if the fault f_a can activate only a fault action in another automaton. If it does not activate fault action in another automaton, then the effects of the fault are internal to the automaton S_i .

property values in the composed automaton S .

default-action-status

- =OFF if the default-action-status is OFF either in S_i or S_j
- =ON if the default-action-status is ON in both S_i and S_j

activation-list = activation-list (a, S_i) ⁵ \cup activation-list (a, S_j)

deactivation-list = deactivation-list (a, S_i) \cup deactivation-list (a, S_j)

$lower(a, S) = \max(lower(a, S_i), lower(a, S_j))$

$upper(a, S) = \min(upper(a, S_i), upper(a, S_j))$

The result of the composition of compatible automaton signatures is

$S^6 = \prod_{i \in I} S_i$ where,

$in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$

$out(S) = \cup_{i \in I} out(S_i)$

$int(S) = \cup_{i \in I} int(S_i) \cup \{ \cup_{i \in I} in(S_i) \cap \cup_{i \in I} out(S_i) \}$

$normal(S) = acts(S) - \cup_{i \in I} fault(S_i) - \cup_{i \in I} recov(S_i)$

$fault(S) = acts(S) - \cup_{i \in I} normal(S_i) - \cup_{i \in I} recov(S_i)$

$recov(S) = acts(S) - \cup_{i \in I} normal(S_i) - \cup_{i \in I} fault(S_i)$

If $(S_1, L_1), (S_2, L_2), \dots, (S_i, L_i), \dots$ are environment free pairs, then the composed environment free pair (S, L) ⁷ is defined as the automaton S is formed by the above composition definition and the set of executions $L = \cup_{i \in I} L_i - int(S)$.

The **extended automaton** defined above together with the properties on its behaviours and the composition operator is capable of specifying a spectrum of faults and fault tolerant systems.

4 Specifying Fault Tolerance and Recovery Requirements using Extended I/O Automata

The faults that occur in systems can be broadly classified into *fail-stop*, *omission* and *Byzantine faults*.

FAIL-STOP faults:

The fail-stop faults causes the system to halt, and there is no further activity. Hence a FAIL-STOP fault action must deactivate all actions.

FAIL-STOP:

- default-action-status: ON
- activation-list = ϕ
- deactivation-list = $\{(a, exec)^8 : a \in acts(A)\}$ ⁹

⁵activation-list (a, S_i) is the activation-list of action a in automaton S_i

⁶Let $acts(S) = in(S) \cup out(S) \cup int(S)$

⁷It may be noted that (S, L) is also environment free.

⁸ $(a, exec)$ indicates that the action 'a' is deactivated upon execution of the FAIL-STOP action.

⁹The deactivation list includes the FAIL-STOP action itself,

In case of a system with *recovery* actions the deactivation-list is $\{(a, exec) : a \in acts(A) - recov(A)\}$

OMISSION faults:

Omission faults can be classified as either *send-omission* faults or *receive-omission* faults. When a *send-omission* fault occurs, a message that is to be delivered is not sent. This corresponds to the omission fault action deactivating a normal output action.

SEND-OMISSION FAULT:

- default-action-status: ON
- activation-list = $\{(ra, exec) : ra \text{ is a recovery action}\}$
- deactivation-list = $\{(no, enab) : no \text{ is a normal output action}\}$

Similarly, a *receive-omission* fault can be specified as

RECEIVE-OMISSION FAULT:

- default-action-status: ON
- activation-list = $\{(ra, exec) : ra \text{ is a recovery action}\}$
- deactivation-list = $\{(ni, enab) : ni \text{ is a normal input action}\}$

BYZANTINE faults:

A Byzantine fault causes the system to behave in an arbitrary manner and produce erroneous outputs.

BYZANTINE FAULT:

- default-action-status: ON
- activation-list = $\{(ea, enab) : ea \text{ is an output action}\}$
- deactivation-list = $\{(na, enab) : na \text{ is an normal output action}\}$

'ea' is an output action which produces arbitrary output which has no relation to the input which could have caused the 'na' output action to generate output in the normal mode of the automaton.

The **extended I/O automata** can also specify various kinds of fault tolerant systems. We now specify the following kinds of fault tolerant systems: *masking fault tolerant system*, *t-fault tolerant system* and *gracefully degrading fault tolerant system*.

Masking Fault Tolerant system:

In a masking fault tolerant system the effects of the fault actions are masked. Hence the activation and deactivation lists of a fault action are empty. The action signature is

MASKING FAULT TOLERANT SYSTEM:

Normal Actions:

...

Fault Actions:

- FAULT-ACTION-1:**
- (* an arbitrary fault*)
- default-action-status: ON
- activation-list = ϕ
- deactivation-list = ϕ

t-Fault Tolerant systems:

A t-fault tolerant system can mask atmost 't' faults.

since further faults cannot occur.

The occurrence of $(t+1)$ faults causes the system to fail-stop. So, the action signature of a t -fault tolerant system will have a T-FAIL-STOP action which is activated when t faults occur and is enabled when a $(t+1)$ th fault occurs.

t-FAULT TOLERANT SYSTEM:

Normal Actions:

...

Fault Actions:

T-FAIL-STOP:

default-action-status: OFF

activation-list = ϕ

deactivation-list = $\{(a, \text{exec}): a \in \text{acts}(A)\}$

Gracefully Degrading Fault Tolerant systems:

In a fault tolerant system, when a fault occurs a certain time elapses before a recovery action is initiated and the service provided by the system is resumed. A system that can provide services continuously inspite of the occurrence of faults although at a degraded level is called a gracefully degrading fault tolerant system. The fault actions in the action signature of a gracefully degrading fault tolerant system activate a normal action and also a recovery action. The activated normal action provides the system service at a lower level of functionality. The recovery action upon execution initiates another normal action which provides a possibly better service than the normal action initiated by the fault action. The normal action initiated by the fault action is deactivated by the recovery action.

GRACEFULLY DEGRADING SYSTEM:

Normal Actions:

NA-1:

default-action-status = ON

activation-list = ϕ

deactivation-list = ϕ

NA-2:

default-action-status: OFF

activation-list = ϕ

deactivation-list = ϕ

NA-3:

default-action-status: OFF

activation-list = ϕ

deactivation-list = ϕ

Fault Actions:

FA-1:

default-action-status: ON

activation-list = $\{(NA-2, \text{enab}), (RA-1, \text{enab})\}$

deactivation-list = $\{(NA-1, \text{enab})\}$

Recovery Actions:

RA-1:

default-action-status: OFF

activation-list = $\{(NA-3, \text{exec})\}$

deactivation-list = $\{(NA-2, \text{exec})\}$

5 Specifying Fault Tolerance Requirements of a Fire Control System of a CVES

The fire control system(FCS) of a combat vehicle(CV) has two major requirements. They are the fire control system should continue to direct fire at enemy targets inspite of electronic counter measures (ECM) undertaken by enemy. The firing accuracies may be reduced in the presence of ECM environment and the fire control system should ensure that *nothing bad ever happens* in the event of a malfunction in the firing mechanism of the combat vehicle.

The first requirement is a functional requirement, while the second is a safety requirement. The first specifies that the FCS should function as a gracefully degrading fault tolerant system. The second requires that the FCS should be fail-safe(i.e. fail-stop). The fire control system of a combat vehicle has four major functions. They are *loading of ammunition, target tracking, trajectory calculation* and *firing of ammunition*. These functions though separate are interrelated. All the functions are related to the functional requirement mentioned earlier. The first and the last functions in addition impinge on the safety requirement.

The ammunition loading function is modelled by a Load input action, target tracking by a Track action and firing by a Trigger action. These operator actions are *normal* input actions of the FCS automata. The faults in a FCS are a Mis-fire i.e. the ammunition doesn't leave the barrel of the gun inspite of a trigger action, a Loader-jam i.e. the loader mechanism is jammed and is no longer possible to load the ammunition, nor is it safe to leave the loader in a jammed condition and an ECM-P i.e. the presence of electronic counter measures causing reduction in the capabilities of the tracking radar. In the above three, the first two are internal *fault* actions. Hence they do not appear in the specification automaton. However these two fault actions should initiate a fail-safe (Fail-fire) fault output action, which shuts down all normal actions of the FCS. Fail-fire is a FAIL-STOP action. Two *recovery* input actions CLR-GUN and CLR-LOADER are defined to recover from these two faults. The recovery actions for the the ECM-P fault action is either an operator initiated ECCM (electronic counter counter measures) or the withdrawal of ECM measures by the enemy. These two are recovery input actions. An

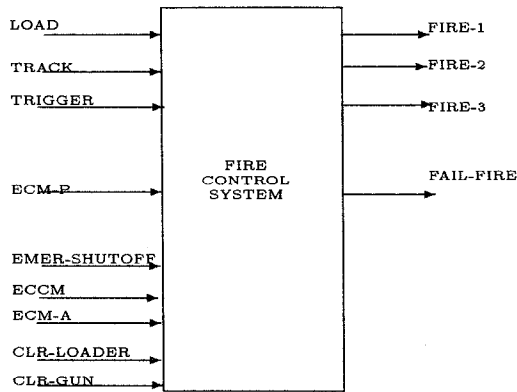


Figure 1: Fire Control System of a Combat Vehicle: Block diagram

emergency shut-off of the FCS is always possible in a combat vehicle. This cannot be a normal input action but is to be modelled as a recovery input action (Emer-shutoff). The presence of ECM environment causes a reduction in the firing accuracies. We define three *firing output* actions FIRE-1, FIRE-2 and FIRE-3 with decreasing firing accuracies. These are *normal output* actions. The fire control system of the combat vehicle has deadlines to meet. They are: 1) a *fire* action should take place within 5 seconds of a trigger action. Otherwise the trigger will result in a misfire. b) a *mis-fire* action should result in a *fail-fire* action (safety action) within 10 seconds of its occurrence.

The safety property imposed on the fire control system is *any FIRE action is always preceded by a TRIGGER input action (i.e. no spurious firing occurs)*. The liveness property of the fire control system is *a TRIGGER action is always followed by a FIRE or FAIL-FIRE action*¹⁰.

The fire control system of a generic combat vehicle can be modelled as in Figure 1. The action signature of the FCS is

FIRE CONTROL SYSTEM:

Normal Actions:

LOAD: (loads ammunition into the gun)

```
default-action-status: ON
activation-list = {(TRIGGER, exec)}
deactivation-list = {(LOAD, exec)}
```

TRACK: (track the specified target)

```
default-action-status: ON
activation-list =  $\phi$ 
deactivation-list =  $\phi$ 
```

TRIGGER: (fire the ammunition)

```
default-action-status: OFF
```

¹⁰It may be observed that the liveness property encompasses the second safety property.

```
activation-list = {(FIRE-1, exec), (FIRE-2, exec), (FIRE-3,
exec)}
```

```
deactivation-list = {(TRIGGER, exec)}
```

FIRE-1: (fire action with a 99% kill probability)

```
default-action-status: OFF
```

```
activation-list = {(TRIGGER, exec), (LOAD, exec)}
```

```
deactivation-list = {(TRIGGER, enab), (FIRE-1, exec), (FIRE-
2, exec), (FIRE-3, exec)}
```

FIRE-2: (fire action with a 80% kill probability)

```
default-action-status: OFF
```

```
activation-list = {(TRIGGER, exec), (LOAD, exec)}
```

```
deactivation-list = {(TRIGGER, enab), (FIRE-1, exec), (FIRE-
2, exec), (FIRE-3, exec)}
```

FIRE-3: (fire action with a 60% kill probability)

```
default-action-status: OFF
```

```
activation-list = {(TRIGGER, exec), (LOAD, exec)}
```

```
deactivation-list = {(TRIGGER, enab), (FIRE-1, exec), (FIRE-
2, exec), (FIRE-3, exec)}
```

Fault Actions:

ECM-P: (electronic counter measures - presence; modelled as fault action)

```
default-action-status: ON
```

```
activation-list = {(ECCM, exec), (FIRE-3, exec)}
```

```
deactivation-list = {(FIRE-1, exec)}
```

FAIL-FIRE: (failure in the FCS resulting in a shutdown: safety requirement)

```
default-action-status: OFF
```

```
activation-list = {a : a  $\in$  recov(FCS)}
```

```
deactivation-list = {a : a  $\in$  acts(FCS) - recov(FCS)}
```

Recovery Actions:

CLR-LOADER: (clear the loader jam)

```
default-action-status: OFF
```

```
activation-list = {(LOAD, exec)}
```

```
deactivation-list =  $\phi$ 
```

CLR-GUN: (clear the gun barrel in the case of mis-fire)

```
default-action-status: OFF
```

```
activation-list = {(LOAD,exec), (FIRE-1,exec)}
```

EMER-SHUTOFF: (emergency shut off by operator)

```
default-action-status: ON
```

```
activation-list =  $\phi$ 
```

```
deactivation-list = {(a,exec) : a  $\in$  acts(A)}
```

ECCM: (electronic counter counter measures of tracking radar)

```
default-action-status: OFF
```

```
activation-list = {(FIRE-2, exec)}
```

```
deactivation-list = {(FIRE-3, exec)}
```

ECM-A: (electronic counter measures absence)

```
default-action-status: ON
```

```
activation-list = {(FIRE-1, exec)}
```

```
deactivation-list = {(FIRE-2, exec),(ECCM, exec)}
```

PART-1: {FIRE-1, FIRE-2, FIRE-3} [0,5]

PART-2: {FAIL-FIRE} [0,10]

The PART-1 defines an equivalence class of the out-

put *firing* actions. The closed ended time interval $[0,5]$ specifies that the actions are to be executed within 5 seconds of they being enabled. This represents the first deadline imposed on the system. The PART-2 equivalence class defines that the FAIL-FIRE output actions is to be executed within 10 seconds of it being enabled. This represents the second timing constraint on the fire control system.

The FIRE actions (FIRE-1, FIRE-2, FIRE-3) are initially inactive. They are activated by the TRIGGER input action. The FIRE actions again become inactive after their completion. This ensures that a FIRE action is possible only after a TRIGGER input action. This satisfies the *safety* condition imposed on the FCS.

The set 'L' of executions for the FCS automaton are defined as {FIRE-1, FIRE-2, FIRE-3, FAIL-FIRE, \perp }. In order that the pair (FCS,L) is environment free [1], the strategy function f is defined as choose either a FIRE action¹¹ or a FAIL-FIRE action when the input is a TRIGGER action. For all other input actions choose \perp action.

Lemma 2: The (FCS, L) pair is environment free.

Proof: The FCS automaton chooses either a FIRE action or a FAIL-FIRE action when the input is a TRIGGER action. Other input actions result in a \perp action. These three different executions are the outcome of the strategy function f . Hence, $O_{(g,f)}(\alpha, I) \subseteq L$. Hence the pair (FCS, L) is environment free.

This FCS specification can be realised by composition of various sub systems of the combat vehicle.

6 Conclusions

I/O automata provide an appropriate and powerful model for discrete event systems consisting of concurrently operating components. In this paper, we have extended the basic I/O automaton model to capture notions of fault tolerance and graceful degradation in mission critical systems. We have defined an additional attribute called the action status (either active/inactive). At any given instant the automaton behaviour is defined by the set of active actions. The automaton signature evolves dynamically based on the set of active actions. Thus the proposed framework is capable of specifying dynamically evolving systems (e.g mission critical system like the fire control system of a combat vehicle). This extended I/O automata can also be used to specify internal design details of mission critical high assurance systems. A self-reconfiguring mission critical FCS for a combat vehicle

¹¹FIRE-1, FIRE-2 or FIRE-3

is being designed using the extended I/O automata.

A specification of a system can be realised by composing automata of different subsystems giving rise to different design alternates. In order to evaluate the alternates specific criteria are necessary. In mission critical systems service availability inspite of faults and responsiveness are two important criteria. The degree of service availability inspite of faults can be estimated by the *liveness* property of the extended automaton representing the mission critical system. If the *liveness* property L (in the tuple (A,L)), encompasses all the normal output actions, then it can be guaranteed that all services specified will be provided inspite of the occurrence of the faults. If the set of actions in L contains only a subset of the normal output actions, the degree of service comes down to that extent. This concept can be formalised to arrive at a mathematical expression for calculating performance in the presence of faults. We are currently working towards arriving at such quantitative performance metrics. This could be a quantitative analog to the qualitative measure defined in [4] viz. degree of fault tolerance.

References

- [1] R. Gawlick, R. Segala, J. Sogaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. In *Proceedings of ICALP '94*, 1994.
- [2] K. J. Goldman, M. D. Anderson, and B. Swaminathan. The programmers' playground: I/o abstraction for heterogeneous distributed systems. Technical Report WUCS-93-29, Department of Computer Science, Washington University, Saint Louis, June 1993.
- [3] C. Heitmeyer and N. A. Lynch. The generalised railroad crossing: A case study in formal verification of real-time systems. Technical Report MIT/LCS/TM-511, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1994.
- [4] K. H. Kim. Action-level fault tolerance. In S. H. Son, editor, *Advances in Real-Time Systems*. Prentice Hall, 1995.
- [5] L. Lamport. The temporal logic of actions. Technical Report Research report 79, Digital Systems Research Center, Palo Alto, California, 1991.
- [6] N. A. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufman, 1994.
- [7] M. Merritt, F. Modugno, and M. R. Tuttle. Time constrained automata. In *CONCUR 91: 2nd International Conference on Concurrency Theory*, Aug. 1991.
- [8] S. P. Rana. Mission-critical distributed systems: An overview of basic concepts, issues and techniques. Technical report, Department of Computer Science, Wayne State University, 1994.
- [9] S. P. Sarkar and S. P. Rana. Enhancing dependability of combat vehicles. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 1039-1044, Oct. 1995.