Unified System Level Fault Modeling and Fault Propagation

Hara Gopal Mani Pakala
Vignana Bharathi Institute of Technology, Aushapur, Hyderabad - 501301, India

Dr. KVSVN Raju
Computer Science & System Engineering, Andhra University, Visakhapatnam, 530003 India

Dr. Ibrahim Khan
RGUKT, Nuzvid- 521201, India

ABSTRACT
Fault modeling is an important step towards both understanding and validating the implemented system. The focus of the paper is to present a unified system fault modelling framework, including input related faults model based on system theory. The complex embedded system (CES) is modelled using sequence of communicating FSM (SCFSM) and the system faults are represented by two components of system fault vector - system behaviour fault vector and system communication fault vector. These two system fault vector components are related and are components of the "change parameter vector Θ", a general system level fault model based on Dynamic system theory. Input related faults are defined and their contribution is discussed. Faults propagation in mixed hardware software communication architecture is discussed, with the help of protocol example.

General Terms
Fault modeling and propagation, Embedded Systems development.

Keywords
System level Faults model, Fault propagation, system fault vector components, behaviour fault vector, communication channel fault vector, embedded system, Sequence of CFSM.

1. INTRODUCTION
The Embedded systems (ES) design and testing process are very challenging [1]. The growing system complexity and application domain demands are directly impacting the validation and or verification issues. Fault-based testing focuses on generating tests to detect particular faults, which is an important advantage compared to other testing approaches [2]. The two popular testing strategies, viz. conformance and composition testing rely on specific fault models [2]. The resulting tests are also effective in detecting faults in other classes [3]. Fault modeling is an important step towards development of tests. To efficiently test any system, it is important to know the possible failures, causes and the way they propagate. The first step is the identification and classification of design faults that occur during the early developmental stages. In general, fault taxonomy is the starting point for providing techniques and methods for assessing the quality [4].

Faults of a system can be classified by the phase in which they occur as follows: design faults or *errors* which appear in the design phase, fabrication faults which appear in the manufacturing phase, and operational faults which occur during normal operation. The focus of the paper is to present a unified system fault (USF) modelling framework, including input related fault model based on system theory. The CES is modelled using sequence of communicating FSM (SCFSM) and the system faults are defined by two components of system fault vector - *system behaviour fault vector* and *system communication fault vector*. These two *system fault vector components* are related and are components of the "change parameter vector Θ", a general system level fault (change) model which is based on Dynamic system theory [5]. Input related faults and their contribution to system fault vector are discussed. The USF model formalism is presented in section 3 which depicts the two types of system fault(s) and their relation with already reported embedded system fault models. Section 4 deals with faults propagation in mixed hardware software (MHS) communication architectures.

2. RELATED WORK
Fault modeling is an important aspect for the development of tests for systems. The functional test generation considers mainly two classes of fault models: Behavioral and Functional fault models. Fault models were developed and applied in different domains like Embedded Systems [6-7]; Software [4, 8-12]; and digital hardware systems including VLSI [13-14]. Testing at higher level of abstraction has a lot in common with software testing [6]. The test pattern generation methods can be classified into two main categories, namely, code oriented methods and fault oriented methods. A two-level fault model [6] is described to specify not only the control flow but also the data flow aspects for embedded system testing, based on simple FSMs and the extension of FSMs, i.e. with (p)-EFSM model. Apart from control flow faults [6], other error classes that are not represented by this model are similar to those of software [6]: Specific behavioural error classes and General data flow-related error classes. A timing fault model, the Mis-Timed Event (MTE) fault model, is proposed [7] to model timing-induced functional errors for analysing complex systems.

Software errors, their relationship with the development phase and system complexity are presented [8-9]. The results presented [9] are, (i) conditional faults (ii) operating faults and (iii) erroneous requirements are highly related with safety-related software errors. In several embedded software applications errors in understanding requirements and implementations are major sources of errors [9-10]. Software fault taxonomy for using in the development and evaluation of software in component-based systems is presented in [4]. Component-based software systems are classified into two main classes of faults: service and structure-related faults [4]. These

two classes are further categorized into seven type's viz., Syntactic, Semantic, Non-Functional, Connectors, Infrastructure, Topology and Other faults. A fault class hierarchy that relates literal, term, operator, and expression faults classes used in specification-based software testing is reported [11]. A test case that detects a fault of a stronger class will always detect the corresponding fault of a weaker class amongst the hierarchy. Though the study of faults was carried for Boolean specifications, the results are equally applicable to the code-based testing of predicates [11]. During software integration testing the emphasis is on interactions among modules via their interfaces. Four types of Integration Error (IE) occur when an incorrect value is passed through a module connection [12].To efficiently test any mixed hardware-software system in the design phase, it is important to know the possible failures and their causes.

Models of physical and fabrication faults [13] are commonly required for testing of digital electronic circuits or cores. A survey of behavioural level fault models for testing of cores (SOC) at higher level of abstraction than logic level is given in [13]. The survey indicates that a large number of fault models were developed for validating the designs in IC domain. To model a physical fault inside component like ALU, knowledge of the logic level implementation of the component is required. The co-validation fault models [14] are behavioural-level fault models that are classified by the style of behavioural description upon which the models are based. A co-validation fault model allows the concise representation of the set of all design defects for an arbitrary design. For Hardware functional verification, high-level faults are mapped into logic-level faults and correspondence between behavioural / RT level signals and logic-level nets established. The current co-validation fault models are classified based on the style of behavioural description as textual, control-data flow, state machine, gate level, application-specific, and interface faults [14].

The issue of error-propagation conditions was considered [15] in terms of 'impact' on the program execution in causing a detectable output error, using the notion of 'impact strength' as a quantitative measure of the impact. The paper introduced dynamic impact analysts' technique to determine the effect of components of program on the program output. The paper is on USF modelling framework, including input related fault model based on system theory. Faults propagation to lower level hardware faults in mixed hardware software architectures is discussed for protocol example.

3. COMPLEX EMBEDDED SYSTEM MODEL

The elements of any complex embedded system (CES) are an external process and an Embedded Processing Component (EPC) (hardware component board consisting of at least one dedicated processor, memory, special interface circuit and optionally may contain sensors and actuators). Sensors provide information about the current state of the external process, while actuators communicate to the external process the results (actions) of computing "controller law". The external process, which is known a priori, is a process that can be of physical, mechanical, or electrical nature; is controlled and implemented in real time by mixed hardware software or hardware only. The available strategies for simplifying a complex system in general are: abstraction, partition and segmentation.

The embedded system specification consists of the specification of its environment (external process) and it's EPC, in some formal notation, like state transition model [6] or sequence of communicating FSMs [16-17]. Here Complex embedded systems are modeled as a sequence of communicating FSM (SCFSM) A_i, i=1... k. This model supports well the sequential integration of components of CES or even specific sequence. It is assumed that the component FSM [16]:

(a) A_i is deterministic FSM which communicate asynchronously with each other through bounded input queues, in addition to their communication with the environment through their respective external ports.

(b) In response to an input produces only an internal or an external output

(c) The system has at most one message in transit, i.e. the next external input is submitted to the system only after it produces an external output to the previous input. Then the collective behaviour of the communicating FSMs can be described by a finite *product machine (PM)*.

(d) System does not fall into a live-lock and the component machines are deterministic. Under these assumptions, *composed machine* $(\mathrm{CM}) = A_i \Diamond A_j ; (i \neq j)$ is deterministic.

The complex embedded system $\mathcal{XS}$, with the above assumptions, can be represented as

$\mathrm{XS} = \prod_i A_i ; (i = 1 \ldots k)$; where the product sign represents the sequence.

The embedded system can be broadly divided into two orthogonal operations: computations and communications. These operations are separated in two types of components [18]: *applications* and *interfaces*, and communications to and from these types of components are indicated in figure1. The point-to-point communications from '*applications*' are denoted as *orders* and *results*; and from '*interfaces*' as *requests* and *responses.* For example *port_ send* (port, data, size, mode) and *port_ receive* (port, data, size, mode) orders of applications which the *interface* will execute.

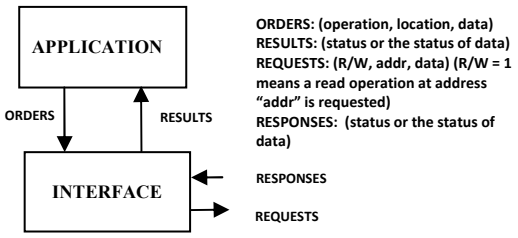

Figure 1: Computation, Communication, and Interconnection

The system partitioning and the design of the communication will be the most important tasks that will greatly influence the performance of the whole system. The various abstraction levels of embedded system components (from implementation point of view) are given in table-I. It is to be noted that *partioning* is not possible when the CES has *emergent properties* and Segmentation is difficult, if not impossible, for highly concurrent processes and associated behaviour. In the next section system level fault model is defined, based on CSFSM representation.

4. UNIFIED EMBEDDED SYSTEM FAULT MODEL

In section 3 the complex embedded system $\mathcal{XS}$, is represented as $XS = \prod_i A_i; (i = 1 \ldots k);$ where the product sign represents the sequence. A "linear fault operator L" can be defined that modifies the operations of individual component machine A_i either the behaviour [6,16-17] or alternately modifies the events of communicating channel [19-21] between the fault-free, communicating component machines A_i and A_j. Formally combinations of these two operations are also can be defined on $\mathcal{XS}$. For simplicity, "fault operators" on behaviour and communication channel only are considered in this work. The effect of linear "fault operator L" on the system can be written as:

$$L\{XS\} = XS_L = L\{\prod_i A_i\}, (i = 1 \ldots k); (1).$$

The faulty behaviour of FSM is another FSM known as mutation machine [16]. Therefore a fault relative to some F be defined as any mutation of F, say F', such that $F \neq F'$ and the fault model be a set $\Re = F^F$ of such all possible modified FSM's, is the *fault domain*. In general a fault model is defined as [16]: A fault model for F is a set $\Re = F^F$ of mutations of F, such that $F \neq F'$ for all $F' \in \Re$ the *fault domain*.

Representing the Mutation Operator by "Ψ" its effect on the system XS can be written as:

$$\Psi[XS] = XS_\Psi = \Psi[\prod_i A_i] = \prod_i \Psi[A_i]; (2)$$

Assuming that the operations of "Mutation Operator Ψ" on each of the component machines are disjoint and denoting the operations of "Mutation Operator Ψ" on each of the component machines by ψ_i the overall effect on the system can be written as

$$XS_\Psi = \prod_i \Psi[A_i] = \prod_i \{(\psi_i)[A_i]\}; (3)$$

The effect of ψ_i on individual component machine A_i can be the error prone version of that machine the 'mutation machine A_i'. The mutation machine may have any of the possible faults [16]: transition has an output fault, transfer fault, missing state and an additional state.

$$XS_\Psi = \prod_i \{(\psi_i)[A_i]\} = \prod_i [A_i']; (4)$$

Denoting $\prod_i [\psi_i] = \Omega_B; (5)$

and calling it as the *subsystem or EPC behaviour fault (EBF) vector*; the equation (4) describes the sequence of mutation machines in the given system.

Alternately, the machine can be assumed to be fault free and the fault(s) if any can be incorporated in the communication channel (or link) between the machines [19-21]. This is similar to separating and encapsulating communication and the computation [19]. It suffices to consider *communication channel fault* (CCF) and assume fault-free component machines. The assumption of the component machine being fault free is very useful since it allows to completely ignoring the internal structure. Based on functionality the subsystem (level) faults can be further categorized into three groups [19-20] based on functionality. It also supports hierarchy. When each process is decomposed, new communication links become visible, and based on the "internal" faults in the original process, the newly visible communication links are modelled as faulty. Referring to figure 2(a), two fault-free communicating component machines A_i and A_j and fault free communication channels V and Z are shown. Figure 2(b) describes the effect of "Fault Operator- Φ" on the individual communication channel 'Z' and the other communication channel 'V' is shown to be fault free, between two fault-free communicating component machines A_i and A_j. The effect of "Fault Operator- Φ" on the communication channel Z can be written as Є = Φ (Z).

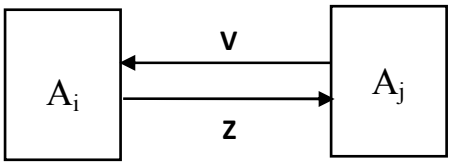

Figure 2(a): A fault free Communication Channel

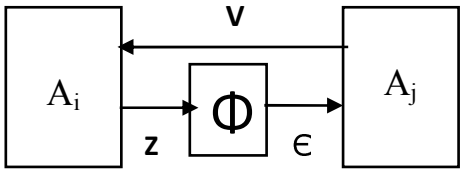

Figure 2(b): A communication Channel fault

Following same lines, the operations of "Fault Operator- Φ" on the system $\mathcal{XS}$ can be written as

$$\Phi[XS] = XS_\Phi = \Phi[\prod_{ij} A_{ij}] =$$

$$\prod_{ij} \{[A_{ij}][\phi_{ij}(,)]\}; (i, j = 1 \ldots k; and, i \neq j); (6).$$

Let V_i and Z_j be the communication channels between the two machines A_i and A_j, then assuming the effect of "faulty process Φ" on the communication channels (i →j) and (j → i) are independent and may be denoted by $\phi_i(V_i), \phi_j(Z_j)$. Using this effect of fault operator can be written as: $\Phi[A_{ij}] = A_{ij}[\phi_i(V_i), \phi_j(Z_j)]$. The effect of communication channel "Fault Operator- Φ" on the system $\mathcal{XS}$ can be represented as:

$$XS_\Phi = \prod_{ij} \Phi[A_{ij}] = \prod_{ij} \{A_{ij}[\phi_i(V_i), \phi_j(Z_j)]\};$$

$$(i, j = 1 \ldots k; and, i \neq j); (7)$$

The above equation (7) can be rewritten as:

$$XS_\Phi = \prod_{ij} [A_{ij}] \prod_{ij} [\phi_i(), \phi_j()]\};$$

$$(i, j = 1 \ldots k; and, i \neq j); (8)$$

Denoting

$$\prod_{ij} [\phi_{ij}()] = \prod_{ij} [\phi_i(), \phi_j()] = \Omega_C; (9)$$

as the *subsystem communication channel fault (CCF) vector*, equation (8) describes the sequence of CFSMs interconnected by faulty communication channels. This model can support multiple channel faults, by associating several ports with the "Fault Operator- Φ", modelling interactions between several channels and/or processes. Equations (5) and (9) indicate that the effect of the "linear fault operator's Ψ and Φ" respectively is to divide the overall system faults, into two hierarchical

component vectors of faults. It is to be noted that these component of fault vector do not represent orthogonal faults. These two component *system fault vectors* are related to the "change parameter vector Θ" shown in Fig. 3, a general system level fault model, as $\Theta = \Omega_B \cup \Omega_C$. The system level fault model is based on Dynamic system theory [5] which assumes that models of the system is available and considers that all the events like - deviation, change, failure, fault, damage and malfunction, are incorporated by *a change in the parameter vector* of a model of the system. Referring to Fig.3 neglecting the dynamics for the moment, the observed data Y can be expressed as:

$$Y = f(\Theta, U + \gamma_i, W_S + \gamma_o + W_o); (10)$$

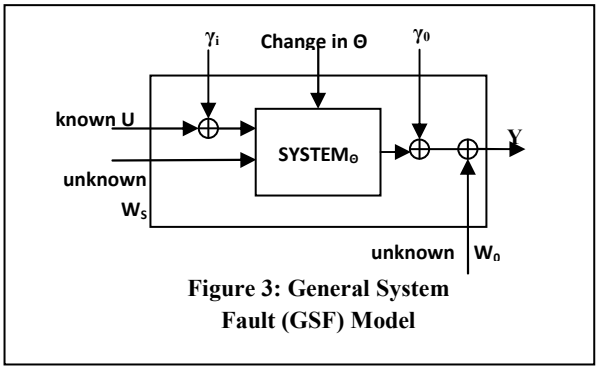

Figure 3: General System Fault (GSF) Model

Input U is assumed to be known (measured). The unknown quantity W_s stands for non-measured inputs, unknown non-stationary excitation or perturbation of the system, and input noise. The unknown quantity W_o stands for output noise. If the system is error free, then the output noise W_o depends on W_s only. Equation (10) shows three types of faults: the first two types of faults Υ_o and Υ_i are additive type and appropriately model some faults due to *sensors and actuators*. The third type of faults modeled by the *change in the parameter Θ*, is often referred as component or system faults.

These systems (processes) can be static or dynamic, linear or nonlinear. They are all subject to component faults. In the case of linear dynamic systems, it is useful to assume that faults as multiplicative faults, because they affect the input-output transfer function in a multiplicative manner [5]. Equation 10 is the starting point for fault diagnosis and isolation of dynamic systems.

4.1 Input Related Faults

The cause-effect analysis [9] shows that conditional faults, operating faults and erroneous requirements are the main contributors of of safety related software faults. Erroneous requirements and operation faults are required to be identified and resolved very early as they warrant modification of software [10] else they create safety related problems. Conditional faults are closely related to limit values. The two input related faults that are due to conditional faults are: *input domain fault* and *input bound-limit fault*.

4.2 Input Domain fault

The system $\mathcal{XS}$, can also be represented as a set of function or mappings according to some group of specifications, *Spec*, from a set of input values (its *domain,* $\mathcal{D}$) to a set of output values (its *range,* $\Re$), as shown in figure 4. A system which implements specification *Spec* should also map from $\mathcal{D}$ to $\Re$. For real numbers R will be m-dimensional space of real numbers - R^m. However for CES the external process is known, which is physical process and under the control of a MHS platform. All physical processes are required to be controlled within certain limits, and as such restrict the input to specific bounds in the domain $\mathcal{D}^B \in \mathcal{D}$, which in turn limits the range to $\Re^B$; $\Re^B \in \Re$. But the input values $U \in D^U \notin D^B \in D$ may also contribute to values $\Re^U \notin \Re^B \in \Re$, which is outside the range. This implies the input values U contribute to violation in domain and denote this as a faulty domain $\mathcal{D}_F$. The input values U that contributes to violation in input domain and triggers faulty result, as shown in figure 3, is called input domain fault – $\mathcal{D}_F$.

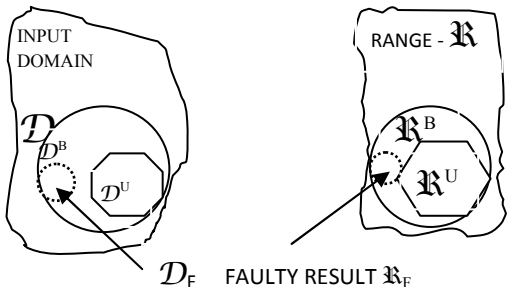

Figure 4: Input Domain Fault

This certainly arises among complex systems that have more than one operation modes, like Radar, Sonar or Missile and also software initiating fault referred as mode confusion [22], an automation design related problem. The application domain dictates the nature or criticality of its effect. This is also similar to the idea of Application Domain Modeling or Input validation analysis [23] and Data Mutation. For obtaining an insight into the nature of program faults, a distinction between the syntactic and the semantic nature of faults is drawn [4]. Obviously, semantic fault size depends on the input domain and output range. For example, if a program P computes on the domain-$\mathcal{D}_F$ produces faulty result $\Re_F$ contain values both in $\Re^B$ and outside $\Re^U$. Note that the domain and the range can be considered for an entire program, an individual program component, a program path or simply a single program location.

4.3 Input bound-limit faults

For physical systems the input values are required to be limited to certain bounds in their range of values: called input bound limits. For a CES, any variation of input that causes one or more input bound limits to be violated is defined as an *input bound-limit fault.* The input bound-limit violation may be due to any input parameter that is specified, including time. A *p*-dimension bound vector, λi, represents the variations in the limits of the input U. A set of these vectors, $\Lambda = \{\lambda 1, \lambda 2 \ldots \lambda N\}$, define the bounds limit in the input vector U. The mapping of the input vector 'u_i', to the output vector 'y_i' is given by $y_i = f(u_i)$. Denoting R^m as m-dimensional space of real numbers R^m ,the *region of acceptability*, *Ra*, is defined by a set of vectors Λ *Lowi* and Λ *Maxi* , where $i = 1, \ldots, p$, representing upper and lower bounds limits. For every 'u_i' $\in R^U$, the corresponding 'y_i' $\in R^Y$

is mapped inside R_a or outside it. This implies that a region of acceptability, R'_a is defined in the physical output space, by evaluating the function $f(\ )$ and is defined as

$$R'_a = \{ y_i \mid f(u_i) \in R_a \}; (11)$$

This map in the physical output space domain provides a definition of a fault in terms of the CES physical parameters. A simple example of input is: Amplitude shift keying (ASK) is used in many communication applications and especially M-ary amplitude–shift keying (MASK).

An M-ary amplitude-shift keying signal s (t) can be defined by

$$s(t) = D_i Cos\ (2\pi f_c t); (0 \le t \le T); (12)$$
$$s(t) = 0; else$$

where $D_i = D[2i - (M-1)]$ for i=1, 2, … (M-1) and M > 4.

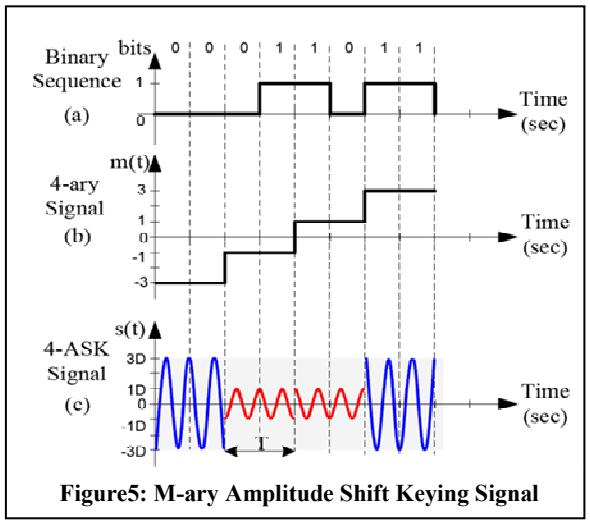

Figure5: M-ary Amplitude Shift Keying Signal

MASK signal, for M=4 is shown in figure 5. Figure 5 shows the 4-ASK signal sequence generated by the binary sequence 00 01 1011. The carrier frequency f_c and amplitude D_i have tolerances as per the application. Typical tolerance for amplitude D and frequency f_c are 1%. Beyond tolerance limits the detected binary data will not be recognized and resulting in malfunction. This application shows tolerance of even time duration T.

The violation of timing constraints on signals within a complex system can create *timing-induced functional errors* which alter the value of output signals without affecting output signal timing. One such error is Mis-Timed Event (MTE) [7] fault and it is 'association of pair of signals' at the 'wrong-time' resulting in receiving 'faulty-data'. This is communication channel output data 'timing association error' for subsequent use. Similar design error is possible in data exchange protocols between any two processes or tasks. The input related faults may trigger or contribute to either of the system fault vector components. But the communication fault vector is logical choice for analyzing input related faults, as the region of acceptability R'_a is defined in the physical output space. Naturally the input noise component W_s generally add up and adversely affect the region of acceptability. Sensors and actuators are likely to develop malfunction more due to input related faults. Several software program defects, like array or string function, are related to improperly bound inputs. An example of this type of defect is the buffer overflow - where memory outside of the intended buffer (or array) is accessed. For example, many arrays are created dynamically meaning that the size is only known at run-time. In addition to the required data value, the proper path is also necessary, to ensure that the array reference is executed. Complete software verification for arbitrary programs with unbounded memory is un-decidable.

5. FAULTS PROPAGATION

For preventing fault propagation, the requirements are [24], "abstractions and models are stable even in case of failures (error containment principle)", which implies partioning of system into independent fault containment units. Generally the *error containment principle* is violated in systems other than Fault Tolerant Systems, allowing fault propagation. In such case the fault diagnosis may be performed at lower levels of abstraction [24]. Faults propagation will be analyzed with reference to different levels of communication abstractions of the embedded systems, shown in table-I. Design faults, hardware, software or *HSI*; affect various levels of system hierarchy in the target MHS platform (TMP) architecture. The error occurrence to fault manifestation is highly implementation dependent.

In section 3 the embedded system operations are separated in to two types of components, [18]: *applications* and *interfaces*, as shown in figure1.Though errors are possible in both 'computations and communications' operations, their propagation greatly depends on implementation of the 'communications architectures' and for this reason the CCF model defined in section3 is very useful for testing. Critical part in embedded system is the hw/sw interactions that are governed by communication and interfacing architectures. Communication includes protocols, physical interfaces and physical channels that carry data and control. One of the general system level communication primitive operations are: *port_ send* (port, data, size, mode) and *port_ receive* (port, data, size, mode).They are example ORDERS of an *application*.

In the TMP architecture the implementation choices are: (i) hardware (ii) and hardware-software (mixed). The communication primitive like the *port_ send* ( ) and *port_ receive* ( ) can be implemented using the SEND and RECEIVE functions. To obtain an insight into the hw/sw interaction, and its propagation through the hw/sw communication, a simple *synchronous wait protocol* example is considered. The synchronous protocol is sufficient to bring out the hw/sw interaction issues very clearly.

5.1 The synchronous wait protocol

The *synchronous wait protocol* shown in figure 6. Though it is simple the completion of communication is certain and one item per cycle data transfer takes between the two functional modules. The SEND and RECEIVE together synchronize the communication by a pair of 'SndRd and RcvRd' signals, as shown in Figure 6. The timing diagram shows that both are ready in the same state. But in practice either of them may have to wait for one more cycle and the sender has to ensure the data remains valid during those cycles on the data bus. Note both modules initiate the transfer of data and there is no notion of master / slave. This protocol is widely used in asynchronous circuit implementations and can also be used with a synchronous

positive edge-trigged design. This scheme requires at least two clock cycles for each transfer to account for the explicit acknowledgment. In this example clocking issues are not addressed. The next section describes SEND function

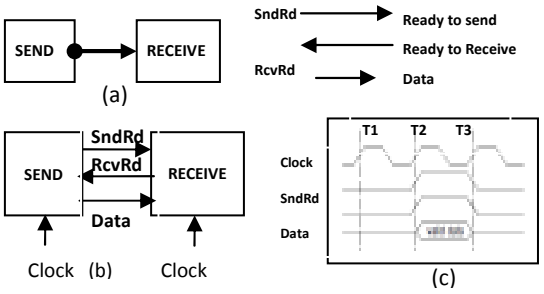

Figure6. Synchronous wait protocol

implementation in hardware.

5.2 Hardware implementation

One possible SEND-function block implementation for an 8-bit data, using 74xxx541 buffer / driver with tri-state outputs, device is given in Figure 7. In this case, there is a direct data connection between SEND and RECEIVE. The implementation assumes that two independent 'SndRd and RcvRd' signal-wires are available. It is further assumed that 74xxx541buffer/driver I0-I7 pins are connected to the source-data-bus or the block itself is generating data. The two inverters (from 7406 or equivalent) use the independent signal-wires 'SndRd and RcvRd' for generating OE1bar and OE2 bar control signals, as shown in figure 7. If data is required to be latched for reading by RECEIVE-function, then the buffer can be replaced with 8-bit octal transparent D-latch (74x573 or its equivalent). RECEIVE-function can be implemented similarly. If data is valid for rising edge of clock then 8-bit octal transparent positive edge triggered D-latch (74x574 or its equivalent) can be used.

The generation of independent 'SndRd and RcvRd' signal

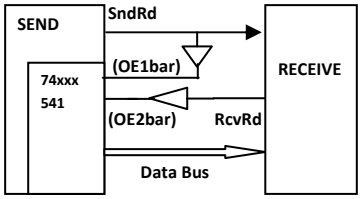

Figure7. SEND - Hardware Implementation

becomes important, as the remaining devices are fairly reliable and mal-function is expected after long use only. The devices are expected to develop "stuck-at-1 or 0" type of faults, in general, after fair use. However such faults are easily detected during independent hardware tests. The RECEIVE block hardware implementation is similar. The independent generation of 'SndRd and RcvRd' signals, if error prone, fault is generated.

5.3 Synchronous hardware-software implementation

The hardware component of the synchronous input interface is made up of the flag and the input bus interface is similar to the tri-state buffer described. The software component of the synchronous interface reads data from the hardware component, and can be declared as a procedure *Rcv_synch_rd* (*v, c, RD*) where the parameter *v* is the holder of the input message, *c* is the selected address and RD is a control signal for reading. The software module of the synchronous interface reads data from the hardware component of figure 8 and RD is a control signal for reading.

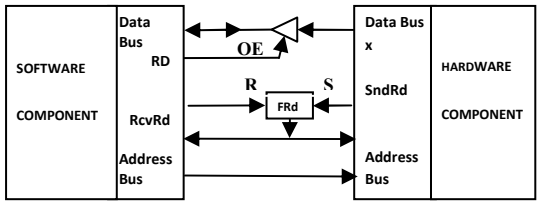

Figure8. SEND – Synchronous Hardware-Software Implementation

Here, the software program waits for the flag '*FRd'* to be set to indicate availability of an input data on the data Bus. The 'RcvRd' signal is connected to both HW and SW components and shown as can be reset by RD, to be consistent with the example. The *reset* wire of the flag is linked to the control wire 'OE', which is driven by the control signal RD. For implementing the example protocol 'SndRd' is connected to 'S' and the 'soft-module' reads the flag-FRd; and also generates 'RcvRd' signal. The generation of 'RD' for reading operation will reset the flag-FRd and the 'RcvRd' signal. Any error in the 'soft-module' prevents input data from hardware component. Apart from timing considerations, initialization of hardware and sequencing of software is important, for error generation. It is possible that the 'flag-FRd' is already set before HW component initiated SEND operation, resulting in reading wrong data or preventing in sending data.

5.4 Mixed hardware - software communication architectures

More general system level communication primitive operations are: *port_ send* (port, data, size, mode) and *port_ receive* (port, data, size, mode). Based on the design partitioning, the interface need to connect two hardware modules, two software modules, or hardware and a software module. In the case where two processes that communicate through ports are mapped to hardware implementation (as in section 4.2), code or software is not involved in handling specific communication. Hardware to software communications can be implemented by either interrupts or using memory-mapped addresses for the polling case. One solution is shown in Figure 10. The bus adaptation layer for the hardware module sends and receives data from the bus. In the case of a memory-mapped communication, a device driver is also required to reside in the processor, for monitoring the bus for activity in the memory-mapped region. The device driver is responsible for transferring data from the bus to the processor memory, to a port structure. The software process will access the port data structure retrieving data and updating event flags.

The application programs communicate with the external world via function calls to the appropriate send and receive operations. The send and receive routines are implemented using the memory "read" and "write" instructions if memory mapped I/O

is used for the specific channel, or the corresponding special programmed I/O instructions if instruction programmed I/O is used. For *interrupt-driven I/O* service routine scheme, the "send" and "receive" routines in software are each split into two operations. In the case of a "send" operation, the processor transfers the data to the I/O unit and proceeds with other tasks. After completion only the "receive" operation is initiated. The direct memory access is similar to the single "send" and "receive" operations above, but parameterized for block transfers. In the interrupt-based communication, the actual data is still transferred through a memory-mapped location to the port structure.

5.5 Discussion on communication architectures

In the previous section SEND block and communication primitive implementations are described. Referring to table-I both "SEND and RECEIVE" are 'event or signals' type in the abstractions levels of communication. A system level communication primitive like *port_ send* ( ) makes use of the same. If there is an error in SEND, it is triggered when a higher level abstraction like *port_ send* ( ) makes use. It is clear that if an error exists at a lower level abstraction, then it is likely to be triggered when a higher level abstraction uses it. As such tests designed using higher levels of abstraction, are likely to detect errors at the lower levels also [3]. Physical hardware is the lowest level. A module or routine's assumptions and initialization errors result in errors similar to hardware errors.

For example OE1bar and RcvRd signals error effect is same as SEND circuit malfunction. It can be said that hardware errors are triggered by the operations or functions that make use of the erroneous part. Based on these examples, it can be strongly stated that-

1. An error at lower level, like hardware, impacts all related higher levels of abstractions.
2. Also initialization errors, control signal errors mimic hardware errors.
3. Error in Software function / module, will effect only when used by other modules.
4. In a data dependent function/module the error may be triggered under certain conditions. Similarly error may not show if alternate path exists.
5. Error at a higher level abstraction may not impact adversely all related lower levels, if an alternate path exists [15].
6. Possible error prone spots are: Hardware dependent Software: Initialization of hardware flip-flops, registers, enable /disable and other control signals, Initialization of flags, pointers, semaphores, mailboxes, and etc.
7. During integration, four types of Integration Error (IE) occur when an incorrect value is passed through a module connection [12]. The incorrect *hardware software interaction* passes through hw/sw communication and results into a fault at the various interface architectures used in the application.

It is also possible that error propagation can be prevented by the intermediary states, either by ignoring the fault or mapping it to a proper state [15]. As such integration tests are to be designed to identify faults. Hence system level functional tests created to verify the system design will be able to detect errors in software and assure correct function [10]. Careful design of functional tests and proper reuse of the same tests during integration are likely to give better results.

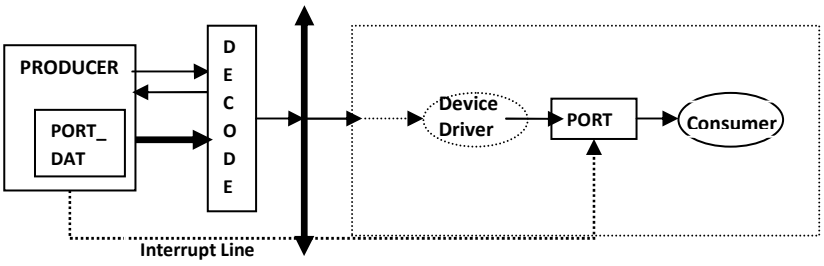

Figure10. Hardware to Software Communication

Table1. Embedded system levels of abstraction

Hardware [25]	Software [26]	Computation [27]	Communication[27]
Domain	Application	Concepts	Services
Transaction	Application Program Interface	Behaviours	Tokens
Behavioral + data types	OS + Drivers + Boot firmware	Processes	Messages
Behaviour	Hardware abstraction layer (HAL)	Processes	Buses/Ports
RTL	---	FSMD	Events or Signals

6. CONCLUSIONS

A hierarchical unified system fault (USF) modelling framework is proposed for the MHS platform. The complex embedded system (CES) is modelled using sequence of communicating FSM (SCFSM) and the system faults are represented by two components of system fault vector - *system behaviour fault vector* and *system communication fault vector*. These two component *system fault vectors* are related to the "change

parameter vector Θ" of generic system level fault model based on Dynamic system theory. Input related faults and their contribution to system fault vector are discussed. The propagation of faults in mixed hardware software architecture is discussed with examples.

7. REFERENCES

[1] Tom Henzinger, Joseph Sifakis. "The embedded systems design challenge". Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science, Springer, August, 2006.

[2] Bernhard K. Aichernig, Martin Weiglhofer, Franz Wotawa "Improving Fault-based Conformance Testing", Electronic Notes in Theoretical Computer Science 220 (2008) 63–77.

[3] Vadim Okun a,b Paul E. Black a Yaacov Yesha, Comparison of Fault Classes in Specification-Based Testing , Preprint submitted to Elsevier Science 2 April 2004

[4] Leonardo Mariani, "A Fault Taxonomy for Component-based Software" Electronic Notes in Theoretical Computer Science 82 No. 6 (2003).

[5] M. Basseville and I. Nikiforov (1993). Detection of Abrupt Changes: Theory and Applications. Prentice Hall, N.J.

[6] A Guerrouat, and Harald Richter, Adaptation of State/Transition-Based Methods for Embedded System Testing" Proceedings Of World Academy Of Science, Engineering And Technology Volume 10 December 2005 Issn 1307-6884

[7] Qiushuang Zhang and Ian G. Harris, "A Validation Fault Model for Timing-Induced Functional Errors", International Test Conference 2001 (ITC'01) Baltimore, Maryland, October 30-November 01.

[8] Boris Beizer, " Software Testing Techniques", Second Edition The Coriolis Group ISBN: 1850328803 Date: 06/01/90

[9] R. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In IEEE International Symposium on Requirements Engineering, pages 126–133, San Diego, CA, 1993. IEEE Computer Society Press.

[10] T. L. Bennett and Paul Wennberg, "Eliminating Embedded Software Defects Prior to Integration Test", Triakis Corporation ; VSILTriakis2005Article-c

[11] Man F. Lau, Yuen T. Yu, "An Extended Fault Class Hierarchy for Specification-Based Testing" ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 3, July 2005, Pages 247–276.

[12] P. Prema, and B. Ramadoss, "A New Type of Integration Error and its Influence on Integration Testing Techniques", World Academy of Science, Engineering and Technology 42 2008.

[13] Abramovici, M., Breuer, M., and Friedman, A. D., Digital Systems Testing and Testable Design. IEEE, 1993

[14] Ian G. Harris, "Fault Models and Test Generation for Hardware-Software Co-validation" IEEE Design & Test Volume 20 , Issue 04 (July 2003) Pages: 40 - 47

[15] Goradia, T., "Dynamic impact analysis: a cost-effective technique to enforce error-propagation", *Proc. Int. Symp. On Software Testing and Analysis (ISSTA '93)*, Cambridge, MA, U.S.A., ACM Press, pp. 171-181 (June 1993).

[16] Petrenko, A., Yevtushenko, N., Bochmann, G., and Dssouli, R. (1996). Testing in context: Framework and test derivation. Computer Communications, 19: 125-140.

[17] Hara Gopal Mani Pakala, K. V. S. V. N. Raju and Ibrahim Khan, " Integration Testing of Multiple Embedded Processing Components", Springer Advanced Computing, Communications in Computer and Information Science, 2011, Volume 133, Part 2, 200-209.

[18] J. Schmaltz and D. Borrione, "Validation of a Parameterized Bus Architecture Model," *Proc. TIMA-VDS*, 2003.

[19] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design", *in Design Automation Conference*, pp. 178–183, June 1997.

[20] D. Panigrahi, C. N. Taylor, and S. Dey, "Interface based hardware/software validation of a system-on-chip", *in High Level Design Validation and Test Workshop*, pp. 53–58, 2000.

[21] Hara Gopal Mani Pakala, Dr. Raju KVSVN and Dr. Ibrahim Khan, "Sensors Integration in Embedded Systems", International Conference on Power, Control and Embedded Systems (ICPCES 2010) Nov 29, 2010 - Dec 1, 2010 Allahabad, India

[22] Victor Vui-Kiat Chong, "Heuristics for Mitigating Mode Confusion in Digital Cameras" A thesis submitted in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in the Department of Computer Science, University of Victoria, 2006

[23] Jane Huffman Hayes, Jeff Offutt, "Input Validation Analysis and Testing" pp1-29, Empirical Software Engineering ©2005 Kluwer Academic Publishers.

[24] Hermann Kopetz, "The Complexity Challenge in Embedded System Design," isorc, pp.3-12, 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008.

[25] John Sanguinetti, Abstraction Levels and Hardware Design, EDA Design Line (07/17/07).

[26] W. Ecker, V. Esen, T. Steininger, and M. Velten. HW/SW interface - implementation and modeling. In W. Ecker, W. M¨uller, and R. D¨omer, editors, *Hardware-dependent Software - Principlesand Practice*. Springer, 2008

[27] Andreas Gerstlauer, Daniel D. Gajski, "System-Level Abstraction Semantics," CECS, UC Irvine, Technical Report CECS-TR-02-17, July 2002.