Chapter 4
Analysis Based on Components and Architecture

Abstract This chapter presents methods for modelling and analysing the component architecture of a system. It deals with the information that can be deduced from components and the way in which the components are connected. Simple and aggregated components are described in terms of their generic properties, which include the service offered by a component in different modes of operations and the conditions under which component faults occur. Properties of selected simple components are discussed and their aggregation into generic components at a higher level is illustrated. Formal methods for describing generic components are introduced. Algebraic and graph-theoretic methods are employed to analyse the propagation of faults through the faulty system.

4.1 Introduction

Architecture models describe a system as a set of interconnected components. This statement is true at any hierarchical level. Low-level components, such as sensors and actuators, are directly interfaced with the process. They provide low-level services: measurement of, or action on some specific process variable. Subsystems, composed of several components, can also be considered. They form higher-level devices which can be aggregated to even higher levels. Higher-level devices provide higher-level services. The primary track control loop in the ship example provides the ship to follow a desired path through shallow water; the cooling unit in a chemical reactor provides control of an exothermic reaction. The highest aggregation level is that of the system itself, when it is considered as one single component.

At any considered level, a component, whether simple or aggregated, can be described by its generic model. The services it provides can be organised in a number of use-modes. At the highest aggregation level, each of the system use-modes is associated with a given number of objectives to perform, and the system services are used to achieve those system objectives. The definition of the use-modes set, and for each use-mode the associated objectives result directly from the specification of the considered system.

M. Blanke et al., *Diagnosis and Fault-Tolerant Control*,
DOI 10.1007/978-3-662-47943-8_4

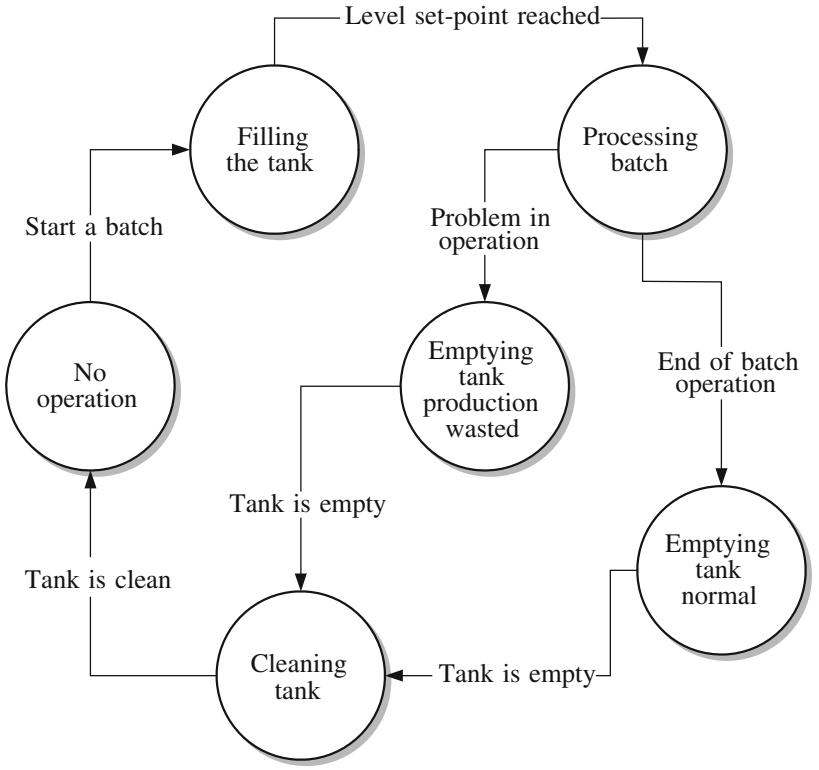

Fig. 4.1 Automaton of a batch process illustrated through use-modes

Example 4.1 Tank system
Suppose that the tank system is used in a food industry batch production process, where the processing of each batch needs the temperature to be controlled at a given value during a given period of time. Six different use-modes (UM) can be distinguished:

UM 0: No operation,
UM 1: Filling the tank,
UM 2: Processing the batch,
UM 3: Emptying the tank via the normal pipe,
UM 4: Emptying the tank via the "lost production" pipe, and
UM 5: Cleaning the tank.

The associated use-mode automaton is illustrated in Fig. 4.1.
There are six objectives associated with the different use-modes, namely

- Objective 0: No action (UM 0),
- Objective 1: Reach the full level set-point (UM 1),
- Objective 2: Regulate the temperature (UM 2),
- Objective 3: Reach the empty level set-point (UM 3 and UM 4),
- Objective 4: Clean the tank (UM 5), and
- Objective 5: Preserve the environment (UM 1, 2, 3, 4 and 5). □

The analysis of fault tolerance should answer the essential question whether a given system, in a given fault situation, is still able to achieve its objective. Overall objectives are associated with use-modes, and they are achieved using the services offered at the system level. Thus, the analysis of fault tolerance first needs the generic model to be derived at the system level. This can be done by defining procedures which aggregate low-level generic models into higher-level ones. The second step is to analyse the situation when faults appear, in order to conclude about the way services are affected.

This chapter presents two approaches for the analysis of fault tolerance using architecture models, the first studying fault propagation mechanisms and the second analysing the availability of services (which means the possibility of achieving the objectives) at the system level.

4.2 Faults in Components and Their Consequences

Having defined availability of services and the key concept in the generic description of components, tools are needed to analyse which conditions could cause a certain version of a service to become unavailable. Faults or partial failure in components would clearly be candidates for a service to become unavailable. This section introduces a Boolean formalism to analyse propagation of faults and the consequences faults can have on the services offered by a generic component.

Shut-down functions and interlocks are commonly used in industrial automation to prevent failures to dilate from one sub-system to another. The use of such functions has, however, the consequence that plant availability is sometimes reduced without good reason. With the ever higher degree of automation, this has been the key cause to increased vulnerability to simple faults, particularly in sensors and actuators. The approach in this text is to obtain dependability by giving a generic component or subsystem an ability to detect and isolate faults and react with actions that accommodate the control system to the fault. Fault accommodation is predetermined at the design stage. The scope of the methods presented in this section is to give a formal technique to obtain a list of which faults should be handled to regain an acceptable version services after faults have occurred locally in a generic component.

Open and closed-loop systems. Handling of faults in open-loop systems, e.g., monitoring and remote control, is technically straight-forward, but the reactions used to accommodate a fault need to be designed with careful consideration to safety and availability of the total plant. Optimisation at a local level may easily violate an overall safety goal.

Handling of faults in closed-loop components is a more difficult and challenging task. Properly designed systems can accommodate the effects of faults whereas less careful designs can let fault effects propagate to other subsystems.

Connection with reliability analysis. For the reasons given above, fault analysis need to incorporate analysis throughout a system. Traditional methods for fault

detection and isolation do not cover this problem. They are very able to detect the presence of a fault as a difference between actual and expected behaviour. Isolation of a particular fault requires a hypothesis about the observed effects from this fault. This is obtained by ad hoc engineering and requires deep process knowledge and engineering skills to make a successful design. It is expensive in terms of both key personnel and time.

Analysis of system reliability is not only mandatory for safety critical systems but is also more and more often used for common industrial systems, driven by the increasing environment and safety awareness in recent years. The state of the art is such that no method can guarantee a complete description of all possible fault modes of a system. Certain forms of risk analysis provide, nevertheless, a very systematic approach to fault modelling once possible component faults have been identified. Faults in common industrial components are subject to constant study, and a methodology based on component fault modelling could use accumulated knowledge for each type of component. The number of principally different components in a certain branch of industry is small enough to make this a manageable exercise.

A systematic approach. A systematic approach can be made if the basic methodology from risk analysis is adopted to the detailed mathematical models needed for real-time fault diagnosis. A link has to be established from quantitative, static risk models at the component level to qualitative, dynamical fault diagnosis descriptions of input–output relations to achieve this goal.

The link is obviously to merge the component-based generic dynamical models (energy, momentum and flow relations) with component fault models from the risk analysis. The generic dynamical models can be extended to subsystem input–output descriptions, for example using a system behaviour description approach. This methodology guarantees that all relevant component faults are included in a mathematical system model, and all relevant dynamical relations are preserved due to modelling being done at the component level.

The systematic approach shall provide the following information:

1. List of faults to detect,
2. Mathematical model for use in fault diagnosis,
3. Basic character/criticality of each fault, and
4. Required reaction to each fault.

This is elaborated in the following.

4.3 Fault Propagation Analysis

Several approaches exist to analyse systems based on the components they comprise. The fields of risk analysis and reliability engineering have developed several approaches to assess the risks associated with component breakdown. On commonly accepted standard in everyday industrial use is the failure modes and effects analysis (FMEA) technique. It is based on description of the failure modes of the individual

components, and would thus serve our purpose. The failure mode and effects analysis technique is well established and supported by both databases with breakdown information and mean-time between failure history for many components. It was hence natural to develop a method for analysis of fault propagation based on such available information on component failure modes.

Failure modes and effects analysis. Failure modes and effects analysis is a tool originally developed by reliability engineers. It analyses potential effects caused by simple or aggregated components ceasing to behave as intended, i.e. they stop providing the service designated to the component. A failure modes and effects analysis procedure starts with listing, for each component, in which ways can this component fail. This is referred to as failure modes. Databases are available with information about failure modes for a large number of industrial components. The output of a failure modes and effects analysis procedure is the effect on the system and its environment that would be the consequence if the particular component should fail in each of the ways available to it (failure effects).

An example for a typical failure modes and effects analysis worksheet is illustrated for a pressure gauge in the following table.

item ident.	*failure mode*	*failure cause*	*failure effect*	*risk assess. sev prob*
press. gauge PG 24	false high reading	defective stuck	toxins not destroyed	4 0.0002
	false low reading	defective stuck	potential burns	3 0.0002

The failure modes and effects analysis worksheet has columns for *item identification*, *failure modes, failure cause, failure effect* and *risk assessment* for the end effects at system/environment level. There are also columns for *risk code* and *actions required*, not shown here.

The information on end effects in a failure modes and effect analysis scheme is firmly linked to the system architecture, which is not explicit in the worksheet. Analysis and design of fault-tolerant systems require a fully flexible representation not offered by the failure modes and effect analysis scheme itself, but the information on component failure modes is very useful and is exploited in the following.

Fault propagation matrix. A traditional failure mode and effects analysis starts with selection of the lowest level of analysis. In the present context, this means sensors, valves, motors and similar components. All potential faults and their effects are determined. A fault propagation scheme for each component shows how fault effects out of the component relate to faults at input, output, or parts within the components. This is illustrated in Fig. 4.2.

Analysis of the propagation of faults is conveniently based on matrix methods. The *fault propagation analysis* (FPA) uses a Boolean mapping of faults onto effects for each component or each set of aggregated components.

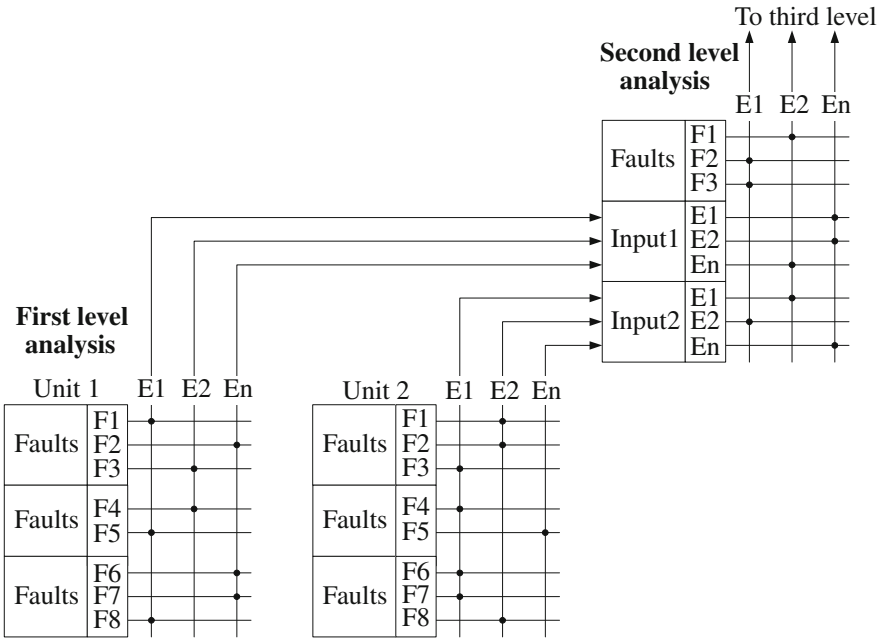

Fig. 4.2 Traditional failure modes and effects analysis scheme illustrated graphically for two component levels

Definition 4.1 (*Fault propagation matrix*) For a given Boolean mapping $\boldsymbol{M}$

$$\boldsymbol{M}: \ \mathcal{F} \times \mathcal{E} \ \rightarrow \ \{0, 1\}$$

of the set of component faults $f_c \in \mathcal{F}$ onto the set of effects $e_c \in \mathcal{E}$, the fault propagation matrix is defined as follows:

$$m_{ij} = \begin{cases} 1 & \text{if} \quad f_{cj} = 1 \Rightarrow e_{ci} = 1 \\ 0 & \text{otherwise.} \end{cases}$$

A fault propagation matrix scheme can be expressed as

$$\boldsymbol{e}_{ci} \leftarrow \boldsymbol{M}_i^f \otimes \boldsymbol{f}_{ci},$$

where $\boldsymbol{M}_i^f$ is a Boolean matrix representing the propagation. The operator $\otimes$ is the inner product disjunction operator that performs the Boolean operation

$$e_{cik} \leftarrow (m_{ik1} \wedge f_{ci1}) \vee (m_{ik2} \wedge f_{ci2}) \ldots \vee (m_{ikn} \wedge f_{cin}).$$

When effects propagate from other components, we get, at level i:

$$\boldsymbol{e}_{ci} \leftarrow \boldsymbol{M}_i^f \otimes \begin{pmatrix} \boldsymbol{f}_{ci} \\ \boldsymbol{e}_{c\,(i-1)} \end{pmatrix}.$$

This is a surjective mapping from faults to effects: there is a unique path from fault to end effect, but several different faults may cause the same end effect.

System descriptions are obtained from interconnection of component descriptions. Merging three levels gives the end effects at the second level,

$$\boldsymbol{e}_{c2} \leftarrow \left(\boldsymbol{M}_2^f \otimes \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ 0 & \boldsymbol{M}_1^f \end{pmatrix} \right) \otimes \begin{pmatrix} \boldsymbol{f}_{c2} \\ \boldsymbol{f}_{c1} \end{pmatrix}.$$

Eventually, end effects at the system level are reached.

Reverse analysis. The effect vector corresponding to a particular fault f_k is hence the kth column of $\boldsymbol{M}^f$. Reversely, given a particular $\boldsymbol{e}$, the set of faults that could cause this effect is obtained by checking which columns of $\boldsymbol{M}^f$ match the observed $\boldsymbol{e}$. This can be written using the operator $\odot$ defined by the operation

$$\begin{aligned} f_i &\leftarrow (m_{i1} = e_1) \wedge (m_{i2} = e_2) \ldots \wedge (m_{in} = e_n) \\ i &= 1, \ldots, \dim f \end{aligned}$$

and apply this on $\left(\boldsymbol{M}^f\right)^{\mathrm{T}}$,

$$\boldsymbol{f}_{\mathrm{c}} = \left(\boldsymbol{M}^f\right)^{\mathrm{T}} \odot \boldsymbol{e}_{\mathrm{c}}.$$

Analysis of the system matrix can easily show where in the system the propagation should be detected and stopped, the operation we would achieve by fault handling. Proper handling of a fault would imply the particular entry(ies) in the $\boldsymbol{M}^f$ matrix change from "1" to "0".

Experience from applying fault propagation analysis to larger systems shows that we might need to include occurrence of one fault and the non-occurrence of another in the description. This would imply to extend $\boldsymbol{f}_i$ to $\left[\boldsymbol{f}_i, \bar{\boldsymbol{f}}_j\right]^{\mathrm{T}}$ in the above expressions.

Analysis of a system with three simple components and a description of their architecture is shown in the following example.

Example 4.2 Propagation with three components
A system with three components and open-loop structure is

$$\begin{aligned} e_{c3} &\leftarrow \boldsymbol{M}_3^f \otimes \begin{pmatrix} f_{c3} \\ e_{c2} \end{pmatrix} \\ e_{c2} &\leftarrow \boldsymbol{M}_2^f \otimes \begin{pmatrix} f_{c2} \\ e_{c1} \end{pmatrix} \\ e_{c1} &\leftarrow \boldsymbol{M}_1^f \otimes (f_{c1}) \end{aligned}$$

The fault effect scheme for this example is

$$
\begin{aligned}
e_{c3} &\leftarrow \boldsymbol{M}_3^f \otimes \begin{pmatrix} f_{c3} \\ e_{c2} \end{pmatrix} \Rightarrow \\
e_{c3} &\leftarrow \left(\boldsymbol{M}_3^f \otimes \begin{pmatrix} \boldsymbol{I} & 0 \\ 0 & \boldsymbol{M}_2^f \end{pmatrix} \right) \otimes \begin{pmatrix} f_{c3} \\ f_{c2} \\ e_{c1} \end{pmatrix} \Rightarrow \\
e_{c3} &\leftarrow \left(\boldsymbol{M}_3^f \otimes \begin{pmatrix} \boldsymbol{I} & 0 \\ 0 & \boldsymbol{M}_2^f \otimes \begin{pmatrix} I & 0 \\ 0 & \boldsymbol{M}_1^f \end{pmatrix} \end{pmatrix} \right) \otimes \begin{pmatrix} f_{c3} \\ f_{c2} \\ f_{c1} \end{pmatrix} \Rightarrow \\
e_{c3} &\leftarrow \boldsymbol{M}_3^f \otimes \begin{pmatrix} \boldsymbol{I} & 0 \\ 0 & \boldsymbol{M}_2^f \end{pmatrix} \otimes \begin{pmatrix} I & 0 \\ 0 & \boldsymbol{M}_1^f \end{pmatrix} \otimes \begin{pmatrix} f_{c3} \\ f_{c2} \\ f_{c1} \end{pmatrix} \equiv \boldsymbol{M}_{sys}^f \otimes f_{sys}
\end{aligned}
$$

Effects are seen to be propagated to the next level of analysis and act as part's faults at that level. This is continued until the system level is reached. The schemes give an surjective mapping from faults to effects: There is a unique path from fault to end effect, but different faults may cause the same end effect. □

It is noted that the Boolean propagation matrix can be split into columns propagating faults and columns propagating input effects,

$$\boldsymbol{M}_i^f = \left(\boldsymbol{M}_{i,f}^f \,|\boldsymbol{M}_{i,e}^f \right)$$

Merging two levels can then be re-written

$$
\begin{aligned}
e_{c2} &\leftarrow \left(\boldsymbol{M}_2^f \otimes \begin{pmatrix} \boldsymbol{I} & 0 \\ 0 & \boldsymbol{M}_1^f \end{pmatrix} \right) \otimes \begin{pmatrix} f_{c2} \\ e_{c1} \end{pmatrix} \Rightarrow \\
e_{c2} &\leftarrow \left(\boldsymbol{M}_{2,f}^f \quad \boldsymbol{M}_{2,e}^f \otimes \boldsymbol{M}_1^f \right) \otimes \begin{pmatrix} f_{c2} \\ f_{c1} \end{pmatrix}
\end{aligned}
$$

This illustrates how faults from the current level are propagated through the component being considered, while faults from a lower level propagate through this lower level and through the present.

Remark 4.1 (*Single-fault assumption*) This discussion of fault propagation is based on the assumption that only a single fault is present. If the analysis should cover the occurrence of multiple faults, or propagation of one particular fault being dependent on a particular other fault not being present, we would need a more complex logic description than introduced above. Results do exist, but are considered outside the scope of the present text. □

Example 4.3 Autopilot - gyro system diagnosis
Failure of ship's motion control systems have been the cause of many severe accidents. Some of these were caused by faults in the gyro-system providing the heading and turn rate motion feedback to the autopilot. Early detection of such faults could prevent the control system from unwanted alteration of the ship's heading. This example illustrates fault detection on a ship's gyro compass and an associated turn rate sensor.

Faults are possible in either of the two measurements and the purpose of the fault detection is to isolate the faulty sensor. Subsequent fault accommodation should then switch the faulty sensor out and estimate the missing signal from that of the good sensor. Fault-tolerance against these faults is obtained by implementing the scheme as an autonomous part of the heading control loop.

FPA scheme for rate gyro. FPA schemes describe the properties of signals or first physical quantities related to the function or output of the component. The effects listed in the first row are quantised descriptions of the properties of the signals. The relationship from input to output is indicated in matrix form in the propagation analysis. The causes due to different effects of the component are listed in the table. These very specific details about component failure are not used in our analysis, but are used as a good starting point for a systematic analysis. FPA schemes are available from several databases of component reliability, in particular from components used in the nuclear, space and avionics industries, where post-failure analysis has been made systematically. The scheme for the rate gyro is illustrated in the following table.

Signal	low	high	fluctuating	undefined
Fault	Electric short Electrical or mechanical defect	Electric short Electrical defect	Wire defect Unit damaged	Wire defect Unit damaged
Input:	low rate	high rate	supply power	dismounted

The FPA matrix for the rate gyro is defined by considering a generic failure of the rate gyro, which can cause any of the output signal conditions: *low, high, fluctuating* or *undefined*.

$$\boldsymbol{e}_{rg} \leftarrow \boldsymbol{M}_{rg}^{f} \otimes \begin{pmatrix} \boldsymbol{f}_{\omega} \\ \boldsymbol{e}_{ship} \end{pmatrix}$$

$$\begin{pmatrix} e_{rg,l} \\ e_{rg,h} \\ e_{rg,f} \\ e_{rg,u} \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} f_{\omega} \\ e_{ship,l} \\ e_{ship,h} \end{pmatrix}$$

Observation of the set of end effects would show which fault(s) could be the cause(s) to a particular end effect combination,

$$\begin{pmatrix} \boldsymbol{f}_{rg} \\ \boldsymbol{e}_{ship} \end{pmatrix} \leftarrow \boldsymbol{M}^{b}_{rg} \odot \boldsymbol{e}_{rg}$$

$$\begin{pmatrix} f_{rg} \\ e_{ship,l} \\ e_{ship,h} \end{pmatrix} \leftarrow \begin{pmatrix} 1\ 1\ 1\ 1 \\ 1\ 0\ 0\ 0 \\ 0\ 1\ 0\ 0 \end{pmatrix} \odot \begin{pmatrix} e_{rg,l} \\ e_{rg,h} \\ e_{rg,f} \\ e_{rg,u} \end{pmatrix}$$

The interpretation is clearly that

$$e_{rg,l} \ \wedge e_{rg,h} \ \wedge e_{rg,f} \ \wedge e_{rg,u} \Rightarrow \ f_{rg}$$
$$e_{rg,l} \ \wedge \neg e_{rg,h} \ \wedge \neg e_{rg,f} \ \wedge \neg e_{rg,u} \Rightarrow \ e_{ship,l} \quad \text{etc.}$$

A complete systematic analysis will show which faults have severe end effects, i.e. effects that could cause the ship to make an unexpected alteration of heading. These include faults on either of the rate or heading gyro units, a fault in the steering gear and a fault in the heading reference to the autopilot. This list of faults is used when modelling the system and the faults to be diagnosed are identified from this list of high severity fault events. The fault propagation analysis uses knowledge of the overall characteristics of the effects of faults. To proceed in further detail with detection, we will later need a model where the specific faults are described as change of the parameters in a generic mathematical model. A generic fault model for the rate gyro is

$$\omega_{3m}(t) = (1 + \alpha_{\omega}(t))\,\omega_3(t) + f_{\omega}(t) + \nu_{\omega}(t),$$

where ω_{3m} is the measured signal, ω_3 the true turn rate. Faults will occur as changes in either of the signals α_{ω} or f_{ω}, both of which are functions of time. The signal $f_{\omega}(t)$ is additive in this model, $\alpha_{\omega}(t)$ is non-additive. Both are zero when no faults are present. The signal $\nu_{\omega}(t)$ represents measurement noise. Note that a non-additive fault can be omitted in the fault model, since a gain fault can be modelled through an additive term as

$$f_{\omega}(t) = \alpha(t)\,\omega_3(t). \quad \square$$

Completeness. Completeness of the fault effect vector is a necessary prerequisite for later fault detection and isolation, since the only faults that can be isolated are those specified in the design. Completeness is obtained if all possible component faults are considered. This is not achievable in a rigorous sense, but engineering experience from risk analysis makes it possible for practical purposes.

It is noted that completeness does not ensure that component fault isolation is possible because the mapping from fault to effects is not an isomorphism (one-to-one mapping): An observed effect could be caused by any out of several component faults.

Definition of generic components. The above introduction of fault propagation matrix to characterise propagation of fault through components leads to include the fault propagation matrix in the formal definition of a generic component

Definition 4.2 (*Generic component model* (*extended*)) A system component is defined by the model given in Definition 4.2 together with the additional model part:

$$
\begin{aligned}
&< \text{FPA input} \mid \text{use-mode} > ::= \{< \text{list of internal faults;} \\
&\qquad\qquad\qquad\qquad\qquad \text{list of input effects} >\mid \text{use-mode}\} \\
&< \text{FPA output} \mid \text{use-mode} > ::= \{< \text{list of output effects} >\mid \text{use-mode}\} \\
&< \text{FPA description} \mid \text{use-mode} > ::= \{< \text{FPA input; FPA output;} \\
&\qquad\qquad\qquad\qquad\qquad \text{FPA matrix;} > \mid \text{use-mode}\}.
\end{aligned}
$$

Example 4.4 Temperature control
This example illustrates the aggregation of simple components into a complex component that provides a temperature control service. The problem considered is to accommodate some of the faults that would stop the primary service of the temperature control loop in Fig. 4.3. A three-way valve controls the mixing of hot water returning from a tank with tempered water from a heat exchanger. The control objective is to keep the cooling water temperature of an exogenous process in the tank at a constant value. The valve is controlled by the temperature control loop, which consists of

- the actuator with AC motor,
- the temperature sensor (T),
- the controller with process interface (AI, AO, A/D, D/A), and
- the filter system.

The control loop is shown in Fig. 4.3.

The temperature control loop is a cascade control with position control of the valve as the inner loop. Stability of the total loop is not guaranteed if the inner loop becomes open due to a component fault.

Three-way valve actuator. The valve is driven by an AC motor that will rotate left or right, when activated to either side by a double acting relay. End-stop switches are supposed to avoid motor overload by preventing the motor from turning further when an end stop switch is reached. The potentiometer gives position feedback. The position loop fails if either potentiometer or end-stop switches fail.

Figure 4.4 shows the graphical representation of the FPA scheme for the closed-loop valve position controller. Bold lines in the scheme show how faults propagate. The important observation is that propagation could be stopped at the points marked with stars. This means that fault handling should be applied exactly at these points.

It is intuitive that accommodation of a position or limit switch fault could be done fairly simple:

1. Use an estimate of the valve position in the motor controller instead of a faulty position signal.
2. Override a limit switch information if both position sensor feedback and an estimated position show that a limit switch fault has occurred.

An observer for this purpose is elementary. The estimated valve position is increased or decreased in proportion to the time either of the two motor relays. This requires no additional hardware but a few lines of observer code. Accommodation of any of these sensor faults will make it possible to continue operation while giving an alert about required maintenance. Without accommodation, the temperature control loop would probably fail due to the loop becoming unstable without the internal position feedback.

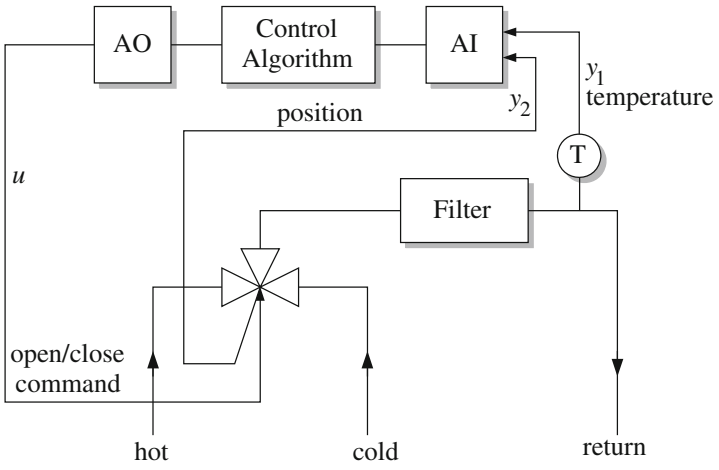

Fig. 4.3 Piping and instrumentation diagram representation of a temperature control loop with three-way valve

Figure 4.4 shows the FPA scheme in block diagram form for the valve control part of the loop. The components are potentiometer, limit switches, motor, three-way valve and digital controller.

Faults in a limit switch will prevent motion in clockwise or counter-clockwise direction—opening or closing of the valve. The consequence is a severe offset of the temperature control if fault handling is not initiated. A breakdown of the position feedback element will cause a breakdown of the temperature control loop because the motor will be driven rapidly to fully open or fully closed position.

Because several faults can cause the same effect, it is necessary to isolate the failure source. When the source is isolated it is possible to decide the reaction:

- **Actuator fault**. Fault in the valve up–down relay switch or in the ac-motor: The position controller must stop immediately. This will cause a loss of the temperature control service.
- **Actuator fault**. Fault in the valve end-stop switch: The position controller switches to use the position sensor and up–down commands for estimation of position. The service continues until maintenance.
- **Position sensor fault**. The controller should be re-configured. In the analytical relation between duration of relay pulses and motor shaft position, a position estimate is readily available. The estimate is used until the fault is repaired.
- **Temperature sensor fault**. The reference to the position controller fails. The controller is re-configured and a time-history roll-back is made of the reference signal and the mean is used as the new reference until the fault has been repaired.

This examples illustrate situations where temperature would deviate significantly or the control would simply fail with a commonly applied controller design. The temperature control service would no longer be available, and the overall use-mode of the tank would need to be changed to emptying, for safety reasons. By contrast, fault accommodation could assure availability of a reduced temperature control service, for several likely faults, thus enhancing the overall plant availability with simple means. □

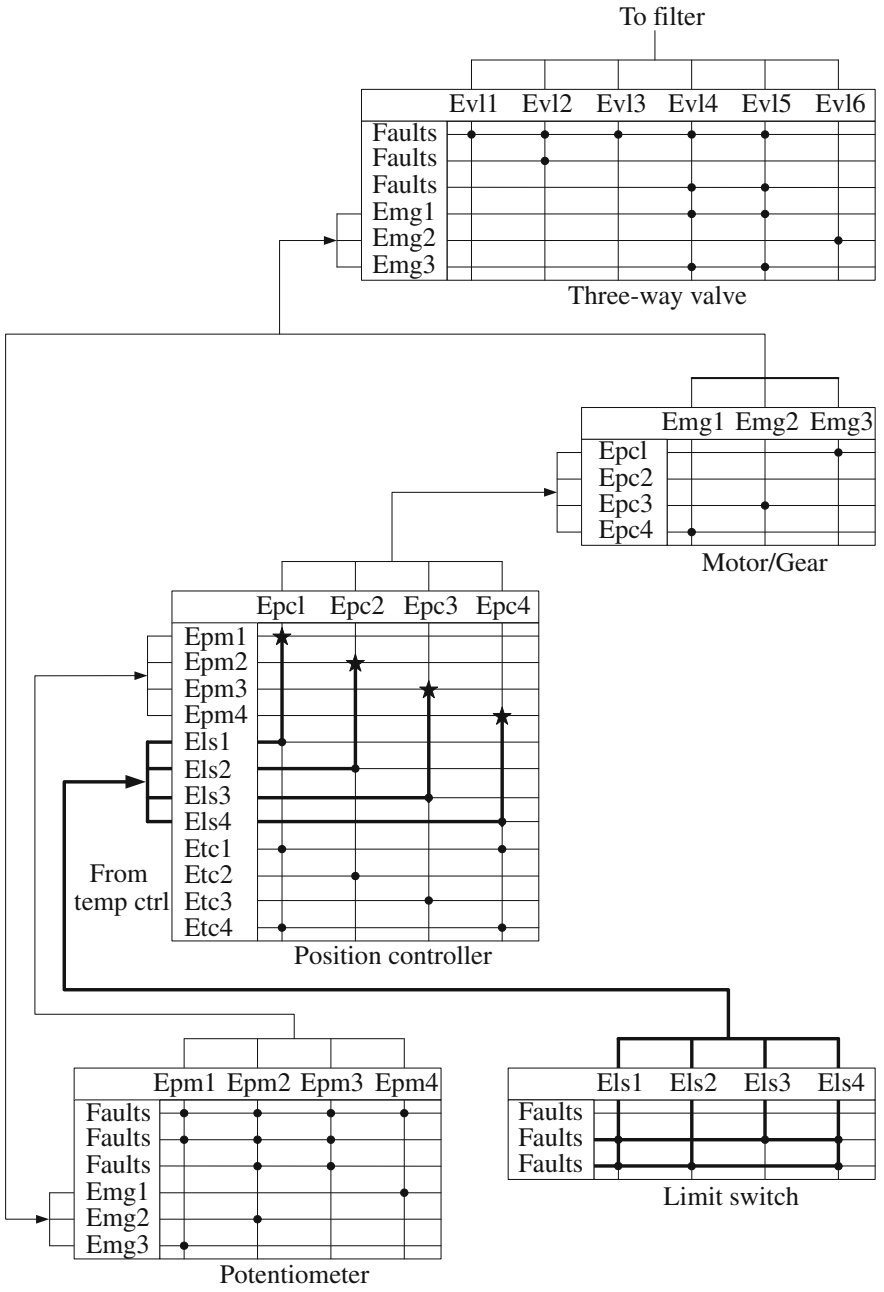

Fig. 4.4 Propagation of fault effects in closed-loop control of three-way valve. *Solid lines* show fault propagation, points marked with *star* show where propagation can be stopped

4.4 Graph Representation of Component Architecture

The above discussion has shown that a block diagram for the FPA analysis of an aggregated component consists of

- the external input faults or effects propagated to the component,
- the FPA representation of lower level components within the aggregated component, and
- the end effects for the aggregated component.

The latter can be considered output of the FPA analysis. The task of dealing with closed-loops in the FPA diagram can be eased by employing a graph formulation. An appropriate FPA graph is first defined. It is then shown that how closed loops are identified and finally how cut sets can be obtained.

Definition 4.3 (*Fault propagation analysis graph*) Let a system be comprised the following items: input effects ι, components with FPA blocks γ and output effects ζ. Define a set of vertices as $V = \{\iota, \gamma, \zeta\}$ of an FPA graph. Connections between the system items are edges of the graph. The edges constitute the set E. The FPA graph Γ is an ordered pair of disjoint sets (V, E). $V = V(\Gamma)$ is the set of vertices and $E = E(\Gamma)$ the edge set.

We further define an orientation of the edges in the FPA graph.

Definition 4.4 (*Orientation*) An edge (i, j) is said to connect a vertice j to i. If an edge is oriented and connects vertice j with i, then $e_{ij} = 1$.

This leads to a matrix representation of the FPA graph with oriented edges.

Definition 4.5 (*Directed adjacency matrix*) The directed adjacency matrix $\boldsymbol{D}$ of Γ, with respect to a given orientation of Γ, is the $n \times n$ matrix (d_{ij}) whose entries are

$$d_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is the positive end of an edge from } v_j \\ 0 & \text{otherwise,} \end{cases}$$

where the number of vertices in the graph is n.

The directed adjacency matrix is thus square. The entries of the ith row show which connections point to the ith item (input, component, or output) in the fault propagation diagram. The cardinality of "1" entries in the directed adjacency matrix is equal to the number of edges in the graph.

Remark 4.2 (*Input vertex*) The ith vertex is an input vertex if and only if the ith row in the adjacency matrix comprises zeroes only. □

Remark 4.3 (*Output vertex*) The jth vertex is an output vertex if and only if the jth column in the adjacency matrix comprises zeroes only. □

Definition 4.6 (*Walk of length k*) A walk of length k in Γ is a finite sequence of vertices in the graph Γ $\{v_0, v_1, \ldots, v_k\}$ such that v_{t-1} and v_t are adjacent for $1 \leq t \leq k$.

Graph theory is very useful in respect to showing some general properties of the graph Γ.

Lemma 4.1 (Biggs) *The number of walks of length k in* Γ *from* v_i *to* v_j *is the entry in position* (i, j) *of the matrix* $\boldsymbol{D}^k$.

A closed loop is a walk that leads from an item and back to itself. Hence, a closed loop of length k will appear as a 1 in the diagonal of $\boldsymbol{D}^k$ for each vertex that is part of the loop. This gives an algorithm to find closed loops in a fault propagation graph.

Theorem 4.1 (Loops of length k in a fault propagation graph) *A graph with a vertex* v_i *has exactly one walk back to itself of length k if and only if the ith diagonal entry of* $\boldsymbol{D}^k$ *is 1. Each vertex in the loop has its diagonal entry equal to 1.*

A vertex v_i *that participates in n closed loops will have a diagonal entry in* $\boldsymbol{D}^k$ *equal to n. The number n will include possible multiple rounds in a loop if its length is an integer fraction of k.*

Diagonal elements of a vertice hence show how many closed loops of length k or k/M the vertice is part of, where M is an integer number. The power k of D used in the calculation shall not exceed the number of vertices in the graph.

Example 4.5 Closed loops in a fault propagation graph
A fault propagation graph is illustrated in Fig. 4.5.

The directed adjacency matrix is D in Eq. (4.1). Powers of the adjacency matrix are

$$\boldsymbol{D} = \begin{pmatrix} 0&0&0&0&0&0 \\ 1&0&0&1&0&1 \\ 0&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&1&0&0 \end{pmatrix}, \quad \boldsymbol{D}^2 = \begin{pmatrix} 0&0&0&0&0&0 \\ 0&1&1&1&0&0 \\ 1&0&0&1&0&1 \\ 1&1&0&1&0&1 \\ 0&1&1&0&0&0 \\ 0&1&1&0&0&0 \end{pmatrix} \tag{4.1}$$

$$\boldsymbol{D}^3 = \begin{pmatrix} 0&0&0&0&0&0 \\ 1&2&1&1&0&1 \\ 0&1&1&1&0&0 \\ 1&1&1&2&0&1 \\ 1&1&0&1&0&1 \\ 1&1&0&1&0&1 \end{pmatrix}, \quad \boldsymbol{D}^4 = \begin{pmatrix} 0&0&0&0&0&0 \\ 2&2&1&3&0&2 \\ 1&2&1&1&0&1 \\ 1&3&2&2&0&1 \\ 1&1&1&2&0&1 \\ 1&1&1&2&0&1 \end{pmatrix}$$

$$\boldsymbol{D}^5 = \begin{pmatrix} 0&0&0&0&0&0 \\ 2&4&3&4&0&2 \\ 2&2&1&3&0&2 \\ 3&4&2&4&0&3 \\ 1&3&2&2&0&1 \\ 1&3&2&2&0&1 \end{pmatrix}, \quad \boldsymbol{D}^6 = \begin{pmatrix} 0&0&0&0&0&0 \\ 4&7&4&6&0&4 \\ 2&4&3&4&0&2 \\ 4&6&4&7&0&4 \\ 3&4&2&4&0&3 \\ 3&4&2&4&0&3 \end{pmatrix}.$$

The diagonal of the $\boldsymbol{D}^k$ matrix shows the special characteristics:

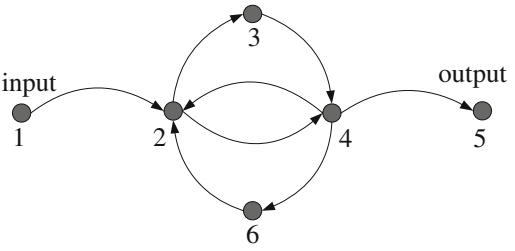

Fig. 4.5 A fault propagation graph example. One vertice is input (*1*), another is output (*5*)

- $\boldsymbol{D}^2$: 1 loop of length 2. It is {(2,4)}.
- $\boldsymbol{D}^3$: 2 loops through 2 and 4, one through 3 and 6. They are {(2,3,4), (2,6,4) }.
- $\boldsymbol{D}^4$: 2 loops through 2 and 4, one through 3 and 6. They are {(4,2,4,2), (3,4,6,2)}.
- $\boldsymbol{D}^5$: 4 loops through 2 and 4, one through 3 and 6. They are {(2,3,4,2,4), (6,2,4,2,4), (2,4,6,2,4), (2,4,2,3,4)}.

Element (5,1) in $\boldsymbol{D}^k$ shows for which k there is a connection from input to output. The shortest walk from input (1) to output (5) has length 3, as seen from element (5,1) in $\boldsymbol{D}^3$.

It should be noted that multiple rounds in loops are indeed part of the loop count for a vertex as seen in the diagonal entry of the $\boldsymbol{D}^k$ matrix. □

4.5 Fault Propagation with a Closed Loop

The failure mode and effects analysis scheme for a set of components connected in a closed logical loop is principally described as

$$\boldsymbol{e}_{ci} \leftarrow (\boldsymbol{M}_i^f \quad \boldsymbol{I}) \otimes \begin{pmatrix} \boldsymbol{f}_{ci} \\ \boldsymbol{e}_{ci} \end{pmatrix}.$$

Looking at the logic operation of this equation, the solution is

$$\boldsymbol{e}_{ci}^{+} \leftarrow \begin{cases} \boldsymbol{M}_i^f \otimes (f_{ci}) & \text{if and only if } \ \boldsymbol{e}_{ci}^{-} = \text{“0”} \\ \text{“}\mathbf{1}\text{”} & \text{if and only if } \ \boldsymbol{e}_{ci}^{-} = \text{“}\mathbf{1}\text{”}, \end{cases}$$

where $\boldsymbol{e}_{c1}^{-}$ is the state prior to the calculation, $\boldsymbol{e}_{ci}^{+}$ is the state resulting from the calculation. It is seen that once triggered, the effect $\boldsymbol{e}_{ci}$ remains permanently true, also after the fault disappears. This mechanism is a penalty of the Boolean representation of faults and their propagation, and the price for this is to get a fast tool for a first analysis of fault propagation.

When a closed logical loop is present, we hence need to cut an appropriate connection within the loop and investigate whether a “true” signal into the broken connection will produce a “true” at the other end of the cut. If this is the case, the loop is a tautology, a formula that is true in every interpretation, which can be eliminated. If the

"true" in produces a "false" at the other end of the cut, the logical loop is a contradiction and no solution exists. We then need to define the input of the cut as a new input in our failure mode and effects analysis description of the system, and analyse the propagation of an imagined "fault" condition = "true" from this point.

The non-existence of a logical model for a closed loop is not equivalent to instability in the continuous model representation. The stability of a closed-loop system cannot be determined from the properties of an over-simplified logical model of fault propagation.

In FMEA analysis, the closed-loop problem is artificially circumvented by treating the closed loop as a unit without feedback. The FMEA approach is hence to ignore the feedback loop as such and consider the closed-loop operation of the component as the functionality of the component and its' closed-loop considered as an extended component. Failure within the feedback loop itself is then treated by modelling this event as a separate fault. In essence, this is exactly what is done in this Boolean fault propagation analysis approach that was presented here. When we meet a closed loop, analysis can be achieved by extending the system with auxiliary faults. This technique is illustrated below.

4.5.1 Cutting the Closed Fault Propagation Loop

The existence and location of closed logical loops can be determined quite easily with the graph-theory-based tool we presented in Sect. 4.5. The directed adjacency matrix $\boldsymbol{D}$ and powers of $\boldsymbol{D}$ up to degree k showed which vertices of the FPA graph are parts of closed loops, if any, and it informed on the paths from input to output.

When a logical loop cut has to be made, it should be made such that the path from input to output is not interrupted, while cutting the relevant loop(s). The cut is conducted by cutting an edge, defining a new input and output as needed. The variables associated with the extra (new) input and output vertices are given by the variables associated with the edge that was cut. Logic analysis of the system is carried out using the new input as additional faults. The new output is observed. If the variable (effects) at the output is identical with the input, the relation is a tautology and can be removed from the analysis. If the result is a logic contradiction, the new input needs to remain in the analysis, and the logical loop remains open.

In conclusion, the representation of a fault effect as a Boolean signal {0, 1} and the fact that Boolean algebra prohibits dealing with closed-loop logic, unless the logic signals are clocked and therefore delayed as in flip-flop circuits, is a serious obstacle to a pure Boolean analysis of fault propagation.

Example 4.6 Ship track control - fault propagation
The propagation of faults from the track error sensor to the track controller are investigated in this example.

Track error sensor. An analysis of the track error sensor leads to the following internal failure modes

- $f_{\text{tes,hardware}}$: hardware fault causing abrupt fault (no output signal)

The input effects are effects from other components or external faults that are propagated to the component. Here effects from faults in the GPS receiver are considered:

- $e_{\text{gps,o}}$: GPS signal offset
- $e_{\text{gps,u}}$: GPS signal unavailable.

The output from the track error sensor is the track error signal. The effects are track_error low $e_{\text{te,l}}$, track_error high $e_{\text{te,h}}$ and track_error unavailable $e_{\text{te,u}}$.

$$\begin{pmatrix} e_{\text{te,l}} \\ e_{\text{te,h}} \\ e_{\text{te,u}} \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} f_{\text{tes,hardware}} \\ e_{\text{gps,o}} \\ e_{\text{gps,u}} \end{pmatrix}$$

Track controller. One internal fault in the track controller is considered in this example:

- $f_{\text{tc,software}}$: Software fault causing constant heading demand signal as output

$$\begin{pmatrix} e_{\psi_{\text{dem,l}}} \\ e_{\psi_{\text{dem,h}}} \\ e_{\psi_{\text{dem,f}}} \\ e_{\psi_{\text{dem,u}}} \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} f_{\text{tc,software}} \\ e_{\text{te,l}} \\ e_{\text{te,h}} \\ e_{\text{te,u}} \end{pmatrix}$$

Combining the Boolean propagation matrices for the track error sensor and the steering controller leads to a description of the propagation of the combined fault vectors for the two components to the output of the steering controller (Fig. 4.6).

$$\begin{aligned} \boldsymbol{M}^{f}_{\text{tes}\rightarrow\text{tc}} &= \left(\boldsymbol{M}^{f}_{\text{tc,f}} \;\; \boldsymbol{M}_{\text{tc,e}} \otimes \boldsymbol{M}^{f}_{\text{tes}} \right) \\ &= \left(\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

The resulting fault propagation is then

$$\begin{pmatrix} e_{\psi_{\text{dem,l}}} \\ e_{\psi_{\text{dem,h}}} \\ e_{\psi_{\text{dem,f}}} \\ e_{\psi_{\text{dem,u}}} \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \otimes \left(\begin{pmatrix} f_{\text{tc,software}} \\ f_{\text{tes,hardware}} \\ e_{\text{gps,o}} \\ e_{\text{gps,u}} \end{pmatrix} \right). \qquad \square$$

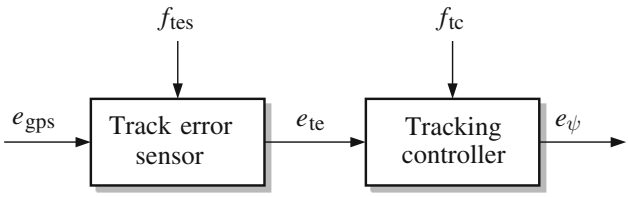

Fig. 4.6 Propagation of faults through the track error sensor and track controller

4.5.2 Assessment of the Severity of the Fault Effects

The consequences of a fault are judged from the implications the end effects could have on safety, on availability of the plant, on environment etc. A judgement of severity should be made of the possible combination of end effects, considering end effects that can arise from any single fault. This has as prerequisite that a single-fault assumption is sufficient for the analysis.

If a double fault is considered, the underlying logic description of fault propagation matrices must support such analysis.

4.5.3 Decision About Fault Handling

The implication is that an automated analysis will need to consider closed loops as special cases. The interpretation of a closed loop in an FPA scheme is merely the observation that closed-loop operation may amplify or attenuate the effects of a fault. Which of the two happens depends on the dynamical properties of the control loop and this question is outside the scope of the FPA analysis.

The component-based analysis can thus provide both a list of fault effects and a suggestion of where in a system fault propagation can be stopped. In the design method, it is then up to the designer to evaluate the severity of each fault effect and determine which fault accommodation actions shall be implemented.

The question how to handle faults will be discussed in Chap. 7.

4.6 Generic Component Models

Generic component models describe the system architecture from a formal point of view, so as to perform systematic manipulations for the purpose of fault diagnosis and fault-tolerant-control design. Contrary to box models, which carry no information about the component behaviour in different operating situations (normal, faulty, different modes), generic component models describe components, which offer services according to the current use-mode. The user may be a human operator (who

directly accesses the component through some man–machine interface) or another component, which accesses the services either through direct or remote connection (as in distributed systems in which services are requested via a local area network).

A generic model of a component describes its operational mode through the services it provides.

4.6.1 Services

From the user viewpoint, a system component provides one or several services. For example, a level sensor provides a signal which is a one-to-one correspondence to the level in the tank. However, the signal may be validated or not, it may be filtered or not, the sensor might memorise the minimal (the maximal) level value encountered on a given time window, it might provide an alarm if the level exceeds a given threshold, etc. All these are examples of services the sensor might provide in the normal operating mode. Other services could be provided in the installation, initialisation, degraded operation, maintenance modes.

Input, output and procedures. A service can be described by input variables, output variables, and some procedure which transforms the former into the latter. For example, a tank consumes input and output mass flows, and produces a stored mass, using an integration procedure (note that the output flow is indeed an input variable for the integration procedure), thus providing an *integration* service whose behavioural model is

$$\dot{h}(t) = q_i(t) - q_o(t),$$

where q_i is mass flow in q_o is mass flow out and $\dot{h}$ is mass increase rate. The *measurement* service of a sensor consumes energy from the outside world and produces a signal which is the image of the measured variable, by means of the transducer. The *controller* service of a controller consumes signals from sensors and produces signals to actuators. It also consumes data (the set-point) that have been previously written in the data base (using the *write* service).

Requests and enabling conditions. Services may be provided unconditionally or on specific request. For example, the *integration* service is systematically provided by a tank (no special request is necessary), at any time and whatever the values of the input and output as long as the tank level is within its rated capacity (no activation condition is needed). A sensor connected on some input port of a microprocessor system would provide the *measurement* service on a *read* request, which would be associated with some specific clock signal (the activation condition) issued to the analog to digital converter. A new set-point would be entered in the regulator memory on a specific *write* request from the human operator (again the request would be associated with an activation condition). The distinction between a request and an activation condition is that the request for a service is issued by the user, while the activation condition is processed by the component.

Resources. The normal running of a service needs some hardware resources. The tank is obviously necessary for the *integration* service to be performed. The transducer, filter, analog to digital converter, power supply and amplifier are necessary for the sensor to perform the *measurement* service. The microprocessor system is needed by the controller to provide the *regulation* service.

Summarising the preceding description under a formal model, a service is a 6-tuple:

<consumed variables, produced variables, procedure, request, activation condition, resources>.

As a consequence, a component is viewed as the set of services it can provide to the users, thus leading to the component model

$$S(k) = \{s_i(k), i \in I_s(k)\} \tag{4.2}$$

$$s_i(k) = \left\{cons_i(k),\ prod_i(k),\ proc_i(k),\ rqst_i(k),\ active_i(k),\ res_i(k)\right\}, \tag{4.3}$$

where $S(k)$ is the set of services of component k, I_s is the set of the indices of the possible services, and the other notations are straightforward.

Modes of operation. Obviously, not all the services provided by a component can be requested at any time during the system's life. A system generally goes through different operating modes, each with its set of prerequisites to function. For example, a request for the level control service from the controller of the single-tank system is denied: when the set-point has not been written (this calls for some initialisation mode); when the tank is empty (during a no-operation mode); when it is emptying during a cleaning mode.

For that reason, definition (4.2) is further extended, by adding some organising structure on the set of services. Normal operating modes are called *use-modes*. They provide the formal model of the structure of the set of services.

Definition 4.7 (*Use-mode* (*UM*)) A use-mode is a subset of services of a component. The set of use-modes covers the set of services, i.e. each service belongs at least to one use-mode, and each use-mode contains at least one service.

Let $M(k) = \{m_i(k), i \in I_m(k)\}$ be the set of use-modes of component k and $S_i(k) \subseteq S(k)$ be the services available in mode $m_i(k)$. Note that the formal definition of a use-mode does not tell which subsets of services have to be selected to form consistent use-modes. This matter is left to the design engineer, who indeed must group into use-modes subsets of services which are consistent in some given operation frame. In the single-tank system, the six possible use-modes are *Filling the tank, Processing batch, No operation* etc.

4.6.2 Introduction of the Generic Component Model

The consequence of the use-mode definition is that the component model must now include a (higher) level description, which models the component possible transitions from one use-mode to another one, and the conditions under which these transitions take place. Indeed, at any time t, the component is in one and only one use-mode, a discrete-event system behaviour which is easily modelled using a deterministic automaton (see Sect. 3.4. for an extensive presentation of discrete-event models). Note also that in order to obtain transitions between use-modes, it is necessary to add new services to $S(k)$. In the controller example, three possible use-modes and the corresponding list of services could be the following:

$$\begin{aligned} \textit{No-operation} &: m_1 = \{\textit{set\_mode\_}m_2,\ \textit{set\_mode\_}m_3\} \\ \textit{Initialise} &: m_2 = \{\textit{enter\_set-point},\ \textit{display\_set-point}, \\ &\qquad \textit{set\_mode\_}m_1,\ \textit{set\_mode\_}m_3\} \\ \textit{In-control} &: m_3 = \{\textit{read\_set-point},\ \textit{calculate\_control\_signal}, \\ &\qquad \textit{set\_mode\_}m_1\}. \end{aligned}$$

Taking into account the services and their organisation into use-modes, the generic model of a component is now defined.

Definition 4.8 (*Generic component model*) A system component is defined by the following formal model[1]:

$$\begin{aligned} &< \text{component k} > ::= < \text{state transition graph } G(M(k), \tau(k), m^0(k)) > \\ &\quad < M(k) > ::= < \text{set of use-modes } \{m_i(k), i \in I_m(k)\} > \\ &\quad < \tau(k) > ::= < \text{set of transitions } \left\{\tau_{ij}(k),\ i, j \in I_m(k)\right\} > \\ &\quad < m^0(k) > ::= < \text{initial use-mode} > \\ &< \text{use-mode } m_i(k) > ::= < \text{set of services } S_i(k) \subseteq S(k) > \\ &\quad < \text{service } s_1(k) > ::= < \text{pre-ordered versions} \\ &\qquad\qquad \left\{s_1^j(k), \quad j \in J(s_1(k))\right\} > \\ &\quad < \text{version } s_1^j(k) > ::= < \text{consumed vars } cons_1^j(k), \\ &\qquad\qquad \text{produced vars } prod_1(k), \\ &\qquad\qquad \text{procedures } proc_1^j(k),\ \text{request } rqst_1(k), \\ &\qquad\qquad \text{activation cond. } activ_1^j(k), \\ &\qquad\qquad \text{hardware and software resources } res_1^j(k) > \\ &< \text{transition } \tau_{ij}(k) > ::= < \text{condition } c_{ij}(k),\ \text{origin } m_i(k), \\ &\qquad\qquad \text{destination } m_j(k) > . \end{aligned}$$

This definition will be extended later.

[1]The notion of versions has been introduced in Sect. 3.2 and will be elaborated in more detail in Sect. 4.6.4.

4.6.3 Simple Components

A simple component is described by the services offered and its use-mode automaton. A component is considered simple when it has no internal means to change the services it provides. Simple components are typically without build-in computational means. Two examples of an actuator and a sensor are treated below.

Actuator for flow control. Flow control is the most widespread actuator function in machinery systems. It is used where a shut-off of a pipe connection is needed, where the flow of a medium is to be controlled, and where control loops manipulate a flow of a liquid in order to change a temperature. Flow control can be open/closed, variable throughput, or redirection of flow from one pipe into two (three-way valves).

The actuator system consists of a three-way valve. It has a common port and two other ports, referred to as A and B. The rotor position determines the opening area between the common port and ports A and B. The valve distributes the flow between ports A and B. The distribution is controlled by the rotor angle. An electro-mechanical device is attached to the valve to control the rotor position.

The electro-mechanical valve actuator consists of a motor and a gear. The rotor position is changed by running the motor in clockwise or counter-clockwise directions. The motor can be in one of the following states: stopped, rotation clockwise, or rotation counter-clockwise.

The state is controlled by activation of two relay contacts. They are denoted "open" and "close", respectively. A potentiometer is used to measure the actual rotor position. Figure 4.7 shows the principle in the actuator operation and electrical connections.

Limit switches on the rotor provide indication of rotor end positions and provide overload protection by preventing the motor to turn further in the direction of which the limit switch has been activated.

The service provided by the flow control valve is described by the six-tuple:

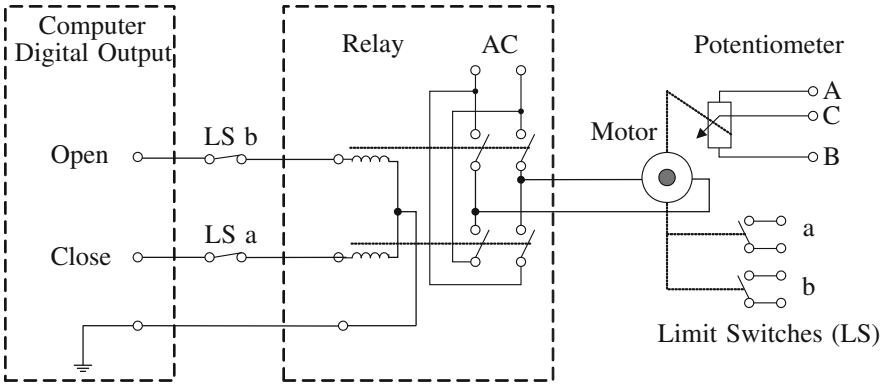

Fig. 4.7 Operation of three-way valve actuator with relay-operated induction motor. (Abbreviations: *o* open, *c* close, *LS* Limit switch, *AC* Alternating Current)

$$
\begin{aligned}
&< \textit{consumed variables} > ::=< \textit{open},\ \textit{close},\ \textit{in\_limit} >,\\
&< \textit{produced variables} > ::=< \textit{angle of output shaft},\ \textit{measured angle},\\
&\qquad\qquad \textit{in\_limit} >,\\
&< \textit{procedure} > ::=< \textit{angle} = \int \textit{up} \cdot \mathrm{d}t - \int \textit{down} \cdot \mathrm{d}t >,\\
&< \textit{request} > ::=< \textit{none} >,\\
&< \textit{activation condition} > ::=< 220\,\mathrm{V}\ \textit{present} >,\\
&< \textit{resources} > ::=< \textit{pot. meter},\ \textit{limit switches},\ \textit{controller},\\
&\qquad\qquad \textit{geared motor} > .
\end{aligned}
$$

The produced variable is the angle of output shaft, which influences the flow. Various faults can cause the flow to differ from the expected or desired value. The following table summarises various faults that can cause such a deviation.

Component/ Effect	Flow too low	Flow not related to control angle	Fluctuating flow	Flow too high output
Fault	Pipe broken, setpoint low, power low	Pipe clogged, pipe leak	Setpoint fluctuating	Too high input flow, setpoint high
	Pipe clogged	Pipe A or B broken, clogged or leak		
	Damage, wear	Hysteresis	Damage, wear	

Potentiometer. A potentiometer changes the position of contact between a resistance element and a wiper when the turning angle is changed. The potentiometer can be considered a voltage divider with a division ratio that is a function of the turning angle. Figure 4.8 shows the typical connection diagram.

Component/ Effect	Signal too low	Not related to angle	Fluctuating signal	Signal too high
Fault	Broken wire at A, short at A-B	Loss of supply	Vibration	Broken wire at A, short-circuit A-C
	Short B-C	Broken wire at C	loose conection	
	Stuck, shaft or element broken	Wiper fault		

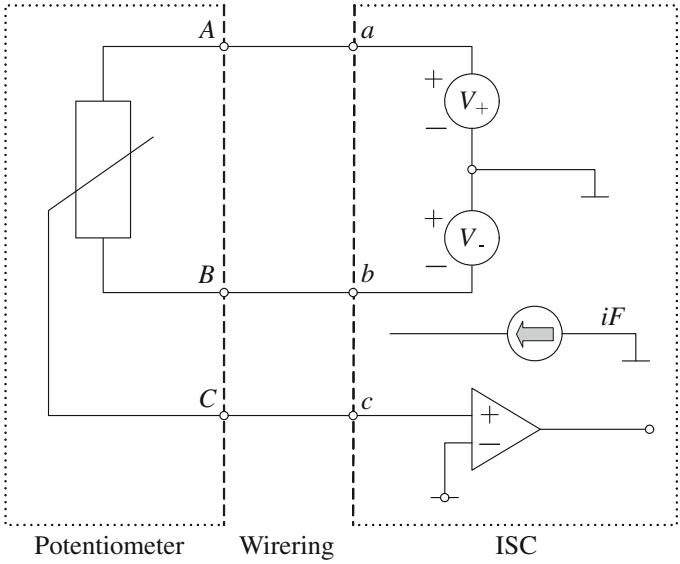

Fig. 4.8 Electrical diagram of potentiometer and computer interface to enable fault detection at the single sensor level

The service provided by the potentiometer as a sensor is an electrical signal proportional to the physical angle of rotation. Several faults will cause loss of this service. Short-circuit of any terminal to supply or to ground, or arbitrary wire disconnection are common events to cause component faults.

4.6.4 Complex Components

From a formal point of view, a component is a set of services. The consideration of aggregated, complex components leads to extend this description to the consideration of fault-tolerance capabilities.

Versions of services. Let $s_i(k)$ be a service provided by component k, and suppose that embedded fault tolerance is available. This means that the service $s_i(k)$ would not be interrupted even if the resources it needs were no longer available. This is only possible if it exists within component k under several versions, namely $s_i(k) = \left\{ s_i^j(k),\ j \in J(s_i(k)) \right\}$ where $s_i^j(k)$ is the jth version of service $s_i(k)$.

From this extension, definition (4.2) can now be stated as:

$$S(k) = \{ s_i(k),\ i \in I_s(k) \}$$
$$s_i(k) = \left\{ s_i^j(k),\ j \in J(s_i(k)) \right\},$$

where each version $s_i^j(k)$ of service $s_i(k)$ is a 6-tuple like (4.3), which can be used indifferently for the same purpose.

Versions pre-ordering. Designing the activation conditions of service versions rests on the definition of a pre-ordering. The versions of a service are separated into classes such that a service in class l is preferred to a service in class m if and only if $l < m$. For example, some versions might be more precise, faster, less consuming, etc. than others.

The design of the activation conditions is then straightforward: at time t when the request for service $s_i(k)$ is issued (or at any time if no request is necessary), the version which is to be run is the lowest rank one whose resources are all non-faulty. Two services in the same class are not ranked, which means that the choice is indifferent. Thus the activation conditions of each version can be chosen arbitrarily, provided they are mutually exclusive. Two examples of commonly used strategies are as follows:

1. Give an arbitrary preference order. Use version 1 as long as its resources are not faulty. Move to version 2 should version 1 fail.
2. Use the preceding scheme but regularly change the service ranking (e.g. in a circular way) so as to distribute the operation equally over time of the different versions.

A simple example of versions ranking is given by the *measurement* service of a sensor which includes two redundant transducers to measure the same variable. Let

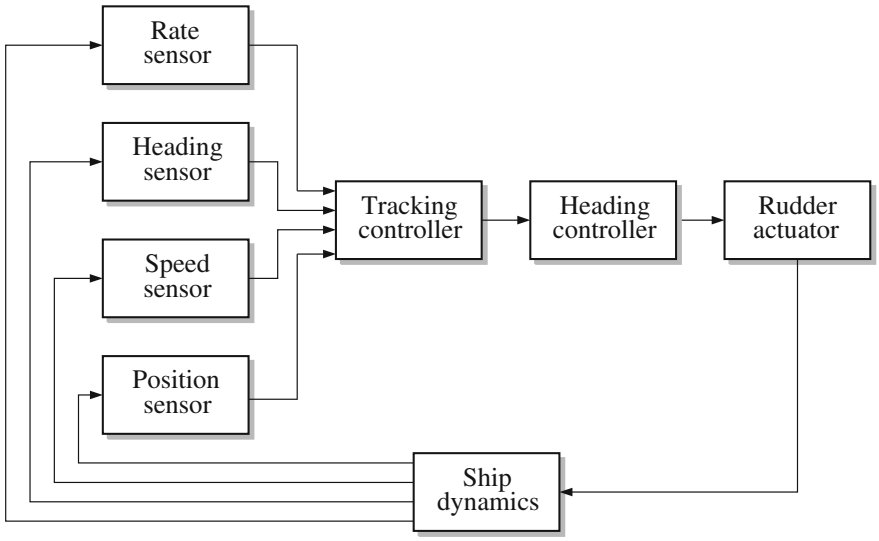

Fig. 4.9 Fault propagation in the ship steering problem

$$y_1(t) = x(t) + \varepsilon_1(t), \quad \varepsilon(t) \sim N(0,\ \sigma_1)$$
$$y_2(t) = x(t) + \varepsilon_2(t), \quad \varepsilon(t) \sim N(0,\ \sigma_2)$$

be the two measurement equations, with $\sigma_2 > \sigma_1$. The following table gives the different versions of the *measurement* service which are provided by this (intelligent) sensor, along with their ranking.

Class	Procedure	Fault situation
0	$y(t) = \frac{1}{\sigma_1+\sigma_2}[\sigma_2 y_1(t) + \sigma_1 y_2(t)]$	No fault
1	$y(t) = y_1(t)$	Transducer 2 faulty
2	$y(t) = y_2(t)$	Transducer 1 faulty

Example 4.7 Component analysis of ship track control
To illustrate the component analysis, it will be applied to the ship steering example introduced in Sect. 2.3 (Fig. 4.9). The subcomponents of the ship steering controller are as follows:

- Rate sensor (rate gyro),
- Heading sensor assembly (gyro compass),
- Track error sensor (Navigation computer with GPS input),
- Speed sensor (ship's log),
- Track control algorithm (software), and
- Heading control algorithm (software).

Ship steering controller. The ship steering controller has the following use-modes:

UM 0: No operation,
UM 1: Hand steering,
UM 2: Heading control mode, and
UM 3: Tracking control mode.

For each use-mode, a set of services are offered to the user (person or other subsystem)

$s(0) = <$ *set_track*; *set_heading* $>$
$s(1) = <$ *set_track*; *set_heading* $>$
$s(2) = <$ *set_track*; *set_heading*; *keep_heading* $>$
$s(3) = <$ *set_track*; *keep_track* $>$.

The *keep_heading* service calculates the necessary rudder action in order to maintain the heading of the ship based on the measured rate and heading. The service can be defined as

$$\begin{aligned} \textit{keep\_heading} = \quad & < \psi, \psi_{\text{ref}}, \omega_m >; \\ & < \delta >; \\ & < \textit{heading controller} >; \\ & < \psi_{\text{ref}}\ \textit{defined} >; \\ & < \textit{none} >; \\ & < \textit{Gyro},\ \textit{rate sensor} >. \end{aligned}$$

The *keep_track* service calculates the necessary heading in order to maintain the track of the ship based on the measured rate, heading and position. The service can be defined as

$$
\begin{aligned}
\mathit{keep\_track} = \quad & < \mathit{track\_error}\ e,\ \mathit{track},\ \mathit{ship\ speed}\ v_1 >; \\
& < \psi_{\text{ref}} >; \\
& < \mathit{track\ controller} >; \\
& < \mathit{track},\ \mathit{reference\ defined} >; \\
& < \mathit{none} >; \\
& < \mathit{Gyro},\ \mathit{rate\ sensor},\ \mathit{speed\ sensor} > .
\end{aligned}
$$

The *set_track* service prompts the user to input waypoints for the desired track:

$$
\begin{aligned}
\mathit{set\_track} = \quad & < \mathit{waypoints} >; \\
& < \mathit{track} >; \\
& < \mathit{track\ planner} >; \\
& < \mathit{user\ input} >; \\
& < \mathit{none} >; \\
& < \mathit{electronic\ seamap} > .
\end{aligned}
$$

The *set_heading* service prompts the user to set a desired heading:

$$
\begin{aligned}
\mathit{set\_heading} = \quad & < \psi_{\text{in}} >; \\
& < \psi_{\text{ref}} >; \\
& < \psi_{\text{ref}} = \psi_{\text{in}} >; \\
& < \mathit{user\ input} >; \\
& < \mathit{none} >; \\
& < \mathit{none} > .
\end{aligned}
$$

In the above service definitions, some resources are not considered, for example electrical power. □

4.6.5 Building Systems from Components

Systems are built from the interconnection of different components. Components are interconnected because the services delivered by some of them consume variables which are produced by services of others. The *measurement* service of a sensor, for example, consumes variables produced by the environment of the system, and produces variables which are consumed by the *regulation* service of the regulator, which in turn produces variables which are consumed by the *power modulation* service of the actuator.

In the generic model approach, interconnections are taken into account by considering higher-level components which are composed of lower level ones. Therefore, systems are built following a bottom-up approach.

Indeed, system architectures can be described at different hierarchical levels. Sensors, actuators, process components are at the field-level. Higher-level components can be built from the aggregation of lower level ones at any hierarchical level. For example, the aggregation of a tank, an input valve, an output pipe, a level sensor and a regulator (with consistent connections between them) is a high-level component,

the "single-tank system". Whatever the component level, its generic model can be built defining its use-mode automaton, and the services which are available in each use-mode. Aggregation procedures have to be defined in order to build the generic model of high-level components from the generic models of the low-level components they are composed of. High-level services allow to fulfil the system mission, and the analysis of the overall system fault tolerance can be based on the search of the existence of different versions of high-level services.

Aggregation of operation modes. The generic model of a component is first given by its use-mode automaton. In each use-mode, the component is able to perform a set of services (each of them under a variety of versions) in order to achieve some prespecified objective. Recall that the use-mode automaton which describes a component is a graph $A(M,\ \tau,\ m^0)$. Let $A(M(k),\ \tau(k),\ m^0(k))$ and $A(M(l),\ \tau(l),\ m^0(l))$ be the deterministic automata associated with two components k and l, and let kl be a higher-level component, which aggregates these two ones. The automaton $A(M(kl),\ \tau(kl),\ m^0(kl))$ associated with the component kl is obviously contained in the asynchronous product of the two automata $A(M(k),\ \tau(k),\ m^0(k))$ and $A(M(l),\ \tau(l),\ m^0(l))$:

- $M(kl) \subseteq M(k) \times M(l)$
- $\tau(kl) \subseteq \tau(k) \cup \tau(l)$
- $m^0(kl) = \left(m^0(k), m^0(l)\right)$

Let $(\alpha,\ \beta) = \mu \in M(k) \times M(l)$. This means that the mode μ of the high-level device kl is defined as component k being in mode α and component l being in mode β. Since not every such association is meaningful, the set of modes $M(kl)$ has to be selected by the designer, by eliminating from $M(k) \times M(l)$ the non-significant or non-allowed associations.

Example 4.8 Temperature controller
Consider a temperature controller as a high-level component, obtained by the aggregation of three low-level ones, namely a temperature sensor, a PI regulator, and a heating valve. The use-modes of the low-level components are as follows:

Sensor: {*off, calibration, automatic*}
Regulator: {*off, on* }
Valve:{*off, manual, automatic*},

where the calibration mode of the sensor and the manual mode of the valve are used for maintenance operations. Then, the asynchronous product of the three use-mode automata gives 18 compound modes. Many combinations are inconsistent, e.g. (*off, on, manual*) or (*calibration, on, automatic*), leaving only three consistent modes to describe the temperature controller device. The three use-modes of the high-level component are, therefore, {*off, maintenance, automatic*}, and they are defined as follows from the use-modes of the low-level components:

$$\begin{aligned} \mathit{off} &= (\mathit{off}, \mathit{off}, \mathit{off}) \\ \mathit{maintenance} &= (\mathit{calibration}, \mathit{off}, \mathit{off}) \vee (\mathit{calibration}, \mathit{off}, \mathit{manual}) \\ &\quad \vee (\mathit{off}, \mathit{off}, \mathit{manual}) \\ \mathit{automatic} &= (\mathit{automatic}, \mathit{on}, \mathit{automatic}). \end{aligned}$$

□

Aggregation of services. Let $S(k)$ and $S(l)$ be the services offered by two low-level components k and l, and let us consider the high-level component kl which is their aggregation. Let $(\alpha, \beta) = \mu \in M(kl)$ be a consistent use-mode, then any combination of the services $S_\alpha(k)$ (available when component k is in mode α) and $S_\beta(l)$ (available when component l is in mode β) can be used. In other words, any *program* using the services of $S_\alpha(k)$ and $S_\beta(k)$ as *instructions* can be a service available in the mode μ. Again, every combination is not consistent, and only programs with functional interpretations in the application framework are to be considered.

Example 4.8 (cont.) Temperature controller
The following program defines a high-level service, available in the automatic mode of the temperature controller component:

Regulation service:
Repeat,
  Request the measurement service of the sensor,
  Request the calculation service of the regulator,
  Request the actuation service of the valve,
Until end of the regulation service.

Note that if the measurement service of the sensor is available under three versions, the calculation service under two versions, and the actuation service under two versions, then the regulation service is available under twelve versions. □

Hierarchical levels. System architectures can be described at different hierarchical levels. Sensors, actuators, process components are at the field-level (they exchange data at fieldbus level). Higher-level components can be built from the aggregation of lower level ones at any hierarchical level. For example, the aggregation of a tank, an input valve, an output pipe, a level sensor and a controller (with consistent connections between them) is a high-level component which can be named the single-tank subsystem. Whatever the component level, its generic model can be built defining its use-mode automaton, and the services which are available in each use-mode. Aggregation procedures have to be defined in order to build the generic model of higher-level components from the generic models of lower level components. High-level services enable fulfilment of the system mission, and the analysis of the overall system fault tolerance can be based on the search of the existence of different versions of high-level services.

4.7 Fault-Tolerance Analysis

Fault tolerance is defined as the possibility of achieving a given (set of) objective(s) in the presence of a given (set of) fault(s). In the generic model, objectives and services are associated with each use-mode. Thus, the system is fault tolerant in the current use-mode as long as services which allow to achieve the current objectives are available in this use-mode.

Therefore, the analysis of fault tolerance rests on three points.

1. Are there services which allow to achieve the current objectives?
2. How are these services to be managed when faults occur?
3. How are the use-modes to be managed when faults occur?

4.7.1 Relation Between Services and Objectives

The generic component model describes the normal behaviour of the system components, at any hierarchical level, since high-level components are built, following a bottom-up approach, from the aggregation of low-level ones.

At the highest hierarchical level, the system itself is modelled as a single component, which aggregates all the elementary components, and whose services correspond to the missions or objectives that it has to achieve. Different aggregation paths can be followed between the field-level (associated with elementary components) and the system level. A natural way of building the bottom-up aggregation procedure is to make use of the intuitive decomposition of the system into subsystems whose functions (and services) can be clearly defined.

System pyramidal decomposition. Hierarchical decomposition splits a system into a set of subsystems, which can themselves be further decomposed, each subsystem being associated with a clear functional viewpoint. For example, a chemical process can be decomposed into

- a subsystem which aims at controlling the pH,
- a subsystem which aims at controlling the level, and
- a subsystem which aims at controlling the temperature.

The pH control subsystem may be further decomposed into the valve controlling the acid inflow, the valve controlling the base inflow and the stirr motor.

Since some components may belong to several subsystems, it is convenient to use a pyramidal decomposition (Fig. 4.10) whose number of levels is decided by the

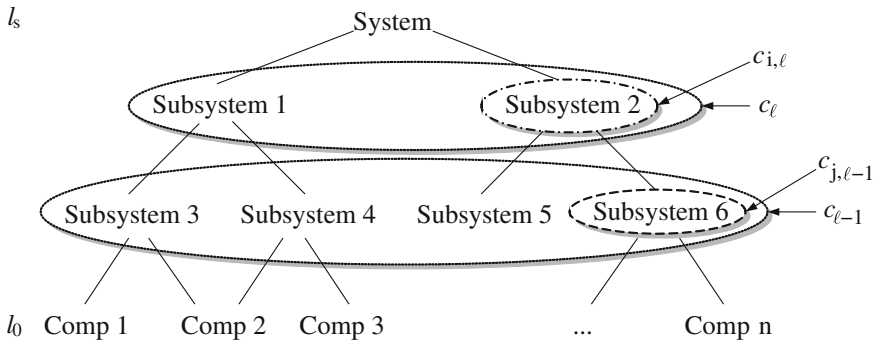

Fig. 4.10 Aggregation of low-level components into high-level ones

designer so as to obtain the view of the system which suits him best. Let $l = 1$ be the lowest decomposition level (the level of the field components) and $l = n$ be the highest decomposition level (the level of the system itself).

In a pyramidal decomposition, the following properties hold ($l \geq 2$):

- Each component of level $l - 1$ belongs to at least one component of level l.
- Any component of level l includes at least one component of level $l - 1$.

High-level services. Let C_l be the set of components modelled at level l, ($l = 1, \ldots, n$), and let $c \in C_l$ and $\gamma \in C_{l-1}$ ($l \geq 2$). Remember that the services of high-level components are behaviours which use the services of the low-level components they aggregate. Of course, not any combination of low-level services makes sense, and it is necessary, for the subsystems of the pyramidal decomposition to be consistent, that component $\gamma \in C_{l-1}$ is included in component $c \in C_l$ only if at least one service of component γ is used in at least one program which defines a service of component c associated with a known functional interpretation. Indeed, associating elementary services to define a high-level service is always realised to answer a functional requirement of the application. The system pyramidal decomposition is in this case, a real help define high-level services from low-level ones.

Let $S(\gamma)$ be the services offered by a component $\gamma \in C_{l-1}$ and let $c \in C_l$ be the aggregation of a set Γ of such components. Let

$$cons\,(c) = \cup_{\gamma \in \Gamma} \cup_{s \in S(\gamma)} cons\,(s)$$
$$prod\,(c) = \cup_{\gamma \in \Gamma} \cup_{s \in S(\gamma)} prod\,(s).$$

Note that $cons(c) \cap prod(c)$ may be non-empty, since some components in Γ may consume variables produced by some other ones. Note also that any relation between a subset of variables of $cons(c)$ and a subset of variables of $prod(c)$ can be obtained as the result of a program using the services of $\cup_{\gamma \in \Gamma} S(\gamma)$. If there exists such a relation which makes sense from a functional point of view in the system, then the subsystem $c \in C_l$ can be created at level l and the procedure which establishes such a relation can be defined as a service of the aggregated component c. Note finally that there might exist several subsets of components in Γ and several procedures which establish the same relation between the above mentioned variables. Then, the service exists under several versions. The set of all the versions of a service which can be obtained by aggregation of lower level ones can be found in a rather automated way, for simple kinds of programs composed of sequences and parallel executions.

Once the services of the subsystems have been determined at any level (including the overall system level), the ordering of the versions is left to the designer.

Example 4.9 High-level regulation service
Consider the three following low-level components:

- γ_1 is the temperature sensor, whose measurement service is defined by

$$< \theta,\ \hat{\theta},\ f_1,\ rqst_1,\ enable_1,\ res_1 >$$

where θ is the actual temperature, $\hat{\theta}$ is its estimate provided by the sensor, f_1 is the procedure which is used to produce the estimate: $\hat{\theta} = f_1(\theta)$. The request, enabling conditions and resources are not of interest here.

- γ_2 is the controller, whose computation service is defined by

$$< (\hat{\theta},\ \theta^*),\ u,\ f_2,\ rqst_2,\ enable_2,\ res_2 >$$

where θ^* is the temperature set-point, u is the control signal, and f_2 is the procedure which is used to produce the control signal: $u = f_2(\hat{\theta},\ \theta^*)$.

- γ_3 is the actuator, whose heating service is defined by

$$< u,\ \pi,\ f_3,\ rqst_3,\ enable_3,\ res_3 >$$

where u is the control signal, π is the delivered heating power, and f_3 is the procedure which is used to produce the heating power: $\pi = f_3(u)$.

Considering the set $\Gamma = \{\gamma_1,\ \gamma_2,\ \gamma_3\}$ as a candidate for aggregation into one higher-level component c, the above sets are $cons\,(c) = \left\{\theta,\ \hat{\theta},\ \theta^*,\ u\right\}$ and $prod\,(c) = \left\{\hat{\theta},\ u,\ \pi\right\}$, from which it is seen that the simple program sequence "measure, compute, actuate" can provide a relation between $\theta,\ \theta^*$ and π, whose functional interpretation is of course that of a "regulation" service.

Moreover, note that if the measurement service is provided e.g. under four versions (as in the following example), and the heating service is provided under two versions, then the regulation service is provided under eight different versions. □

4.7.2 Management of Service Versions

Consider a service s at the system level. It is a set of pre-ordered versions $s = \left\{s^j,\ j \in J(s)\right\}$. Each version can be used for the same purpose, namely to achieve the system objective(s) in a given use-mode, but the pre-ordering expresses a preference between them.

Two conditions have to be fulfilled for service versions to be enabled at some given time. First, the service must belong to the list of services of the current use-mode. This is a straightforward condition, which insures that only services consistent with the current objectives can be run.

The second condition is related with faults. Suppose that some resource from the set res^j is detected faulty by the fault diagnosis procedure at time t. Then, obviously, running the version s^j of the service s would produce incorrect values of the variables *prod*, and this should be forbidden, putting $enable^j = 0$. Note that this might be impossible, since faults might have ever-lasting delivery of some service as a consequence.

Example 4.10 Unavailable and ever-lasting services
Let *V_open* and *V_close* be two services delivered by an on/off valve, and suppose that the valve is blocked closed. Then, the service *V_open* becomes unavailable while the service *V_close* is permanent in time. □

Thus, the consequence of faults is that some services become permanent in time, while some others exist under a number of available versions which depend on the remaining, non-faulty resources. The status of these service obviously depends on the number of available versions, according to the following classification:

- at least one version is available: the service is available,
- no version is available: the service is unavailable.

Note that when more than one version is available, the lowest rank version of the service (which is the most preferred among the available ones) is to be run when the service is requested. Note also that the severity of the failure of a given resource with respect to the service can be evaluated by counting the number of versions which still are available after the failure has occurred. A resource for which this number is zero, or whose failure causes the service to run permanently in time is called a *critical resource*.

Example 4.11 Management of service versions in an intelligent sensor
Consider again the *measurement* service of the temperature sensor and suppose that it includes two redundant transducers and an observer. Let

$$y_1(t) = x(t) + \varepsilon_1(t), \quad \varepsilon_1(t) \sim N(0,\ \sigma_1)$$
$$y_2(t) = x(t) + \varepsilon_2(t), \quad \varepsilon_2(t) \sim N(0,\ \sigma_2)$$

be the two local measurement equations, and

$$\hat{y}(t) = f(z_1(t), z_2(t))$$

be the observer algorithm, where $z_1(t),\ z_2(t)$ are remote measurements obtained from a local area network communication (LAN) system. The following table gives the different versions of the *measurement* service which are provided by this sensor, along with their ranking.

Class	Procedure	Faultsituation
0	$y(t) = \frac{1}{\sigma_1+\sigma_2}(\sigma_2 y_1(t) + \sigma_1 y_2(t))$	no fault
1	$y(t) = y_1(t)$	T_1 ok., T_2 faulty, A/D ok.
1	$y(t) = y_2(t)$	T_2 ok., T_1 faulty, A/D ok.
2	$y(t) = \hat{y}(t)$	T_1 and T_2 or A/D faulty, LAN ok.

Suppose that the measurement request is issued by the system clock according to some sampling period, and that the measurement service is consistent with the current use-mode. Then, it will be provided under version 0 if both transducers are operating well, and under version 1 in the presence of a single transducer fault (note that the two versions 1 are mutually exclusive,

so that no conflict is possible). If the local Analog to Digital Converter (ADC) fails, version 2 can still be used, and the service will become unavailable only when local measurements and fieldbus communication will be all faulty. Remind that the local ADC was a critical resource when no observer was included in the sensor. Note also that version 2 of the measurement service might be much more unprecise than versions 0 and 1, thus the service would be degraded when this version is used. However, it would still be acceptable, otherwise version 2 should never have been included in the list of versions by the design engineer. □

4.7.3 Management of Operation Modes

Remember that the set of services is organised into use-modes, whose behaviour is described by a deterministic automaton. Let $A(M,\ \tau,\ m^0)$ be the use-mode automaton at the system level

- $M = \{m_i,\ i \in I_m\}$ is the set of the use-modes. Remember that each of these modes m_i is associated with the set of the services $S_i \subseteq S$ by which it is defined,
- $\tau = \left\{\tau_{ij},\ i,\ j \in I_m\right\}$ is the set of transitions,
- m^0 is the initial use-mode.

Critical services. In this chapter, use-modes have been further associated with objectives which have to be fulfilled thanks to those services. Let O_i be a set of objectives associated with use-mode m_i. As long as the set of services $S_i \subseteq S$ associated with m_i are available, the objectives O_i can obviously be achieved (otherwise, the component would be inconsistently designed). Note that this is true, by definition, whatever the version of the service. Versions of high rank provide degraded service, thus achieving the objective in a degraded but still acceptable manner. If not, the versions of that rank should not have been included in the list of possible versions of the service.

Suppose now that some fault has occurred such that some services of S_i become unavailable or run permanently. Then some objectives of O_i might become impossible to achieve. Let *critical services* be services whose unavailability or permanent running implies that at least one objective of the mode to which they belong cannot be achieved. Then, the $A(M,\ \tau,\ m^0)$ automaton model is extended as follows: $M = \{m_i,\ i \in I_m\}$ is the set of the use-modes. Each mode m_i is associated with the set of objectives O_i and the set of the services $S_i \subseteq S$ which is decomposed into $S_i = S_i^c \cup S_i^{nc}$, where S_i^c are the critical and S_i^{nc} are the uncritical ones.

Example 4.12 Critical service in the single-tank system
Consider the single-tank system given in the introduction, used in a food industry batch production process, and suppose the current use-mode is UM_2: processing the batch. In that use-mode, the system objective is to regulate the temperature, and the regulation service is thus a critical resource, since UM_2 objective cannot be achieved if this service is lost. □

Staying in a mode. Consider the system operation in a given current use-mode, and suppose that faults occur which cause the loss or permanent running of services (remember that as long as at least one service version is available, the service is

not lost). When non-critical services are lost or run permanently, the system can obviously remain in the current use-mode, since this use-mode objectives can still be achieved—eventually in a degraded manner—and thus the system is fault tolerant with respect to the current objectives and the current fault situation.

On the contrary, when critical services of the current use-mode are lost or run permanently, the objectives associated with that use-mode can no longer be achieved, and the system is to be given other objectives. This strategy is called an objective reconfiguration strategy. The only way it can be implemented is by firing a transition towards another use-mode, whose objectives will become the current ones (for example change the production recipe, or stop the production and transfer the system to a safe state, in which maintenance can be undertaken).

In general, several other use-modes can be reached from the current one, and the choice of the destination use-mode (i.e. of the new system objectives) is a difficult decision problem, which has to be considered in the system design stage. Unless the system objectives can be ranked according to a total ordering relation, the solution to that problem can in general only be partially automated, thus leaving a very important role to human operators in fault situations.

Transitions between modes. When objective reconfiguration is necessary, the system is commanded to another use-mode whose objectives will become the new ones. The system should, obviously, be able to achieve these new objectives, which means that in the destination use-mode, no critical service is unavailable nor is permanently running as a result of the current fault situation.

This remark is also valid when the transition does not follow from an objective reconfiguration strategy but from the normal operation of the system.

4.8 Exercises

Exercise 4.1 Fault propagation analysis for industrial actuator
Consider the component block diagram of the position servo shown in Fig. 4.11. The blocks in the figure are motor, amplifier, controller, potentiometer, gear and reference.

1. Construct a fault propagation matrix M_{p} for the potentiometer (use the FMEA matrix from p. 102). Use angle as input and voltage v_C at terminal C as output. Consider only the fault f_{p1}, which indicates the broken wire at C.
2. Using the above figure, define the component architecture in form of the directed adjacency matrix $\boldsymbol{D}$.
3. Determine which node is an input node from the adjacency matrix.
4. Determine the number of closed loops and the length of these.

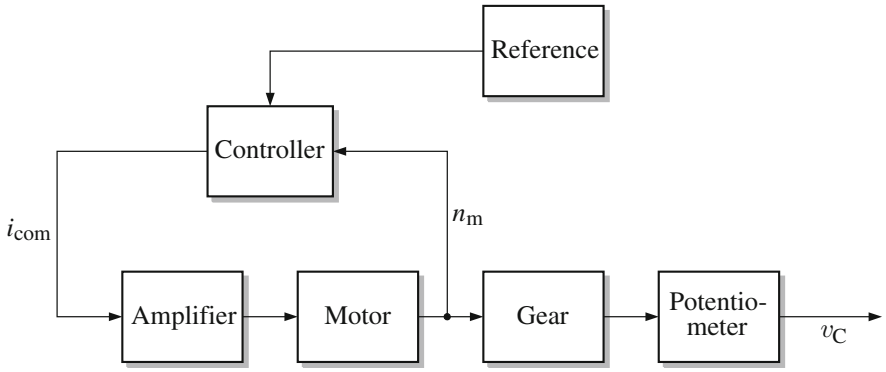

Fig. 4.11 Component diagram for speed loop part of the industrial actuator

The combined fault propagation matrix for motor and amplifier is defined by input and internal faults:

f_{a1} low gain in amplifier
f_{m1} brush partial disconnect
e_{a1} low input command
e_{a2} high input command
e_{a3} fluctuating input command

output:

e_{n1} low output speed
e_{n2} high output command
e_{n3} fluctuating output speed
e_{n4} output speed not related to input command

$$\boldsymbol{M}_{am} = \begin{pmatrix} 1\,1\,1\,0\,0 \\ 0\,0\,0\,1\,0 \\ 0\,1\,0\,0\,1 \\ 1\,1\,0\,0\,0 \end{pmatrix}.$$

Without further explanation, assume that the gear has the propagation matrix

$$\boldsymbol{M}_{g} = \begin{pmatrix} 1\,0\,0\,0 \\ 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \end{pmatrix}.$$

5. Determine the fault propagation from i_{com} to θ_m, using $\boldsymbol{M}_{am}$, $\boldsymbol{M}_g$ and $\boldsymbol{M}_{\text{p}}$.
6. Compute the inverse fault propagation matrix, i.e. observe the end effects on θ_m and see which faults or input effects could cause the end effects. □

Exercise 4.2 Component-based analysis of battery charger
A battery charger component diagram is given as shown in Fig. 4.12. The behaviours of the individual components are not known in detail but are given as

$$
\begin{aligned}
&\textit{Power stage mean current:} && I = d \cdot \max(V_{\text{sup}} - V_{\text{bat}}, 0)\\
&\textit{Current control:} && d = f_i(I_{\text{com}} - I_{\text{mes}})\\
&\textit{Voltage control:} && I_{\text{com}} = f_v(V_{\text{ref}} - V_{\text{mes}})\\
&\textit{Current sensor:} && I_{\text{mes}} = I\\
&\textit{Voltage sensor:} && V_{\text{mes}} = V_{\text{bat}}\\
&\textit{Harness:} && I_{\text{bat}} = I\\
&\textit{Battery voltage:} && V_{\text{bat}} = \frac{\alpha_{\text{bat}}}{C_{\text{bat}}} \int_0^t I_{\text{bat}}(t)\,\mathrm{d}t, \quad \alpha_{\text{bat}} \simeq 0.7.
\end{aligned}
\tag{4.4}
$$

1. Define one fault for each of the components: PWM converter, controller block (current and voltage control), current sensor, voltage sensor. Assume that the supply voltage and battery cannot fail.
2. Determine the fault propagation matrices for these components.
3. Determine the closed logical loops in the battery charger.
4. Cut the loop at the signal d_{com} and determine whether the logical loop has a solution. If not, define d_{com} as an ancillary input.
5. Determine the end effects (on I_{bat} and V_{bat}) for the faults you defined.
6. Express the inverse propagation and list the end effects.
7. Suggest how a fault in the current sensor could be accommodated. □

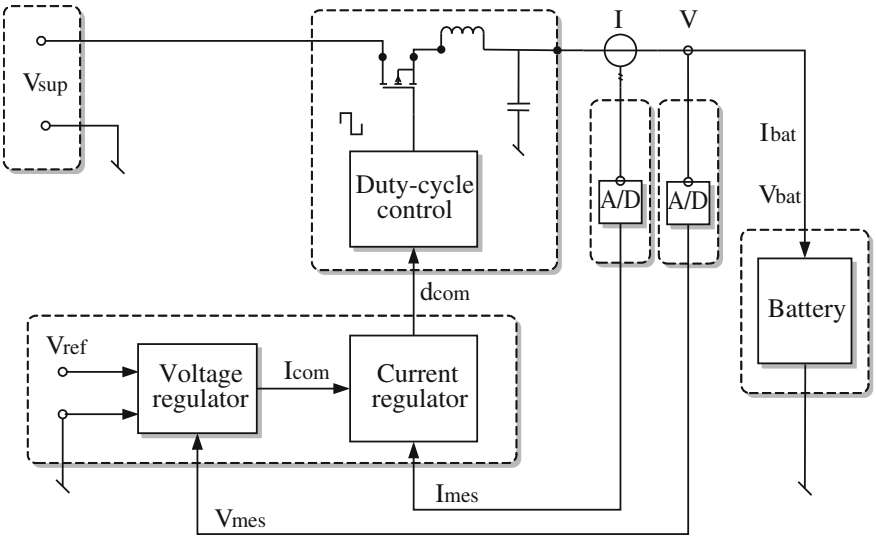

Fig. 4.12 Component diagram of battery charger

4.9 Bibliographical Notes

Architecture models. Basic bibliographical notes on architecture and generic component models were given in Chap. 2. With the purpose of describing complex interconnected systems, modelling extensions have been suggested in form of a bottom-up procedure, which allows to describe high-level devices by the interconnection of low-level ones, see [49, 50].

Generic component models. A semantics for services based on generic component models was developed in [324] where a graphic analysis was found to be very useful. A review of the use of graphical methods was provided in [47].

Failure modes and effects analysis. FMEA is a classical and widely used tool in industry [147]. Presentation of a matrix formulation suited for computational treatment of FMEA schemes was first presented by [192]. The fault propagation analysis was proposed in [22] and further elaborated in [42].

Fault reasoning and data validation. Reasoning and detection of faults, and hierarchical data validation was introduced in [14], while alarm filtering applications were considered in [258]. Systematic analysis of fault propagation [22, 42] was shown to be an essential tool for determination of severity of fault effects and for assessment of remedial actions early in the design phase. Reference [72] presented a comprehensive diagnosis methodology for complex hybrid systems and an application to aircraft power generator diagnosis was included in [358].

Fault tolerance. The analysis of fault propagation has also penetrated to design of industrial systems, Fault-tolerant steering by wire was obtained in [35] and a fault-tolerant three-phase speed drive for AC motors was described in [118].

Distributed systems. Fault tolerance in distributed systems has mainly been based on modular models [165]. The generic component model was used for defining the faulty resources management and the mode management layers [120]. Reconfigurability analysis was developed more recently [121]. Important application areas is prognosis and health management and an application for a distributed medical equipment a distributed concept was illustrated in [102].

Other applications and trends. Consequences of faults and how faults propagate attain continuous attention and documentation of fault effects properties are mandatory in product development in many areas. Transportation systems are among those expected and required by society to be safe. In these and many other systems, the complexity is an obstacle when attempting to conduct covering assessments. The complexity of computer-based safety systems was considered in [271], where a new method was proposed for joint design optimization and engineering failure analysis. The criticality of component failures in the petrochemical processes were treated in [136].

Software. All major functionalities in automation systems depend on software implementation and computer hardware. The complexity of software is immense and

analysis of failure modes and of fault propagation in software is a discipline in itself, which is not within the scope of the present text. Nevertheless it could be mentioned that [135] analysed and experimented with software fault injection aiming at FMEA.