Best Practice in Robotics (BRICS)

Grant Agreement Number: 231940

01.03.2009 - 28.02.2013

Instrument: Collaborative Project (IP)

Robust autonomy

Yury Brodskiy, Stefano Stramigioli, Jan Broenink,
Peter Bredveeld, Cagri Yalcin

Deliverable: D6.1

Lead contractor for this deliverable:	University of Twente.
Due date of deliverable:	November 01, 2010
Actual submission date:	October 19, 2010
Dissemination level:	Public
Revision:	0.01

Contents

Executive summary

Deliverable D6.1 provides specifications for the research performed in Work Package 6 (Robust Autonomy) of the European research project BRICS. The main purpose of this document is to review a number of techniques to increase robots' level of robust autonomy and to give recommendations on how to better start tackling the issues within the BRICS project. The cross sectional nature of the WP6 is reflected in this work by addressing a subject of robots robust autonomy in the development process, architectures(Software&Hardware) and algorithms.

More specifically the deliverable contains initial results on the task 6.4 "Design principles, implementation guidelines, and evaluation criteria for robust autonomy". The main goal of the task is to define the best methods and design patterns to achieve robust autonomy. The initial steps towards this goal is done by reviewing and combining the results obtained in the dependable computing research field and experience from the design of safety critical systems. The results from task 6.1 "Inventory and classification" are used to create a coherent overview of the field. The proposed directions of future research are cooperative recovery support in robots' architecture, development of reusable detection and recovery components.

The deliverable also contains midterm results of the results on task 6.2 "Identification of criteria for robust autonomy". Herein is a proposal for the number of metrics that allow to identify the level of robust autonomy. The metrics are developed based on the concept of the 'human-centered automation' applied to abnormal event management process.

1 Introduction

The development of the mobile manipulation robots that can perform its duties for a long time in an unstructured environment with a limited human assistance is of particular interest to the robotic community. The BRICS project addresses this interest by developing the methodology, the framework and the tool-chain for such applications. One of the important facets of such a methodology is a support for a controllable investment into robustness of robots autonomy.

It is necessary to define the basic concepts applicable for robots robust autonomy. Definitions of the basic concepts will help the communication and the cooperation among scientific and technical communities during development of a system with an explicit level of robust autonomy. This is addressed by developing a crisp one sentence definition of robust autonomy and identifying relations with a similar research field in Chapter 2.

A robot development should be structured in a way that reveals decisions that affect non-functional requirements such as robust autonomy. The engineering process can be described as a combination of design procedures and artefacts (deliverables). An artefact is the most explicit way to ensure fulfilment of the requirements. Therefore, it is good practice to include robust autonomy requirements in it. The evaluation criteria of robust autonomy presented in Chapter 3 are aimed to support this practice. The design procedures that allow to fulfil the requirements of robust autonomy are reviewed in Chapter 4; it also contains a taxonomy of faults that could be used as a support for the design procedures.

The developments in the dependable computing research field and safety critical system design provide numerous techniques that are applicable to robotics robust autonomy. The review is presented in Chapter 5. The initial guidelines and challenges for the architecture that will support the autonomous and robust behaviours of the system are presented in section 5.2. The taxonomies of algorithmic solutions for robust autonomy are presented in sections 5.3, 5.4.

2 Background information

2.1 Basic concepts

In section 2.2, it will be shown that robust autonomy is a super set of autonomous behaviours and behaviours described by fault tolerance. A fault tolerance is a concept defined in dependable computing. The field of dependable computing is concerned with justification of trust in software systems. Research in this area has produced a number of definitions that aim to describe the processes of software failures. The developed definitions are elaborated in the terms of system interaction and behaviours which make them general and applicable to other spheres of engineering. The basic taxonomies and definitions of dependable computing have been already described elsewhere [2],however it is convenient to include a brief account of it here with emphasis to the robotic domain.

We discuss robot behaviour in term of system interaction. A system consists of a number of mutually related parts that are relevant for the behaviour in which we are interested. It is an elementary entity in this analysis. Everything outside the system is called an environment. The barrier between system and environment is called system boundary. A system can be compound, consist of several systems, or atomic. The system that uses a service of an other system is called a user.

The behaviour of a robot can be analysed as hybrid discrete-continuous automata. Such systems have internal and external states. Sequences of external states are called system behaviours. The desired behaviours are called services. Other behaviours are called failures. The cause of a failure is a fault. The important deviation in the study of robust autonomy is the concept of acceptable service presented in the next section. It is the ability to retain some functionality in the presence of faults. This concept is utilised as one of the metrics for robust autonomy of the robot 3.4

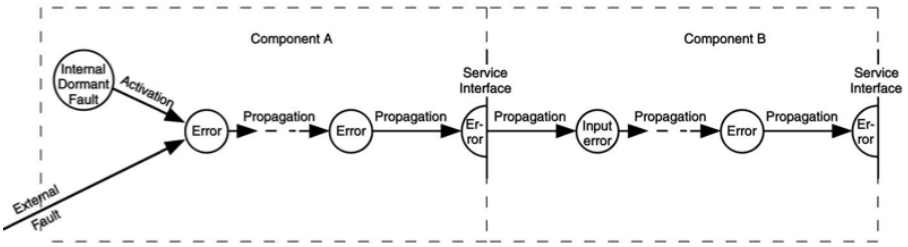


Figure 2.1: Fault, error, failure propagation process [2]

A fault, an error and a failure propagation process is presented in figure 2.1. Any fault is a deficiency in the system design. When a fault is active the system is transferred into an error state. The error propagation in the system generates the failure to deliver the service. Using these concepts we will elaborate the definition of robust autonomy.

2.2 Definition of robust autonomy

The autonomy in a robotic context is the potential of an intelligent man-made mechanism to interact with its environment without human intervention in its decisions about this interaction. The robustness of robot autonomy should be such that this interaction remains sufficiently safe, accurate, etc. (i.e. within specifications) over a maintenance interval. [5]. Since robust autonomy is a compound term , we find it necessary to clarify the meaning and relations to similar fields of research. Only few definitions could be found in the literature, more often the expression has been used to reinforce the meaning of one part. For example, in the

2001 AAAI Spring Symposium on Robust Autonomy, only one paper contains a description of the term and 2 out of 20 papers contain indirect description of the term:

- "Robust autonomous systems will need to be adaptable to changes in the environment and changes in the underlying physical system." [16]
- "Robust autonomy on the part of software agents requires, at least in part, the ability to deal intelligently with novel and unexpected situations." [9]
- "One aspect of robust autonomy is the ability to react to faults and special conditions." [30]

As it can be seen, robust autonomy is mostly considered as ability of the system to react to changes in the environment, unexpected situations or special conditions. The phenomenas to which the system is supposed to react can be summarised as abnormal events, happenings outside the normal workflow. Without special considerations the abnormal events become the cause of the system failure. From this point of view, robust autonomy resembles the means to achieve system dependability.

Future developments of the definition exhibit a lot of interrelations with the dependability computing research. Correlation between fault tolerance and robustness for autonomous systems is discussed by Lussier which concludes that robustness and fault tolerance both characterise the resilience of a system towards particular adverse situations; robustness characterises resilience towards uncertainties of the environment, and fault tolerance characterises resilience towards faults affecting the system resources.[20]. However this conclusive separation of the fault tolerance and robustness is self contradictory since the process of defining system boundary is involved. Depending on system boundary definition the same mechanisms will appear to be fault tolerance or robustness. This inconsistency is reflected in the following statement of Lussier: "Development faults are hardly addressed by fault tolerant mechanisms in autonomous systems; robustness techniques somewhat compensate this problem, but are surely insufficient for critical applications."[20]. Therefore we assume that robustness is a synonym for fault tolerance.

As a requirement for an autonomous system Lussier et al. define the concept of "acceptable service" since the operational field of the autonomous robot is an open environment where unexpected adverse situations are inevitable, therefore, a correct service cannot be guaranteed. This directly brings the need of alteration on the definition of a correct service given in the context of traditional dependable systems. Hence, acceptable services corresponds to the less successful versions of the correct services where less objectives are achieved. As a result, we see the concept of an acceptable service as an important part of the robust autonomy definition.

We suggest that the need for additional mechanisms directly depends on the level of autonomy and the related architectural constraints of the system. On the other hand, all of the above descriptions contain the implicit assumption that an autonomous system operates without or under limited human supervision. Taking this into account, the final definition to be used in our context can be phrased as:

Robust autonomy is the ability of the system to react on both explicitly-specified adverse situations as well as unexpected adverse situations in both the environment and the underlying system (hardware and software) without or under limited human guidance in order to complete its mission in an acceptable way.

3 Evaluation criteria

3.1 Model of autonomous fault tolerance

The robust autonomy is a non-functional requirement and as such does not have well-defined specifications or metrics. The foremost step in the development towards robustness for autonomous robots is to determine the concepts that can be used for optimisation of design solutions.

The amount of control that a robot has over its actions, reflects different levels of autonomy [13]. The ability to prioritise and make decisions is a property of a system with a high-level of autonomy. This ability becomes more apparent in the case of abnormal events. Exceptional situations unforeseen by the main operation workflow, confront a robot with the challenge of choice. For example, an automatic ground collision avoidance system (auto GCAS) on an aircraft has a priority of the crew safety, which can override a direct command from the pilot controls in case of a ground collision course [28]. In this case the automatic system exhibits the highest level of autonomy, taking over the responsibility of making decisions.

The process of a robot service can be seen as a sequence of states of the system consisting of a robot and the environment. During a normal workflow transitions between different states are determined by certain pre-programmed behaviours. However, deviations due to irregularities require an unpredicted switch in the behavioural model. Such a switch might require employing one of the predefined solutions or developing a new one based on the system goals and constraints. The ability to independently detect abnormal events and to recover from them makes a robot autonomous and robust. In essence, *robust autonomy* is the ability of a robot to deal with abnormal situations with minimal human involvement [16, 27].

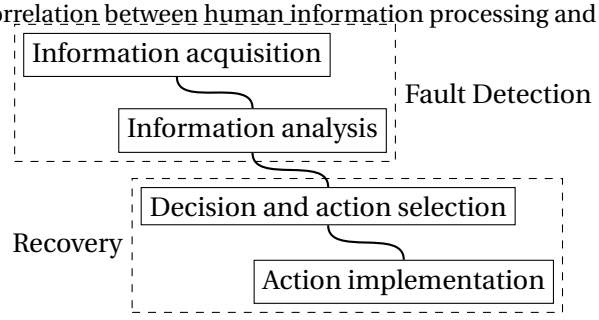
The metrics of robust autonomy can be identified through the analysis of processes taking place in the system upon an event. An abnormal event is a violation of the operating assumptions and thus can be treated as an external fault on the boundary of the system consisting of the robot components. Following Avizienis, [2] taxonomy of Dependable and Secure Computing, the fault handling can be modelled by two processes: fault detection and recovery from it. Metrics of robust autonomy can be defined as a combination of characteristics of the fault detection and recovery processes.

The field of service robotics is based on the idea of robots co-existing with humans; therefore, the human-machine interaction aspect of robust autonomy should be quantified too. The concept of 'human-centered automation' is based on the analysis of human information processing and on the amount of tasks devoted to automated systems (robots). To provide a robot with autonomy means to (partially) exclude the human operator from the robot's exception handling processes. By adding new levels of automation a designer creates a more independent, and thus more autonomous robot. Every layer of automation, directed towards this goal, provides a partial replacement or minimisation of human support. This implies that excluding humans from robot exception handling is a gradual process.

Human actions intended to help a robot to solve an exceptional situation can be analysed by means of a model of human information processing. A four-stage model was proposed by Parasuraman to analyse levels of human interaction with automation. This model is oversimplified and debatable, but it highlights the main activities and as such suffices for this kind of analysis [24].

The correlation between this model of human information processing and the model of the strategy for fault handling of Avizienis is displayed in figure 3.1. The information acquisition and analysis is equivalent to fault detection. The recovery is corresponding to action selec-

Figure 3.1: Correlation between human information processing and fault handling



tion and implementation. Both models are competent but a human information processing can provide more details. Based on the two models discussed above for abnormal event management we will define separate metrics for fault detection (section5.3), decision and action selection (section3.3) and action implementation (section3.4).

3.2 Information acquisition and analysis

An important feature of an autonomous robot is its ability to detect when its normal workflow is interrupted and to identify the cause of this interruption. Different types of Fault Detection and Isolation (FDI) systems are employed to achieve it. The FDI system is responsible for the data acquisition and analysis process. The system performs a transformation from raw sensory data to a possible cause of failure. To evaluate the quality of this process, it is necessary to review the characteristics of the FDI system. Similar analyses have been done for FDI systems in chemical process engineering by Venkatasubramanian [29]. The resulting characteristics most relevant to the robust autonomy of a robot can be summarised as follows:

The response speed is characterised by a time delay between a failure event and the system response. In most robotic systems, this delay should be between certain boundaries. The minimum response speed is determined by the dynamics of the system and by the consequences of the failure. For example, in a simple detection system it is the delay between deviation of the process parameters and detection of this deviation. Filtering and thresholds, added to the system to avoid false detection, generally create this delay.

The robustness of the abnormal event management unit can be measured by the amount of noise required to be put into the sensory data in order to trigger a false detection. A robot performs its functions in an unstructured environment. This yields fluctuations of the system states. The application must remain stable in presence of these disturbances, since false detections can also jeopardise the system autonomy.

Isolability is the ability to identify a system part in which a fault has occurred. It can be measured as a percentage of the system that is considered faulty upon a given event. It is a particularly important part of the information analysis process. In general, a robot is preprogrammed for a certain number of behaviours, each addressing particular goals. In order to select the correct action, it has to consider all constraints imposed by an abnormal situation. For example, a robotic arm that has several sensors on a joint can isolate failures of one without losing its performance. In case the arm has only one sensor per joint, failure of the sensor requires isolation of the whole joint thus reducing the working envelope.

Novelty identifiability is the ability of a system to detect new (unknown) faults that were not defined at design time. If a system is constructed in a structured way with attention to fault identification at every level of abstraction and component, it could isolate failed

Table 3.1: Levels of Decision and Action selection automation

		A robot:
High	10.	decides everything, ignoring a human
	9.	decides to inform a human or not
	8.	informs a human on request
	7.	informs a human after execution
	6.	gives a human time to cancel execution
	5.	requests approval before execution
	4.	suggests one alternative
	3.	offers a prioritised list of actions
	2.	offers complete information on its status
Low	1.	offers no assistance: a human takes decisions

elements without a priori knowledge of the cause. The metrics can be measured the same way as isolability but for new (unknown) faults.

Multiple identifiability - Continuous work for a long period of time might require from a robot to react on simultaneous faults. Multiple identifiability characterises how many simultaneous faults a system can recognise and respond to.

The Correctness and Classification probability estimate describes the probability of a correct response to a certain fault. An unstructured and noisy environment can affect the identification and isolation processes in such a way that the system is able to detect irregularity in its behaviour, but is unable to classify it properly, resulting in an incorrect response.

3.3 Decision and action selection

The decision and action selection layer reflects the process of choosing an appropriate response to a classified abnormal situation. Analysing this part of the system behaviour from the point of view of human involvement provides a generalised approach. The results of this analysis are independent of the robotic context.[1] The level of autonomy does not pose constraints on the structure of information processing of the robot, since it is evaluated based on the output.

It was noted in the introduction that the level of elimination of human influence from the robot exception handling process can vary across a continuum of levels. Table 3.1 shows a 10-point scale of levels in decision automation proposed by Sheridan [24]. The lowest level robot requires complete human support in selection of the appropriate action before it starts execution. This level is similar to human-robot cooperation in tele-robots when resolving exceptional situations is exclusively dependent on operator skills. At this level an operator has to make a choice without robot assistance. At level 2 the system performs an analysis and provides the operator with FDI information, but there is no automated mapping between the actions available to the robot and the detected exceptions. At the 3rd level the system provides limited assistance in making a choice. The operator receives a prioritised list of available actions. At the 4th level of automation that is similar to an expert system behaviour, it selects one option based on all existing constrains and goals. The 5th level of automation is the lowest at which the system is completely dependent on the operator. An example of a system at level 4-5 of automation is a Spacecraft Emergency Response System (SERS) [4].This system supervises the spacecraft, but depends on the operator decisions. At levels 6 to 9 the operator can interfere

[1] The robotic context is a combination of the robot, its task and its operating environment.

Table 3.2: Levels of ability of robot to implement action

	A robot:	
High	6.	actively changes itself and/or the environment to solve the problem before completing the task
	5.	reconfigures itself to ignore the problem and to complete task as good as possible
	4.	requests assistance and suspends execution until the detected problem has disappeared
	3.	continues execution neglecting the problem
	2.	suspends execution until the detected problem has disappeared
Low	1.	interrupts execution by shutdown

with the robot decision, but is not required in the process. The highest level represents complete independence in the process of choosing the action. A human is ignored in the decision process at this level of automation.

3.4 Action implementation

In order to evaluate a robot's ability to implement actions, we propose to use the concept of *Quality of service (QoS)*. In service robotics the robot is designed to deliver multiple services. Unstructured and linguistic task descriptions leave room for deviations in the results. Therefore, it is possible to describe a robot state as a continuous approximation of an ideal result. An abnormal event changes the robotic context by introducing a new constraint, a new goal or an alteration in the robot-environment system state. In other words, an abnormal event brings the system into a state where one or several constrains are violated. An appearance of new restrictions might create an over-constrained system, such that the ideal solution can not be reached anymore. The concept of quality of service reflects the distance between the optimal solution, i.e. the solution a system can reach during normal workflow, and the solution the system can reach in the presence of faults.

To give an example of quality of service measurement, we will apply this concept to the Middle-Size RoboCup omni-directional robot developed by Graz University of Technology [12]. This robot can retain functionality in the presence of faults, revealing a high level of robust autonomy. The actuation system of the robot consists of 3 omni-wheels. The goal of the actuation system is to move the robot along a desired trajectory with a given orientation. It is a fully actuated process. Failure of a motor introduces a new constraint for the system and it becomes impossible to fulfill the main goal. However, the abnormal event management implemented in the robot can reconfigure the controls and planners to generate a new trajectory and use the remaining omni-wheels to move along the desired path, but without keeping the desired orientation. In this case the loss of one degree of freedom is not fatal to all tasks at hand. Since the robot retains 2 out of 3 degrees of freedom, we can say that it can provide *around* 60% of its quality of service, although the real application will dictate the importance of each degree of freedom which should be expressed in form of weight factors.

The ability of the robotic system to implement actions can be evaluated based on its influence on the context. At the first level a robot is limiting its influence by interrupting the workflow through shutdown of the involved components. At the second level a robot can suspend execution waiting until conditions will satisfy the workflow requirements. A next level robot persistently tries to continue execution neglecting the problem. At the fourth level a robot is looking for assistance to solve the problem. At level five a robot reconfigures itself in order to ignore the

problem and to complete the task as good as possible. At the highest-level, a robot is actively changing itself or the environment to solve the problem before completing the task.

3.5 Conclusions

The ability of a robot to deal with unexpected situations is an essential requirement for robot-human co-existence. This ability of robust autonomy stands apart from demanding functional requirements, but it is a barrier that separates a service robot at home from a prototype in the lab. In order to support designer decisions the evaluation of robust autonomy should be based on precisely defined concepts and measures. In this part we have combined several different approaches from other application areas to create a measurement framework for robust autonomy evaluation.

We have defined robust autonomy as the ability of a robot to deal with abnormal situations with minimal human involvement. The models of abnormal event management were used to analyse this ability. A three-stage assessment approach reflects separation of concerns in the system. Fault detection represents the context-based information acquisition and analysis. The decision and action selection reflects the amount of responsibility the robot is allowed to take. Action implementation indicates the robot's ability to fulfil its goals.

This three-stage assessment approach makes it possible to use this framework for the layered design of an abnormal event management process while interdependence between different layers aids to create a well-balanced system.

4 Fault forecasting

It has been proven effective to start development of fault tolerant system with systematic analysis of possible faults. A number of techniques were developed to itemise, assess probability and indicate relation between system faults. In the following taxonomy of Avienzis, such techniques are called fault forecasting. The application of fault forecasting techniques is widely spread in the development of safety critical systems. It is most often applied for the development of hardware elements. However it is interesting to note that in software developments fault forecasting techniques have proven to be more efficient than diversity in design(N-version programming) or the collaborative development [21]. The fault forecasting methods rely on:

- Scenario based analysis - Failure mode effect and criticality analysis (FMECA) [23], Software Architecture Reliability Analysis approacH (SARAH) [26]
- Cause and effect analysis - Fault Tree Analysis (FTA), Event Tree Analysis (ETA)
- Risk assessments - Stress Strength Analysis[3], Reliability prediction [1]

The scenario based approach to fault forecasting is a bottom up analysis of the system reaction to element failures. Techniques such as Failure Mode Effects and Criticality Analysis start with some assumptions related to the failure of some component. The behaviour of the system is examined after fault activation; this is the scenario. The resulting effects on the system and the environment are then assessed to decide if the system requires any alteration. The analysis process is harnessed by the strict documentation guidelines, exemplified in [23]. For FMECA all results are collected in a table like the one on figure 4.1

FAILURE MODE AND EFFECTS ANALYSIS

SYSTEM _____ DATE _____
INDENTURE LEVEL _____ SHEET _____ _____
REFERENCE DRAWING _____ COMPILED BY _____
MISSION _____ APPROVED BY _____

IDENTIFICATION NUMBER	ITEM/FUNCTIONAL IDENTIFICATION (NOMENCLATURE)	FUNCTION	FAILURE MODES AND CAUSES	MISSION PHASE/ OPERATIONAL MODE	FAILURE EFFECTS			FAILURE DETECTION METHOD	COMPENSATING PROVISIONS	SEVERITY CLASS	REMARKS
					LOCAL EFFECTS	NEXT HIGHER LEVEL	END EFFECTS				

Figure 4.1: Example of FMEA worksheet [23]

The Cause and effect approach is a fault forecasting technique that consists of a top down systematic enumeration of the plausible system events(fault/failures). An event is a failure of a system element, it is a basic element of a fault tree analysis. In Fault Tree Analysis, the most evident failure of the system such as failure to fulfil the main task is considered top event. Any event or combination of events that cause the top event to happen are lower in the event hierarchy. The hierarchy of the events is captured by a logic three. Logic gates such as AND, OR,

XOR represent the correlation between lower and higher events. The analysis is concluded with a basic event. A basic event is the failure of the atomic element of the system, it can not be affected by the designer. If changes in the system are considered necessary the analysis should be repeated for the part of the system that has been redesigned. An example of a fault tree is presented in figure 4.2.

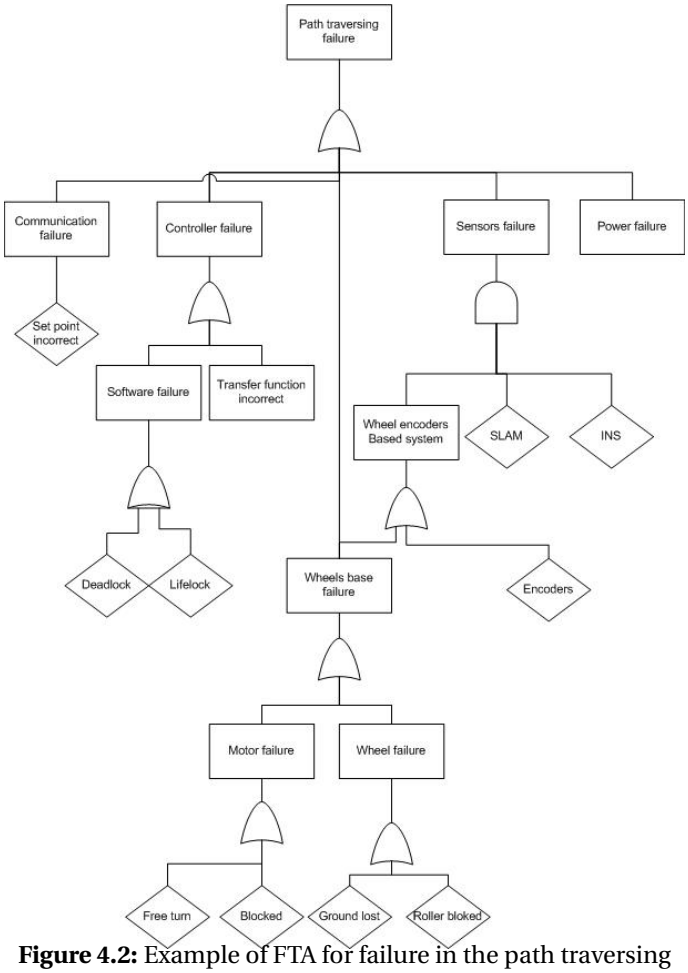


Figure 4.2: Example of FTA for failure in the path traversing

Risk assessments is a fault forecasting technique based on failure probability estimates. The probability estimate is based on experimental data. There are two main approaches possible: either the system is tested as whole or system parts are tested separately. If the system composition is known and the reliability estimate of all parts are known then it is possible to compute the probability of failure for the whole system. It is also possible to control the system dependability by changing the structure of the system or by increasing reliability of its components. Examples of detailed processes for risk assessment procedures are presented in the Military Handbook: RELIABILITY PREDICTION OF ELECTRONIC EQUIPMENT [1].

The techniques presented above are based on the a priory/experimental knowledge available in the domain and highly depended on experience of engineers. To improve the process of failure analysis, the domain knowledge should be captured in the same manner. One of the best ways to represent knowledge is through hierarchical structures such as taxonomies.

The taxonomy presented in here is orthogonal to the taxonomy of faults presented by Avienzis [2]. In order to see a full ontology view, this taxonomies should be combined. The fault analysis and classification is strictly dependent on the chosen architectures and layer structuring.

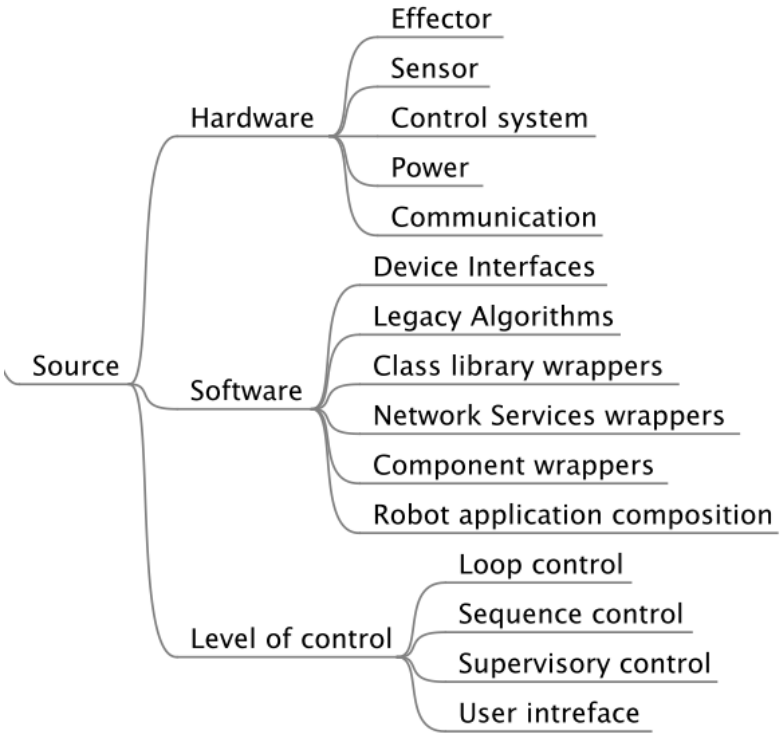


Figure 4.3: Taxonomy of faults

The sources of the failure can be analysed from three different prospectives: from hardware, software organisation, control organisation. From a hardware point of view there are 5 functional types of elements in a robot[7]:

Effectors are elements representing parts with which robots can affect the environment. All types of limbs, wheels or actuation mechanisms compose this category. Effectors include mechanical structures, actuation motors and wiring up to a controller.

Sensors are elements responsible for perception. The wiring of the sensor to controller is also included in this category, hence there is no difference from control system point view between failure in the sensor itself or wiring to it.

Control systems are elements responsible for computation of any kind. This type of elements cover micro controllers, computers and hardware that support the communication between parts of the control system on the robot.

Power class denote any source of energy that is available for the robot as well as all power related wiring.

Communication is parts of the system responsible for communication to environment.

Another view on source of failure is from software organisation. The software organisation for Robotics or Generic BRICS Robot Application Software Architecture was proposed by Gerhard K. Kraetzschmar [10]. This architecture is based on 6 layers of abstraction. The failures accordingly can originate form each level of abstraction.

Layers of control is another way to see a robotic system (fig. 4.4). Each of the layers can contain or be a source of the failure.

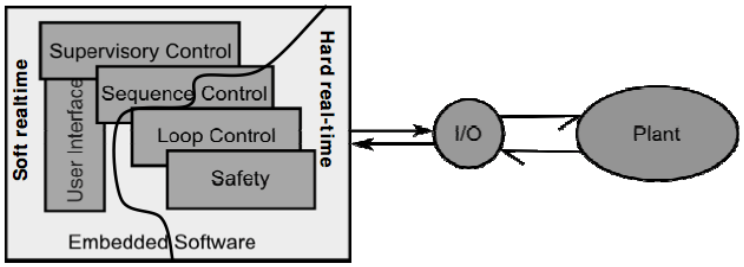


Figure 4.4: Layers of control

Loop Control - This level is concerned with dynamics of the system. Failures on this level result in the extensive flows and efforts, which could lead to destruction of the hardware or environment. High detection of speed or forces is essential for fault tolerance on this level; recovery procedures can be reduced to suspending the execution.

Sequence Control - This level is related to the kinematic status of the system. Failures on this level result in incorrect movements. This is similar to human slip type of failure [7]. The fault tolerance on this level is mostly a trade-off between speed and robustness of fault detection; the complexity of possible recovery procedures is limited by speed of the response.

Supervisory Control - This level n is related with a current context. Failures on this level result in incorrect actions. Failures on this level are similar to human mistakes [7]. The fault tolerance on this level is concentrated on robustness of fault detection, correct probability estimate and high isolation properties; the main recovery procedures operate on this level.

User Interface - This level is related to robot human interaction. Failures on this level result in incorrect directions to the robot. It is similar to a communication failure. The fault tolerance on this level is implemented trough command confirmation and assistance requests .

The fault forecasting techniques are aimed to justify the trust in the system ability to deliver a service. If the risk of failure is considered to be unacceptable the design procedure should be repeated until the system satisfies the requirements. Redesign of the system can be tackled by either fault removal or fault tolerance. Fault removal is the most direct contribution to dependability of the system, but in a robotic system it is not always possible due to an open environment. Fault tolerance contributes to dependability indirectly and some components are allowed to fail but the system overall will deliver a correct service. The concept of robust autonomy is an extension of fault tolerance; it allows some functionality to fail but the system will retain other functionalities and will deliver an acceptable service.

The result of preforming the fault forecasting analysis is a knowledge base that contains interrelation between the components, functionaries, events and failures. This knowledge base can be utilised in two different ways. As major guidelines for system redesign to support the robust autonomy concept and as part of automated fault tolerance algorithms.

5 Design guidelines

5.1 Design for robust autonomy

One of the main trends in robotics is to ensure component reuse and functionality encapsulation. From the field of secure and dependable computing this trend is supported by means of increasing dependability. The main emphasis created by such an approach is the development of dependable components. Although dependable components is at the basis of a dependable system, there are limitations.

Current levels of complexity for robotic components render impossible to maintain a consistent level of dependability through all system parts. Components with la ow dependability level and without special precautions will jeopardise the dependability of the whole system. It is often a requirement in robotics to construct a reliable system from less reliable components, which is a typical challenge of safety critical systems. Limited human interventions in a robot workflow make the ability of detecting and recovering from failure of a component on the system level an essential feature for a robot to ensure its robust autonomy.

Dependable components encapsulate certain type of functionalities making them reusable, however the fault tolerance is enclosed and fused with the component functionality. It is known that "in operation software systems often more then two thirds of the code is devoted to detecting and handling exceptions" [8]. Separation of the normal execution flow from the exception handling allows to create more reliable system, more understandable and more reusable systems.[15]

Dependable components are developed with the assumption that all the faults should be tolerated by means of the same component. However there is only a limited set of faults that could be tolerated under this assumption. Convoluted states of a robotic system created on different levels of abstraction create a set of complex abnormal situations. A recovery process required to let the system return into an error free state needs to be more involved and requires cooperative efforts of several components. Much like combination of components used to create a new system behaviour, a combination of the components should be used to create a complex recovery process.

These requirements (to tolerate failure of the component at system level, to separate the exception handling and to create more complex recovery process) present the idea of cooperative recovery. From a component point of view cooperative recovery is a process in which a system (component) is transferred to an error free state by a sequence of environmental states (actions of cooperative components). For example, software rejuvenation is a solution that allows to remove or prevent failure of the component through restart which initiated by other element of the system.

One of the concepts of component based development is hierarchical organisation of the system. This concept implies that combination of the components used to create new component with more complex behaviour. The hierarchical architecture of the components reinforces the concept of cooperative recovery. On each level of hierarchy, components capable of cooperative recovery will create a level of protection for the system.

The development of the system that would support cooperative recovery consists of the analysis of existing solutions and limitations, synthesis or adaptation of the solutions and evaluation. Here we will start with the analysis of existing solutions to fault tolerance problems. There are several facets of the fault tolerance process that need to be reviewed:

- architectural problem - how the components should be organised to support cooperative recovery.

- detection problem - how the failure could be detected/recognised outside of the component.
- recovery problem - how recovery process should be organised.

5.2 Architectural problem

The cooperative component recovery should be supported by a proper system architecture. A system architecture is what enables to generate a desired behaviour from a behaviour of the components. Cooperative recovery can be seen as an other type of desired behaviour. Addition of new parallel type of behaviours in the system will increase complexity of the system, eventually reducing system maintainability level and jeopardising dependability. Moreover a requirement to interrupt the normal workflow in case of the fault activation or to enter a special state demands a mechanism of switching the execution paths, which also contributes to system complexity. There are three topics that address the complexity issue: organisation of a single component, system partitioning and flow of execution control.

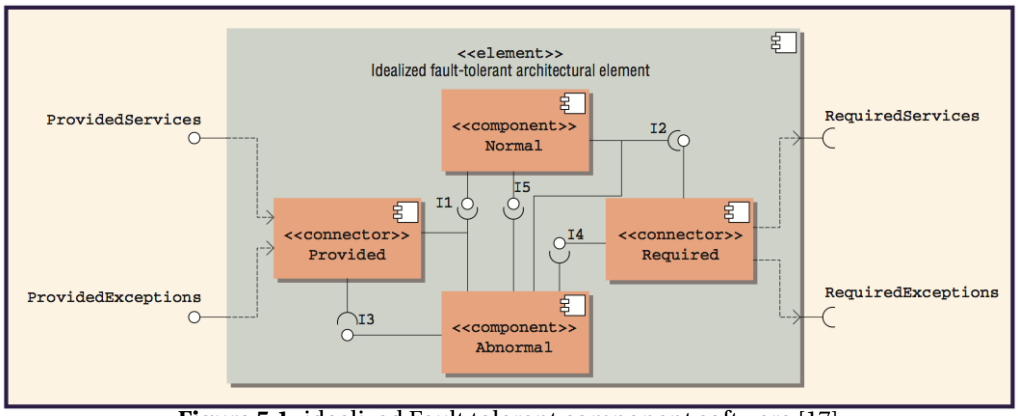


Figure 5.1: idealized Fault tolerant component software [17]

There are several requirements to the organisation of a single component. To ensure reusability and avoid introduction of the unnecessary complexity in the system component, it must have a clearly defined boundary. The behaviours of the components should be encapsulated as well as a fault handling mechanisms. This is achieved through defined interfaces for normal and abnormal workflow. For an abnormal work flow a component should receive the information about the failures in cooperative components and distribute information about faults it cannot handle. Internally the component should also support a separation of the exception handling from normal flow. An example of the component architecture that meets these requirements is presented on the figure 5.1. It was initially proposed by Brian Randell [25] and elaborated for software systems by Anderson, Lemo, Brito. The similar design was used in a hardware design by Marcel A. Groothuis [11] to create cooperative control systems working in parallel figure5.2 .

The iFTE has four types of external interfaces:

- ProvidedServices which is responsible for the provision of (fault-tolerant) services;
- SignalledExceptions which responsible for signalling either interface or failure exceptions;
- RequiredServices which specifies the required services;
- ReceivedExceptions which specifies the external exceptions that need to be handled.

The decomposition of the system in recoverable units is a trade-off between error confinement and development overhead. It is clear from figure 5.1 that each component responsible for the normal behaviour is supported by 3 other components. Although it is possible to create iFTE from every system component, it might not be beneficial. This is because development and

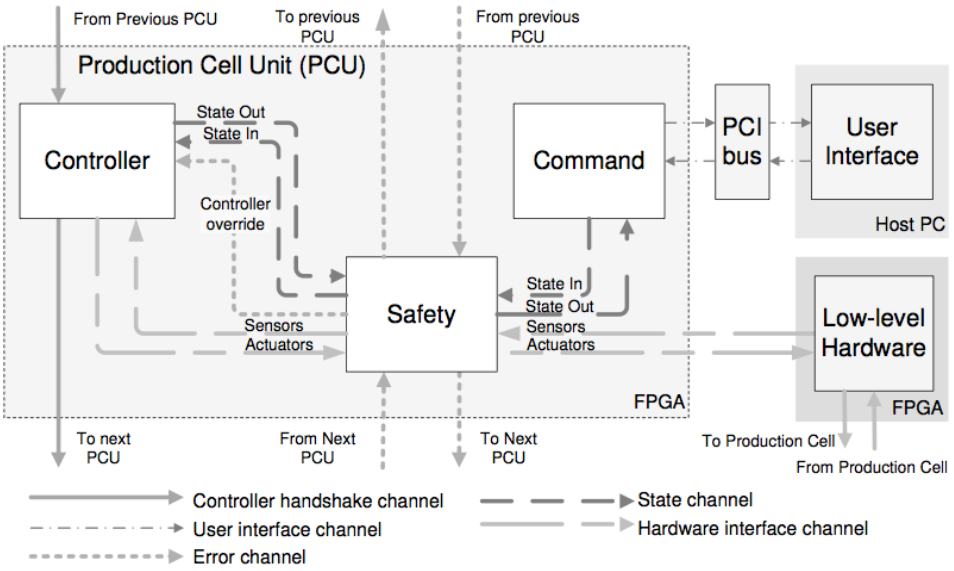


Figure 5.2: idealized Fault tolerant component hardware [11]

computation overhead created by additional components require additional resources possibly making the component unusable. On the other hand decomposition of the system on smaller elements allows better fault isolation, thus increasing fault tolerance. Implications of this trade-off in software are discussed by Hazan Sözer in his PhD thesis [26]. The proposed solution is partitioning the system on recoverable units(RU) based on minimisation of function calls between the recoverable units. The main limitation of the proposed solution is scalability, since the system should be analysed at run time and the decomposition is based on the solution of a set partitioning problem. Another part of the solution is to create another view of the system that will explicitly represent recoverable units and communications between them. This type of the view on system in the tool-chain(BRIDE) would allow to model exceptional behaviours, planning the recovery modes and the system decomposition. The process of system decomposition on recovery units should be investigated more to create guidelines that would address problems of scalability and a complex recovery process.

Control of the execution flow based on the element failures is similar to exception handling mechanisms developed in the software engineering. There are 5 groups of handling models[6]: termination, resumption, hybrid, retrying and nonlocal transfer. The control of the execution flow is rarely supported on the component level abstraction. Therefore further investigation is required to identify the best practice in that area. The set of guidelines for exception handling mechanisms in concurrent cooperative system are presented by Dusko Jovanovic [15].

In summary although it is possible to derive some initial guidelines for the system architecture that will support fault handling from the literature, more research is required in the area software decomposition and exception handling.

5.3 Detection problem

The detection of a component failure addresses the signalling of system failures. From the detectability point of view there are two types of failures signalled and un-signaled. The signalled failures are detected inside of the component and indicated for users. If such indication does not occur, the failure is called un-signaled. The development of a detection mechanism is directed at creating a components that will reduce the set of un-signaled failures in the system.

The main characteristics of such components were presented in Chapter 3. In here we will review the existing directions of the research in the area of detection systems.

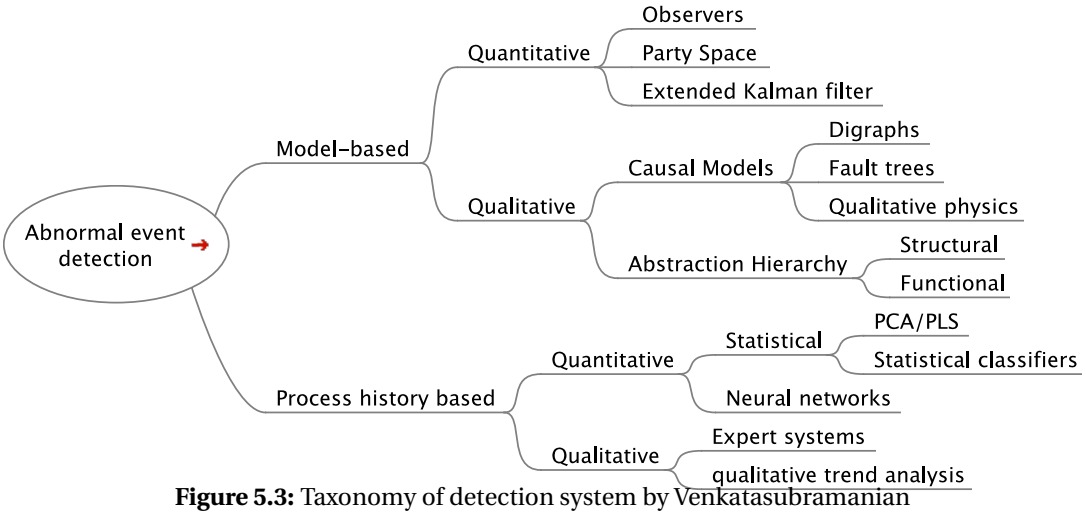


Figure 5.3: Taxonomy of detection system by Venkatasubramanian

The development of detection systems in software and hardware have been going on in parallel. However there are many similarities due to the results in both fields. The detection systems for hardware are based on process data and the laws of physics, much like the detection systems for software are based on execution data and the contracts and definitions of service. In both cases some priory knowledge is supplied to a detection system. Figure 5.3 presents the taxonomy of Venkatasubramanian[29] developed for a detection system used in chemical engineering. Similar work has been done by Isermann[14] and Carlson[7]. Based on the knowledge used in the system there are three main types of FDI systems, namely:

Model based system is created with assumption of a priory knowledge about the component. The components are presented as white boxes for which their internal execution can be described and monitored.

Process history based system is created with the assumption that the only accessible information is the process history.

Hybrid systems combine both types of information and are called hybrid systems.

Further distinction could be drawn between *quantitative* and *qualitative* methods. Quantitative systems are monitoring and comparing exact values creating precise views on how much the system deviates from the expected behaviour. The qualitative system classifies the behaviours.

Several comparative studies have been done in order to identify optimal detection algorithms for the task at hand. However a well developed theory is rarely elaborated into reusable practical solutions. The automatic detection components are developed from scratch for a specific system. The reuse of the detection algorithm is complicated due to specific knowledge included into the solution. To the best of our knowledge there is no open source/free frameworks that supports inclusion of detection blocks. Presence of configurable components for detection software and hardware malfunction could become a distinct feature of “BROCRE”.

Taxonomy of fault sources4.3 is a first step to identify similarities between faults in the robotics. Similar types of faults require similar types of detection systems. For example monitoring of the effector faults can be done using s bond-graph approach [18, 19]. This technique is particularly

promising because elements of the bond-graph structure are reusable and transformations of the bond-graph model into detection algorithms is a rigid procedure [22]. Although the clustering by sources of failure provides an initial clustering for the detection methods, the same source can generate different types of failures. There are several types of failures that have to be addressed by detection blocks. Figure 5.4 presents a taxonomy of failure types for the robotics domain:

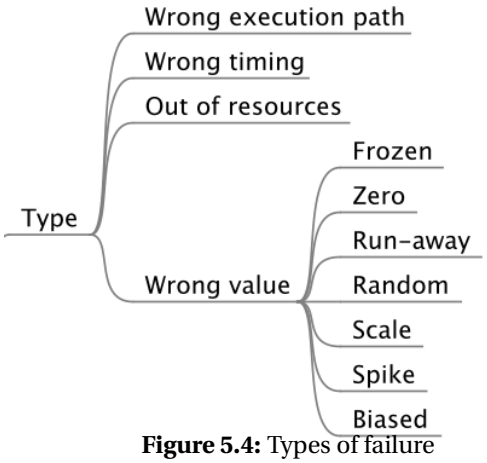


Figure 5.4: Types of failure

wrong value - the failure of the component results in wrong output value. The examples could be the sensor that give incorrect output, joint that does not respond to control system commands or incorrect computation results

wrong timing - the failure of the component results in delayed or a too early output. The examples could be deadlock, live lock or change in the system dynamics

out of resource - the failure of the component results in the request of resources that does not exist or is unusual for this component. The examples could be the attempt to overload CPU, memory failures or power drain.

wrong execution path - the failure of the component results in initiating an incorrect workflow. The examples could be a incorrect service request.

The detection system should be structured in the same way as it was presented in the section 5.2. Each detection element is responsible for one recoverable unit(RU). Increasing complexity of the RU will increase the complexity of the detection algorithm and reduce quality of isolation. Therefore it is important to take in the consideration detection process during system decomposition. The monitoring approach is selected based on the taxonomies presented on figures5.4, 4.3 and results of fault forecasting for this component. This will support reusability of the detection algorithms together with RU.

5.4 Recovery problem

From the point of view of dependable computing, a recovery process is a transformation of the system state that contains errors or faults into a state without detected errors and without faults that can be activated again. From this definition it follows that there are two trends in the recovery process: error handling and fault handling. Fault handling is a process that prevents faults from further activation. Error handling is a process of eliminating errors from the system state.

The ability of the robot to respond to exceptional situations to a large extent consists of recovery processes. Recovery algorithms are aimed to transfer the system into a correct state after the fault has been activated in 3 different ways: compensation, forward recovery, backward recovery.

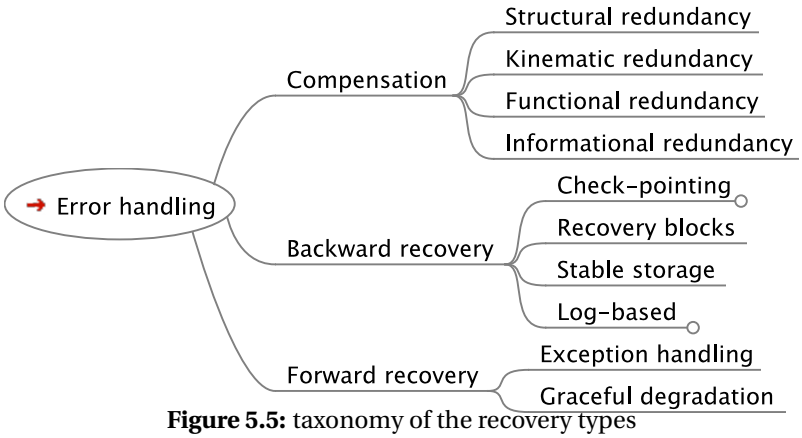


Figure 5.5: taxonomy of the recovery types

Compensation is a online replacement of the failed component with a redundant one. Such a system does not lose any functionality in case of a component failure. There are 2 types of redundancy possible, replication of the element or replication of its function those named accordingly: structural redundancy, functional redundancy.

Structural redundancy indicates a complete replication of system parts. It is used to protect the system from failures in the replicated component, with assumption that the behaviour of the component has been designed correctly, but there is a probability of failure of subcomponents. Replication is often used to increase reliability of safety critical electromechanical systems.

Functional redundancy represents the replication of the function, but not with the same design. This approach provides a system with design diversity. It protects form design mistakes as well as offers component replication. The functional redundancy can be categorised depending on the implementation. We will call the complete replication of the function of a specially designed component N-version design (after N-version programming in software engineering). If combination of several elements of the system can be used to replace a function of the failed components it is named a complimentary part. Unitising a prior knowledge about system behaviour to compensate the failure is called informational redundancy.

Forward recovery is a process of masking the failure of the component. The system interrupts the normal workflow and attempts to provide the service by an alternative execution process. Depending on the system ability to achieve the goal there are two types of forward recovery: exception handling and graceful degradation. The exception handling is a recovery process that allows a system to deliver a *correct* service by means of the alternative execution process. It is most typically applied to mask transient faults.The graceful degradation is a recovery process that allows system to deliver an *acceptable* service by means of the alternative execution process. This type of recovery process is applied to mask permanent faults.

Backward recovery is a process of transferring system to a known error free state. The system constantly records its states, these records are used to back step to an error free state. There are two distinct types of backward recovery: recovery blocks, check-pointing. Recovery blocks contain three elements the functionality, the checking mechanism and rollback procedure. If the functionality of the recovery block fails the checking mechanism detects the failure and initiate rollback procedure. In check-pointing system states are recorded to be replayed in case

of system failure. The failure of the component is then due to the inability to transfer it to an initial position after that the component can enter a last known error free state(check-point) before continuing execution.

6 Use cases

6.1 Robot drive failure – Omni wheels

TBW

6.2 Robot arm failure – Reducing working envelop

TBW

6.3 Path planner failure – N-version programming

TBW

7 Conclusion

TBW

Bibliography

[1] *MILITARY HANDBOOK RELIABILITY PREDICTION OF ELECTRONIC EQUIPMENT*. USA Department of defense, 1991.

[2] Algirdas Aviezienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, volume 1. IEEE Computer Society, January 2004.

[3] B. S Blanchard and W. J Fabrycky. Systems engineering and analysis. 2006.

[4] J. Breed, K. Walyus, and J. Fox. Enabling advanced automation in spacecraft operations with the spacecraft emergency response system. In *AAAI Technical Report SS-01-06*. The AAAI Press, Menlo Park, California, 2001.

[5] BRICS. Description of work. Technical report, SEVENTH FRAMEWORK PROGRAMME, 2008.

[6] P. A Buhr and W. Y.R Mok. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820Ð836, 2000.

[7] J. Carlson, R. R Murphy, and A. Nelson. Follow-up analysis of mobile robot failures. In *IEEE International Conference on Robotics and Automation*, volume 5, page 4987Ð4994, 2004.

[8] F. Cristian. Exception handling and tolerance of software faults. *Software Fault Tolerance*, 3:81–107, 1995.

[9] Stan Franklin. An agent architecture potentially capable of robust autonomy. In *AAAI Technical Report SS-01-06*. The AAAI Press, Menlo Park, California, 2001.

[10] Jan Paulus Nico Hochgeschwender Michael Reckhaus Gerhard K. Kraetzschmar, Azamat Shakhimardanov. Deliverable d-2.2: Specifications of architectures, modules, modularity, and interfaces for the brocre software platform and robot control architecture workbench. Technical report, Bonn-Rhine-Sieg University, 2010.

[11] M. A Groothuis, J. van Zuijlen, and J. Broenink. FPGA based control of a production cell system. *Communicating Process Architectures 2008*, 66:135Ð148, 2008.

[12] Michael Hofbaur, Johannes Köb, Gerald Steinbauer, and Franz Wotawa. Improving robustness of mobile robots using model-based reasoning. *Journal of Intelligent and Robotic Systems*, 48(1):37–54, January 2007.

[13] A. R Hudson and L. H Reeker. Standardizing measurements of autonomy in the artificially intelligent. In *Proceedings of the 2007 Workshop on Performance Metrics for Intelligent Systems*, pages 70–75, 2007.

[14] R. Isermann. *Fault Diagnosis Systems – From fault detection to fault tolerance*. Springer Verlag, 2006.

[15] D. S Jovanovic. *Designing dependable process-oriented software, a CSP approach*. PhD thesis, University of Twente, Enschede, The Netherlands, 2006.

[16] David Kortenkamp. The roles of machine learning in robust autonomous systems. In *AAAI Technical Report SS-01-06*. The AAAI Press, Menlo Park, California, 2001.

[17] R. De Lemos, P. A de Castro Guerra, and C. M.F Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE software*, page 80Ð87, 2006.

[18] Chang Boon Low, Danwei Wang, S. Arogeti, and Ming Luo. Quantitative hybrid bond Graph-Based fault detection and isolation. *Automation Science and Engineering, IEEE Transactions on*, 7(3):558–569, 2010.

[19] Chang Boon Low, Danwei Wang, S. Arogeti, and Jing Bing Zhang. Causality assignment and model approximation for hybrid bond graph: Fault diagnosis perspectives. *Automation Science and Engineering, IEEE Transactions on*, 7(3):570–580, 2010.

[20] B. Lussier, A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M.O. Killijian, and D. Powell. Fault tolerance in autonomous systems: How and how much? In *Proceedings of the 4th IARP/IEEE-RAS/EURON Joint Workshop on Technical Challenge for Dependable Robots in Human Environments, Nagoya, Japan*, 2005.

[21] R.A. Maxion and R.T. Olszewski. Eliminating exception handling errors with dependability cases: a comparative, empirical study. *Software Engineering, IEEE Transactions on*, 26(9):888–906, 2000.

[22] A. Mukherjee. *Bond graph in modeling, simulation and fault identification*. CRC Press, 2006.

[23] USA Department of defense. *MILITARY STANDARD PROCEDURES FOR PERFORMING A FAILURE MODE, EFFECTS AND CRlTlCALliV ANALYSIS*. USA Department of defense, 1980.

[24] R. Parasuraman, T. B Sheridan, and C. D Wickens. A model for types and levels of human interaction with automation. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 30(3):286–297, 2000.

[25] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, Los Angeles, California, 1975. ACM.

[26] H. Sözer. *Architecting fault-tolerant software systems*. PhD thesis, University of Twente, Enschede, The Netherlands, 2009.

[27] R. F Stengel. Intelligent Failure-Tolerant control. *IEEE Control Systems Magazine*, 1991.

[28] Donald E. Swihart. Automatic ground collision avoidance system (auto gcas). In *ICS'09: Proceedings of the 13th WSEAS international conference on Systems*, pages 429–433, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

[29] Venkat Venkatasubramanian, Raghunathan Rengaswamy, Kewen Yin, and Surya N. Kavuri. A review of process fault detection and diagnosis. part i: Quantitative model-based methods. *Computers & Chemical Engineering*, 27(3):293–311, March 2003.

[30] Vandi Verma, Reid Simmons, Dan Clancy, and Richard Dearden. An algorithm for Non-Parametric fault identification. In *AAAI Technical Report SS-01-06*. The AAAI Press, Menlo Park, California, 2001.