

353.663

TECHNICAL REPORT

SER. ELECTRICAL ENGINEERING

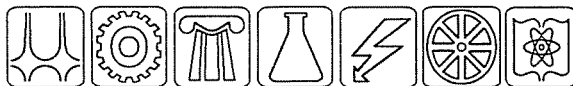
THE APPLICATION OF INTELLIGENT DIAGNOSTIC CENTERS IN THE IMPLEMENTATION OF DISTRIBUTED FAULT-TOLERANT SYSTEMS

K. Tilly, T. Dobrowiecki, I. Majzik, A. Somogyi, A. Várkonyi-Kóczy, I. Kiss,
Z. Dankó, Cs. Tóth, B. Vadász, T. Zsemlye

Department of Measurement and Instrumentation Engineering
Technical University of Budapest

No. TUB-TR-94-EE09

BUDAPEST, March 25, 1994



TECHNICAL UNIVERSITY OF BUDAPEST

TECHNICAL REPORT

SER. ELECTRICAL ENGINEERING

THE APPLICATION OF INTELLIGENT DIAGNOSTIC CENTERS IN THE IMPLEMENTATION OF DISTRIBUTED FAULT-TOLERANT SYSTEMS

K. Tilly, T. Dobrowiecki, I. Majzik, A. Somogyi, A. Várkonyi-Kóczy, I. Kiss,
Z. Dankó, Cs. Tóth, B. Vadász, T. Zsemlye

BME KÖZPONTI KÖNYVTÁRA



K 079 388

Department of Measurement and Instrumentation Engineering
Technical University of Budapest

No. TUB-TR-94-EE09

BUDAPEST, March 25, 1994



TECHNICAL UNIVERSITY OF BUDAPEST

353.663

© TU Budapest, Department of Measurement and Instrumentation Engineering, 1994.

ISSN 1216-3015 (Technical Report, Ser. Electrical Engineering, on paper)
ISBN 963-421-523-8 (on paper)

ISSN 1216-9226 (Technical Report, Ser. Electrical Engineering, on microfiche)
ISBN 963-421-524-6 (on microfiche)



Editor of the Technical Report Series of the TUB

G. Stépán
Periodica Polytechnica Secretariat,
H-1521 Budapest, Hungary
Telefax: + 36-1 166-6808

Produced in booklet form by the Printing Office of the TUB

Microfiche: Central Library of TUB

Requests for Technical Reports or list of published Technical Reports:

Central Library of TUB
Department of International Exchange
H-1521 Budapest, Hungary
Telefax: + 36-1 181-2753, telex: 22-5931 muegy h

THE APPLICATION OF INTELLIGENT DIAGNOSTIC CENTERS IN THE IMPLEMENTATION OF DISTRIBUTED FAULT-TOLERANT SYSTEMS¹

K. Tilly, T. Dobrowiecki, I. Majzik, A. Somogyi, A. Várkonyi-Kóczy, I. Kiss,
Z. Dankó, Cs. Tóth, B. Vadász, T. Zsemlye

Department of Measurement and Instrumentation Engineering
Faculty of Electrical Engineering and Computer Science
Technical University of Budapest
1521. Budapest, Hungary
Phone & fax: +(36-1) 166-4938
e-mail: tilly@mmt.bme.hu

Abstract:

The basic terminology and fundamental methods of design and implementation of fault tolerant systems have been evolving up until today, though some important questions are still open. Traditional solutions are intended for use at design time and they generally capture system information at a very low (hardware or machine instruction) level. On the other hand the management of complex information systems containing many (perhaps many thousands) of autonomous components seems to be hardly solvable by this kind of solutions.

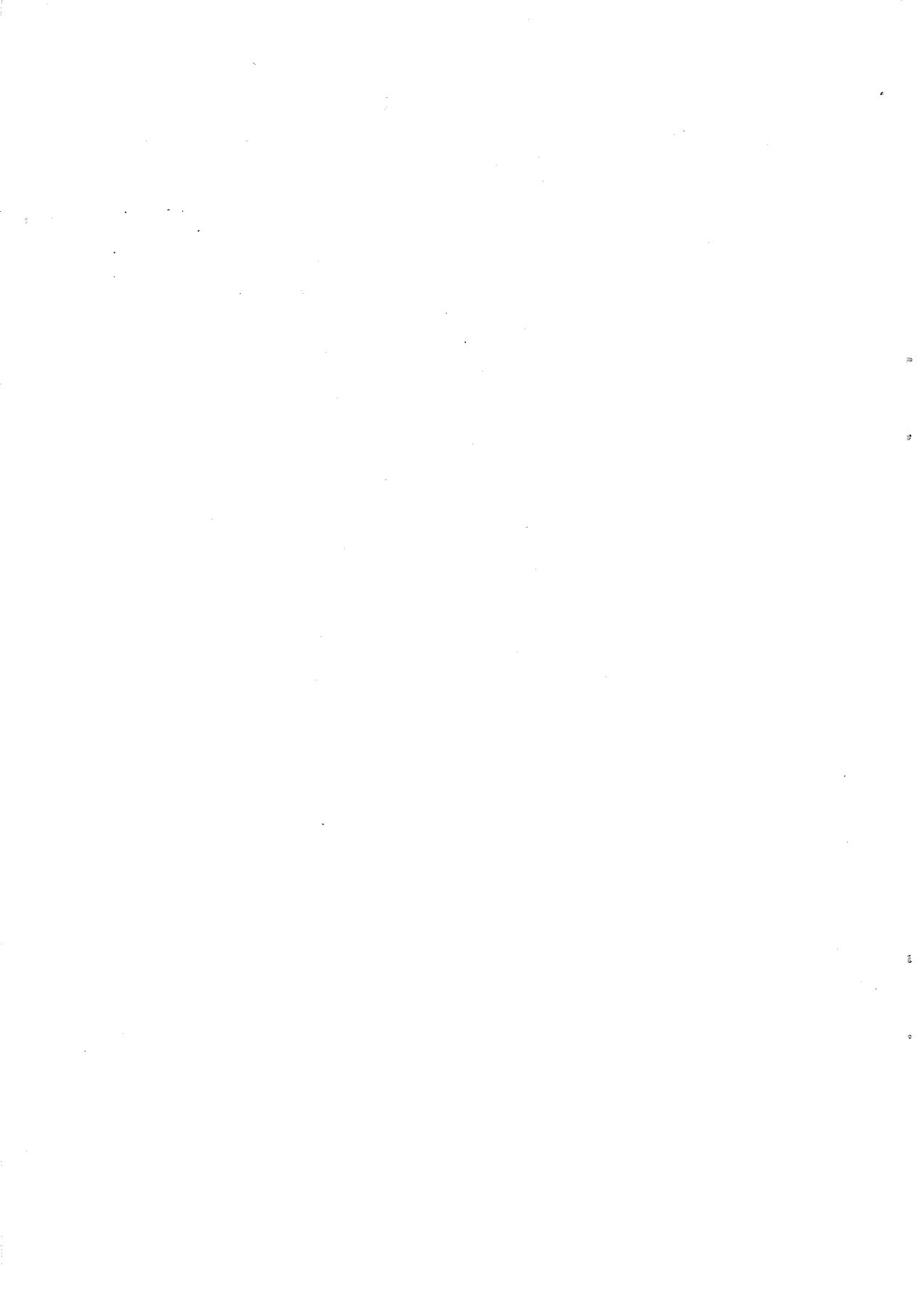
This report presents the first results of a research project which intends to work out a new philosophy for the implementation of large scale, distributed fault tolerant systems. The basic idea of the project is to add relatively small hardware and software overhead to existing systems in the form of intelligent diagnostic centers, which act like high level, generalized watch-dog processors.

After a summary of the basic terminology and the most important state-of-the-art solutions used in fault tolerant systems, general properties of some important distributed system classes (multicomputer architectures, computer networks and industrial technologies) are analyzed and the common properties are outlined. Based upon these common properties a modelling method and an interpretation algorithm is described for representing distributed systems and to achieve fault tolerant operation with diagnostic centers. The basic idea of the approach is to view the distributed system as a hierarchical decomposition of tools for solving specific tasks and to use these tools in every possible way to solve those tasks even if some tools fail. This leads to a certain kind of reconfiguration and failure recovery. This methodology is similar to the technique of a human problem solver, who is able to use its resources in multiple ways.

The properties and interpretation of the model are shown by a simple example.

Keywords: fault tolerant system, distributed system, hierarchical decomposition, fault masking, reconfiguration, fault recovery, artificial intelligence.

¹ This research was sponsored by the Hungarian Fund of Scientific Research (OTKA), under contract no. F007414.



CONTENTS

1 INTRODUCTION	6
2 BASIC CHARACTERISTICS OF FAULT TOLERANT SYSTEMS	8
2.1 Terminology	8
2.2 Classification of impairments to dependability	9
2.2.1 Fault classes	9
2.2.2 Error classes	10
2.2.3 Failure classes	10
2.3 Fault models	11
2.4 Fault tolerant structures	11
2.4.1 Hardware based redundancy	12
2.4.2 Information based redundancy	13
2.4.3 Time based redundancy	14
2.4.4 Software based redundancy	14
2.5 The operation of a fault-tolerant system	15
2.5.1 Error detection	15
2.5.2 Fault treatment	16
2.5.3 Error processing	16
3 A SURVEY OF PRACTICAL FAULT-TOLERANT SYSTEMS AND TECHNIQUES	18
3.1 Concurrent error-detection mechanisms	18
3.1.1 Modular redundancy	18
3.1.2 Master-checker mode	19
3.1.3 Watchdog processors	19
3.1.4 Evaluation of the concurrent error detection methods	21
3.2 Fault diagnosis	22
3.2.1 Hardware level solutions	22
3.2.2 High level methods	22
3.3 System recovery	25
3.3.1 Error recovery for uniprocessor applications	25
3.3.2 Error recovery for multiprocessor applications	25
3.3.3 Error recovery for distributed real-time applications	26
4 PROBLEM FORMULATION AND THE ELEMENTS OF A REPRESENTATION MODEL	28
4.1 The Basic Problem	29
4.2 General properties of a diagnostic center	29
4.3 General characteristics of the model	30
4.4 Elements of a plane	32
4.5 Structural properties of the model	34
4.6 The requirement structure	34
4.7 The service structure	36
4.8 The resource structure	37
4.9 The fault model	37
4.10 The algorithm of the diagnostic center	37

5 AN EXAMPLE FOR THE CONSTRUCTION AND USE OF THE MODEL:	
THE WORLD OF EGG BOILING	40
5.1 Elements of the world	40
5.2 The model	41
5.3 How the model is used?	48
6 RELATIONS TO PREVIOUS WORK	51
7 CONCLUSIONS AND FURTHER WORK	52
REFERENCES	54

TECHNICAL REPORTS OF THE
TECHNICAL UNIVERSITY OF BUDAPEST

SER. ELECTRICAL ENGINEERING:

- TUB-TR-93-EE01 Tamás Dabóczy: Deconvolution of Noisy Transient Waveforms, Jan. 1993
- TUB-TR-93-EE02 Molnár Sándor: Congestion Control and Queuing Models in ATM Networks, Feb. 1993
- TUB-TR-93-EE03 József Zoltai: New Feedback Structure for Monolithic Integrated Instrumentation Amplifiers, March 1993
- TUB-TR-93-EE04 Ákos Jobbágy: Centre Estimation in Marker Based Motion Analysis, April 1993
- TUB-TR-93-EE05 Thaier N. Mohammad: IIR Adaptive Filtering Methods and Algorithms, October 1992
- TUB-TR-93-EE06 Abdulrahim Abdulwahab: Application Problems of Order Statistics Filters, May 1993
- TUB-TR-93-EE07 Reiner Thomä: Spectral Correlation Measurement July 1993
- TUB-TR-93-EE08 A. Iványi: R-Functions in Electromagnetic, December 1993
- TUB-TR-94-EE09 K. Tilly et al.: The Application of Intelligent Diagnostic Centers in the Implementation of Distributed Fault-Tolerant Systems
March 1994

1 INTRODUCTION

The decreasing hardware costs, the increasing proficiency of constructing and programming sophisticated computer systems made it possible to solve complicated problems better and more efficiently than ever before. That is why nowadays the use of complex distributed computer systems is more and more widespread in applications like wide and local area computer networks, multicomputer architectures of hundreds or thousands of processors or different kinds of automated industrial technologies.

Although in such cases reliability is often very important (e.g. in banking networks), results currently available in the literature of fault tolerant systems give no proper answer for several important questions.

Fault tolerant systems created until today are systems with *hardware level fault tolerance*, using solutions like the multiplication of system components, voting, self testing or watch dog processors. The aim of hardware level fault tolerance is to capture faults at the lowest possible level to assure very fast reaction times (e.g. signalling of faults or reconfiguration). On the other hand higher level faults in the system are hard to detect and correct, because at the hardware level it is impossible to create higher level system models. Another important problem is, that the granularity of information used by hardware level fault tolerance is too fine, so the solutions belonging to this class always require an enormous overhead. Hardware level fault tolerance can be considered as a "*bottom-up view*" of system operation.

A major problem is that in large scale distributed systems these traditional fault tolerant solutions cannot be applied because of their complexity and relatively high costs. Another problem is that traditional methods are intended for use in *design time*, though in many cases it would be important to increase with minimal costs the reliability of an existing system, which was *not originally designed* using fault tolerant techniques.

It is possible to extend hardware level fault tolerant computer systems in two directions:

- If the *level* of fault tolerance is considered, we can speak about *system level fault tolerance*, which can be achieved in distributed systems by higher level tools, like system level self test and diagnosis. System level fault tolerance is established upon a "*top-down view*" of the system.
- As fault tolerant systems are not necessarily computers, we can *generalize* the *type* of fault tolerant systems to any complex structures (like industrial technologies), where the ability of fault tolerance has significance. This way the basic notions of fault tolerance can be used, though at a higher abstraction level and with often different semantics.

In this report we will characterize a model of achieving *general system level fault tolerance*. The basic problem is to find solutions, which can improve overall system reliability by adding minimal hardware redundancy and some additional software components. In the approach presented in this report, this task is solved by *intelligent diagnostic centers*, which observe the operation of different system components, compare the incoming information with certain system models and in the case of malfunctions maintain the original services of the system by choosing different methods of functioning.

The operation of a diagnostic center is similar to a human problem solver, who has a model of the problem and has a set of tools, which he can use to solve the problem in certain predefined ways. The complete solution requires the accomplishment of a sequence of

operations. Since any tool may be able to perform different kinds of operations, the whole problem can be solved in many different ways, some of which are optimal, while others may be worse, though still lead to a solution. (E.g. it seems to be the best to drive screws with a socket wrench, though you may apply pliers if you have no socket wrench of the appropriate size.) In this sense we can state that human problem solving is *fault tolerant*, since humans may still be able to solve problems even if certain preconditions which would normally be essential to solve the given problem are missing. Such operation is based upon redundancy which is implicitly or explicitly always present in distributed systems.

To implement human-like actions in a computer system and the need to cope with the algorithmic and representational complexity of describing and interpreting models of large scale systems lead to the application of artificial intelligence techniques. This gives us a starting point, which is significantly different from any previous solutions developed for fault tolerant systems. The model uses a top-down hierarchical decomposition of the system, which makes it possible to assure algorithmic and representational efficiency. The practical viability of the concepts is guaranteed by the wide range of experiences with intelligent monitoring and diagnostic systems.

The results presented in this report are the first steps in a longer research. The main goal of this first stage, which was important from the point of further work, though it proved to be surprisingly difficult, was to find an appropriate set of primitives, terminology and basic run-time algorithms, which can be used to set up a framework for achieving fault tolerant operation in distributed systems by means of intelligent diagnostic centers.

To show the state of the art and to demonstrate the basic differences between our approach and traditional solutions, chapters 2 and 3 give a summary of definitions and most important techniques currently applied in fault tolerant systems.

In chapter 4 based upon the common properties of different kinds of distributed systems a general, high level model is presented for describing and achieving fault tolerant operation at the system level.

Chapter 5 introduces the application of the proposed model to a simple practical example and the operation of the model interpretation algorithms are also shown.

Chapter 6 analyzes the relations between existing methods and systems and the solutions proposed in this report.

Chapter 7 refers to some obvious powerful possibilities of implementing fault tolerant systems using the proposed modelling methods and characterizes the directions of further work.

2 BASIC CHARACTERISTICS OF FAULT TOLERANT SYSTEMS

In this chapter we give a basis for further studies of fault tolerant systems and their applications by introducing the traditional terminology and basic notions of fault tolerant systems. Fundamental methods to achieve fault tolerance will also be shown.

2.1 Terminology

Dependability is the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers (Laprie, 1985). Dependability is the most basic and general requirement in case of computers used in applications where the costs of computer malfunction can be unacceptably high.

Dependability can be achieved through *fault prevention* and *fault tolerance*.

In case of fault prevention the possibility of fault occurrence is prevented, *before* the system is placed into service.

The ability of a computer system to perform unaffected service in spite of existing faults is called *fault tolerance*.

Any deviation from the correct service (given in the specification) is called *failure*. A failure is always due to an erroneous system state, which means that in the given system state an *error* is present, which is liable to lead to a failure. The presumed or real cause of an error is called *fault*.

Using these definitions fault tolerance means preventing errors which would lead to system failure.

To give a better overview of these terms three level of processing can be defined in any information processing systems (Aviziensis, 1982):

- physical level;
- information level;
- service level.

The *physical level* includes all hardware and software components (wires, ICs, programs etc.). Every system component, which exists before the system is placed into service, is part of the physical level. Any deviation from the specified normal state at the physical level will be called a fault. (E.g. short-circuits, wire-breaks, endless loops or illegal instruction codes in programs.)

The *information level* contains every parameter of the system, which can be liable for the current system state. On this level all deviations from the correct functioning of the system are called errors. An error can always be traced back to known or unknown fault(s).

E.g. a stuck-at-0 fault on a gate input does not cause an error until the faulty input is driven by a logical 0 value. Because of the stuck-at-0 fault the output of the gate remains 0, even if the faulty input is driven by a logical 1. This is called an error. When a program contains an endless loop, it is a fault. If the execution of the program enters this loop, it causes a faulty system state, which is called an error.

The *service level* includes all the services performed by the correctly functioning system. Any deviation at this level is called a failure. A failure is the spread out of error(s).

E.g. if the program execution enters an endless loop (this is an error), and therefore the system will not perform

a service in time, it is called a failure. A faulty result among the services is also called a failure.

Based on these three level modelling of systems, there are three possibilities to secure correct service:

- *fault avoidance*: to prevent the formation of faults at the physical level;
- *fault masking*: to prevent existing faults from shaping errors at the information level;
- *fault tolerance*: to prevent errors to become failures at the service level of the system.

Since the boundary between these three levels of information processing systems cannot be defined precisely, the terms fault avoidance, fault masking and fault tolerance accommodate significant overlapping.

E.g. a stuck-at-0 fault on a gate input causes an error if the input is driven to 1. If the service of the module containing the faulty gate remains unaffected, it is called fault tolerance at the gate-level model of this module, but it is called fault masking at the module-level model of the whole system, because the stuck-at fault does not cause any error at the outputs of any module.

According to its general usage *fault tolerance* means all the *intentionally built in* hardware or software features, which secure the correct functioning of the system even if one or more predefined faults are present. Fault masking includes all *design* strategies to construct dependable systems where the majority of faults are prevented to lead to an error (e.g. by filling the unused part of the interrupt-vector-table with RETURN instructions). And finally *fault avoidance* means selecting the best components and securing the best environment for the system in order to prevent faults.

2.2 Classification of impairments to dependability

2.2.1 Fault classes

The classification of faults can be done according several aspects, although the *nature*, the *origin* and the *persistence* of faults are the most important of them (Laprie, 1990).

The *nature of a fault* can be either *accidental* or *intentional*.

The source of an *accidental fault* cannot be traced back to intentional human action, while *intentional faults* are due to intentional human action, like in the case of fault injection for testing and diagnostic purposes (Fujiwara, 1985).

The *origin* of faults can be decomposed according to three main categories:

- phenomenological causes:
 - physical faults;
 - human-made faults.
- system boundaries:
 - internal faults;
 - external faults.
- phase of creation with respect to the system's life:
 - design faults;
 - operational faults.

The *persistence* of faults can be *permanent* or *temporary*.

Temporary faults are often called transient faults, originating from the physical environment. However temporary faults can also be internal faults, termed intermittent or

non-recurrent faults.

If all the combination of fault classes according to the three main viewpoints discussed above were possible, there would be thirty-two different classes. In fact, the number of likely combinations is restricted: only about ten combination are commonly used to describe the properties of a fault.

2.2.2 Error classes

In the case of errors the main question is, whether they will lead to a failure or not. It depends on three major factors:

- *The existing redundancy*, contained in the system. Redundancy can be either *intentional*, which provides the fault tolerance of the system, or *unintentional*, which may have the same but unexpected result as intentional redundancy. This kind of redundancy is not always helpful, since it may prevent some parts of the system from being testable.
- *The system activity*: an error may be overridden before creating damage.
- *The definition of failure* from the user's viewpoint.

2.2.3 Failure classes

The ways a system can fail can be characterized from three aspects: *domain*, *perception* by the system users, and *consequences* to the environment.

The *domain* aspect includes:

- *value failures*, when the value of the delivered service does not comply with the specification;
- *timing failure*, when the timing of the service delivery does not comply with the specification. (The service is fulfilled too early, too late or the service stops etc.).

The *failure perception* aspect includes:

- *consistent failures*, when all users of the system have the same perception of the failure;
- *inconsistent failures*, when the users of the system have different perception of the failure.

The *consequences of the failure* aspect includes:

- *benign failures*, when the consequences of the failure are of the same order of magnitude in term of cost² as the benefit provided by the service delivery in the absence of failures.
- *catastrophic failures*, when the consequences of the failure are incommensurably greater than the benefit provided by the service delivery in the absence of failures.

A system whose failures can only be benign failures is often called *fail-safe system*.

² The notion of cost is considered in a general fashion here and can be represented in the form of e.g. cost functions. Otherwise the failure of an intensive care equipment which leads to the death of a patient could be considered benign, though in fact it is a catastrophic failure.

2.3 Fault models

Although in practice faults can be transient in duration and indeterminate in value, it would be extremely difficult to analyze digital systems if faults were assumed to have these characteristics. Fault models attempt therefore to represent the types of faults that can occur within the system.

Fault models are always according to the level of system model. Systems are generally modelled at the *functional*, *logical* or *physical* level.

At the *physical level* the "physical" behaviour of the system is considered, i.e. the transistor level of an IC. This kind of modelling assures wide possibilities for describing system behaviour. The required computational capacity will however exponentially rise with system complexity. Therefore the physical level modelling cannot be used in the case of complex digital systems.

The *logical level* modelling deals with the internal logical lines of the system. The correct function and the possible faults of the components are described by logical relations. The disadvantage of this approach is that the high scale integrated components (because of the lack of information and computational capacity) cannot be described by logical relations based on their internal logic lines.

The *functional level* modelling traditionally deals with the logical function performed by the component. The component can be an IC itself, can be a module or a whole system as well.

The different levels of system modelling require different fault models. Physical system models generally use the transistor level model for describing the system. In this case transistor stuck-at fault models are used as basic building blocks for describing faults. Logical level system models use the logical stuck-at fault model. In the case of functional system models the absence or unwanted execution of basic function(s) is considered.

Based on these classes the functional model of most typical components of a computer had been developed, such as CPU, memory or buses. The fault model of these modules and of the whole computer-module can be evaluated either from these models, or from the general functional system model.

2.4 Fault tolerant structures

If the dependability of a computer-system does not satisfy the requirements in spite of the applied fault-avoidance techniques, the application of fault-tolerance techniques is necessary. All kinds of fault-tolerance methods are based on redundancy.

We can distinguish between *hardware level redundancy*, *software level redundancy*, *information redundancy* or *time redundancy*.

It should be noticed, that there is no fault-tolerance method where only a single type of redundancy appears. The viewpoints for this classification in case of a fault-tolerant structure is however the origin of fault-tolerance and its projection onto other classes.

2.4.1 Hardware based redundancy

Hardware redundancy always secures hardware level fault-tolerance, protecting a specified module (or group of modules) against one or more predefined faults. If this kind of fault-tolerance works effectively, it will cause that in spite of the presence of some predefined faults, no error will appear at the logical surface of this module. It will secure fault masking at this level, although fault-tolerance is performed from the service level point of view of this module.

Hardware techniques for realising fault-tolerance in digital systems are commonly used. The solutions for reaching fault-tolerance in digital systems can be classified into three groups:

- the *voter scheme*;
- the *master-checker scheme*;
- the *watch-dog scheme*.

All three classes can be interpreted at a different level of system modules, such as gate-level, functional-level (memory, CPU etc.), or computer-board level. The three basic classes will be introduced at computer-board level.

The voter scheme

This solution secures the failure-free operation through operating at least 3 identical modules parallel (*triple modular redundancy: TMR*). The output of the system will be determined by a majority voter. The voter itself should be failure-safe i.e. it should contain further fault-tolerance techniques (e.g. it can also be repeated at least three times).

Comparing the results of parallel operating modules leads to two major problems:

- *Synchronisation problem*: If the results of the parallel operating modules are not synchronised, the voter will compare an inconsistent set of results and thus an invalid decision will be made.
- *Deviation between the correct results*: The results of different modules does not always completely agree, not even in a fault-free environment.

E.g. a pair of sensors, or A/D converters or any kind of interface with analog signals will not surely result the same digital value, not even in a fault-free case.

This problem can be overcome with defining tolerance intervals for the deviations between the results, or with the mid-values technique, where in case of a TMR the average value of the other two will be selected.

The basic concept of a voter scheme is a kind of passive redundancy, since there is no further action performed by the system to achieve fault tolerance. This scheme can be part of system level redundancy, where the faulty module reported by the voter will be replaced by the system itself, activating cold redundancy instead of the faulty module.

Practical applications limit the number of modules, thus the amount of redundancy that can be employed. Power consumption, weight, cost, size are normally not negligible design aspects.

The master-checker scheme

In the case of a master-checker pair, two identical computer-boards are operating parallel within a module. The output of the module is taken directly from one of the computer-boards called the master. The other computer-board, the checker will compare its own results with the results of the master. In the case of discrepancy an error message is generated. Failure free operation is secured by other parts of the system.

The master-checker scheme itself cannot secure fault-tolerance. It is a technique used for detecting errors. The masking of the error is in this case done by other modules outside the erroneous master-checker pair.

This scheme is a kind of active redundancy, since in case of erroneous results error detection is done, and additional system activities should take place to secure failure-free services.

The watch-dog scheme

Further simplifications can be made in the master-checker error detection scheme by reducing the computational capacity of the checker-board. The checker in this case will not compare all details sent out by the master, just validation tests will take place.

The "watchdog-processor" (WP) means not only the classical watchdog-timer functions intensively used in bus systems but even higher-level checking capabilities as well (Mahmood, 1988). An up-to-date watchdog processor is a sophisticated system monitor which can execute the following tasks:

- checking of memory access rights (performing basic memory management functions if there is no built-in memory management unit in the checked system);
- acceptability checks of the computed results (the watchdog-processor as a special coprocessor has a built-in routine set in order to check the correctness of the output results. This way no performance degradation of the main processor is established due to additional checking);
- checking of the control signals (the control signals are compacted as binary vector sequences; the so-derived signatures are compared with stored references);
- basic timer functions;
- control-flow checking.

2.4.2 Information based redundancy

Information redundancy is the addition of redundant information to data to allow fault detection, fault masking, or possibly fault tolerance. Examples of information redundancy are error detecting and error correcting codes.

Information redundancy always requires coding and decoding of information. These procedures can be done with the aid of hardware and/or software redundancy. In this case the function of hardware or software redundancy is primary to secure information redundancy and by this means to assure fault tolerance. The application of information redundancy is based on the results of coding theory.

2.4.3 Time based redundancy

The fundamental problem with the forms of redundancy discussed above is the penalty paid in extra hardware for the implementation of the various techniques. Both hardware redundancy and information redundancy may require large amounts of extra hardware for their implementation. Time redundancy methods attempt to use extra time instead of extra hardware.

The basic concept of time redundancy is the repetition of computations in ways that allow faults to be detected. Two basic schemes can be implemented:

- *repetition of a function through repetition of the same operations;*
- *repetition of a function through executing modified sequences of operations.*

The *repetition of the same sequence of operations* will enable to detect either a permanent fault activated during repetition, or transient faults.

The *repetition of modified sequences of operations* will enable to detect transient faults as well as permanent faults effecting the different sequences in different ways. The modified sequence can mean executing the same computation with encoded data, and decoding the result after the computation, or executing the same operation with the complement of the data (this method can be applied to all functions possessing the property of self-duality (Kohari, 1978)) or many other techniques (Reynolds, 1978; Patel, 1982; Johnson, 1984). Although the repetition of an operation requires time redundancy, if the repetition uses different algorithms to execute the required operation, it also means software redundancy. In this case, time redundancy is a side effect of software redundancy.

2.4.4 Software based redundancy

Software redundancy can be used for *data validity test*, *performance test* or *program validity test*.

Software redundancy implementing data validity test can be either the realisation of some kind of data-information redundancy, or consistency check performing tests using a priori knowledge about the characteristics of the data to verify. Typical examples are interval check or consistency check.

Performance tests are performed to verify that a system possesses the expected capability. Typical examples are memory tests or ALU tests.

Most redundancies discussed so far are only appropriate for detecting hardware faults. There are however techniques to prevent software faults as well. Since software faults originate from the development phase, they are always permanent and no duplication technique will detect them. Intended for detecting software faults, software redundancy therefore always means software diversity between the redundant modules. Examples for software diversity discussed in the literature are based on *N-version-programming (NVP)* or the *recovery blocks (RB)* techniques.

In both cases the whole program or parts of it are implemented by different programmers, using different programming languages, different compilers etc. In the case of NVP the programs are operated parallel. It can be achieved by simultaneous execution of the programs on redundant hardware or by executing the programs on the same hardware. The results are

compared, so that error detection and/or error correction can be made like in a hardware voter system.

In the case of RB only one implementation of the program is active. The results of the active program are tested from time to time. If an error occurs, the operation is continued by another implementation using the previous error-free results.

There are however two major difficulties with these techniques:

- Software designers tend to make *similar mistakes*, especially when they have to start from a common specification.
- If *the comparison of results* is done only at the final result state, the likelihood of a fault to remain latent is very high, since digital results have a finite set of values (e.g. if both modules have the result 0, it does not increase the likelihood of being fault-free). On the other hand if the comparison is done too frequently, it will result in too uniform program structure, increasing the likelihood of common faults.

2.5 The operation of a fault-tolerant system

Fault tolerance - in addition to fault prevention, removal and forecasting - is a mean of dependability. It is carried out by *error processing* and by *fault treatment*.

The role of *error processing* is to remove errors from the system state before being activated, i.e. before failure occurrence.

Fault treatment means preventing faults from being activated again. The first step of fault treatment is fault diagnosis, which is followed by fault passivation and - if it is needed - by a system reconfiguration.

In the following these steps are presented in more details. Additionally, the base of fault tolerance, the error detection is discussed.

2.5.1 Error detection

The base of fault-tolerance - especially in multiprocessors and strongly connected distributed systems - is early run-time error detection, since the effects of transient faults can only be detected by concurrent error detection mechanisms. These techniques involve some well-known classical methods for checking fault-free operation of passive devices in microprocessor systems as

- error detecting (and correcting) codes for RAMs and disks;
- checksum for ROMs;
- cyclic redundancy checks for external communications;
- shut-down circuitry for power supplies;
- scheduled tests for all components of the system.

As it can be pointed out, the passive components are generally checked using information redundancy.

The techniques used for checking the active units are mainly based upon hardware redundancy, however, self-checking active components using error-detection (e.g. arithmetic) codes are often reported. These techniques are not completely elaborated yet and there are

numerous approaches and possible design alternatives to be included into an existing or designed system. The most commonly used applications are the master-checker mode and watchdog processors.

2.5.2 Fault treatment

The first step of fault treatment is fault diagnosis. It determines the cause(s) of error(s), their location and nature. After it, the faults have to be prevented from being activated again. It is called fault passivation. It can be carried out by either removing the faulty component or by the reconfiguration of the system. In case of hard faults the fault passivation is always needed. On the other hand, in case of soft faults the diagnosis is unsuccessful and so the fault passivation is not needed, error processing has to be carried out immediately.

Fault diagnosis in uniprocessor systems is a relatively simple task: test routines have to be scheduled and the test results have to be used to provide automatically a diagnostic result. In difficult systems diagnosis can be supported by simple expert systems using detailed knowledge databases about the system configuration, effects and causes of the possible errors. If the CPU itself is faulty then only external testers or simple test and maintenance processors can be used. In simple systems hardware test methods (e.g. boundary scan) provide enough information to locate the faults.

In distributed systems intelligent components can test each other. This way tester and tested units can be defined. The global system state can be generated by a central diagnostic unit or in a distributed way.

Centralized diagnosis means that a single unit receives the results of the tests (syndromes) and on the basis of this information it generates the global system state which is forwarded to the other units of the system. The main drawback of centralized diagnosis methods is that the diagnostic unit is a communication and information bottleneck and its dependability is critical.

The idea of distributed diagnosis is that the intelligent components of the distributed system test each other and communicate the diagnosis information. After a test and communication phase each unit has to agree and generate a consistent system state. The main difficulty is that faulty state of a tester invalidates the test results. Since the testers regularly change information about the global system state, faulty testers can provide faulty information for the others. To avoid such situations, different requirements and limitations are defined for the system (e.g. t-diagnosability, the number of simultaneously diagnosable faulty components) and sophisticated diagnosis algorithms can be used.

The commonly used diagnostic models are the PMC (Preparata, Metze, Chien), BGM (Barsi, Grandolini, Maestrani) or RK (Russel, Kime) models. They define the diagnosability and the diagnosis properties. Using these models some frequently used distributed diagnosis algorithms are the SELF (Kuhl and Reddy) and the Adaptive Distributed Self-Diagnosis (DSD, Adapt; Bianchini and Buskens) (Kime, 1985).

2.5.3 Error processing

If system redundancy (mainly hardware redundancy) is enough then there is a possibility to deliver an error-free service even from the erroneous internal state. This case is called error

compensation.

If error compensation cannot be applied then the erroneous system state has to be substituted by an error-free one. It is called *error recovery*. There are two approaches which can be followed to restore a valid system state from the erroneous one:

- The *forward error recovery* constructs the new system state from the actual (erroneous) state and the result of the diagnosis. It is an application-dependent solution, only ad-hoc methods exist and no general techniques can be mentioned. The new system state often means the safe or degraded operation of the system.
- The *backward recovery* works as follows: The consistent and fault-free state of all processes is regularly saved on a stable storage. These checkpoints are generated at defined points in time called recovery points. If an error is detected then the system is rolled back to the last saved state, the operation is resumed from this valid state. The base of the scheme is the information redundancy (i.e. the checkpoints). In this way the amount and cost of additional hardware is considerably less than using direct hardware redundancy (e.g. backup processors), and quite efficient since most faults are transient.

The backward and forward error recovery schemes are not exclusive. First the backward recovery may be attempted then - if it was unsuccessful - the forward recovery transforming the erroneous system state into a given (safe) state.

3 A SURVEY OF PRACTICAL FAULT-TOLERANT SYSTEMS AND TECHNIQUES

As in practice the basic notions and techniques of section 2 are used in most cases in a combined fashion to achieve a complex purpose, this section is organized according to the operation of a fault-tolerant system. The three main steps discussed here are the following:

- error detection;
- fault diagnosis and system reconfiguration;
- error recovery.

First concurrent error detection mechanisms are surveyed and evaluated. Then examples for the hardware support of the fault diagnosis and system reconfiguration are presented. The following part of this chapter is an overview of some methods of error recovery. At the end, additionally, some techniques used in distributed real-time systems are mentioned.

3.1 Concurrent error-detection mechanisms

The methods mentioned in section 2.5.1 based on information redundancy are well-known and thoroughly analyzed in numerous papers. However, the most important concurrent error detection mechanisms of microprocessors based on hardware redundancy, e.g. modular redundancy (master-checker mode) and the application of watchdog-processors, are not completely elaborated yet and there is an extensive research on this area.

3.1.1 Modular redundancy

Modular redundancy is a highly expensive technique, but it is still the most frequently used method in responsive systems (Malek, 1991) as power stations, airplanes or train control. It is not only an error detection mechanism but means a fault tolerant system structure as well.

As an example the Shinkansen Train Control can be mentioned (Hachiga, 1993). It is a distributed dependable real-time computer system controlling a Japanese railway system having over 700.000 daily passengers and 1 minute average delay. Different types of modularity are used at different levels of service importance. The units of the Passenger Information System and the Data Processing System are dual computer structures consisting of an always active computer and a cold stand-by machine. The critical Route Control System is a triple computer architecture. Two computers are always active working on the basis of the "same software for different hardware" approach, while the third is a stand-by one used for maintenance and test. The most critical Centralized Traffic Control is a TMR system.

The results of the parallelly working computers are compared by a custom designed VLSI circuit called Dual System Controller. It contains a fail safe comparator (stuck-at faults are detected at the output), a test program trigger and the test output comparator. The input and output, the interrupt handlers and the task scheduling sequence of the machines are continuously checked while the internal data is compared using regularly generated

checkpoints. If the fail-safe comparator reports a discrepancy, then a predetermined test program is triggered checking the instruction set and the internal data paths of the computers. The test output comparator decides which computer is faulty on the base of a stored reference output. The faulty computer can be replaced by a stand-by one.

Another example is the Airbus A320 project. In this case reliability is achieved by hardware and software diversity: multiple programs are running on different hardware platforms. Computer programs controlling the airplanes are written by different producers using the same specification. Two identical computers executing the same program are used in master-checker mode. External faults are also detected, since the outputs of two such master-checker pairs on different platforms are regularly compared. Results to be compared do not coincide, in this way tolerance ranges are defined in the comparator unit in terms of functionality and in time as well. Since the comparator is critical, there are 3 independent units executing this task. If only one would be fault-free, external additional information should be used (interaction of the pilot).

Similarly to the control units, also sensors, actuators and mechanical components have been developed for high reliability. Sensors and actuators are multiplied (e.g. there are 3 independent hydraulic systems), power supplies are duplicated (each single power supply is sufficient).

3.1.2 Master-checker mode

One of the disadvantages of the master-checker mode, i.e. the synchronization problem is avoided by chip designers, since they build the synchronization circuit into advanced processor chips.

As an example the Motorola MC88100 processor and its memory management and cache unit (MC88200) can be mentioned. These powerful RISC microprocessor units have a built-in logic for the comparison and proper driving of the input and output lines. The master and the checker unit can be easily configured using a single external control line.

The off-the-shelf multiprocessor modules offered by Motorola can be set up for master-checker mode without complicated hardware modifications.

3.1.3 Watchdog processors

As practical experiences and fault injection experiments show, the majority of failures originates in transient faults. Since a high percentage of these faults is manifested as a disturbance in the program control flow, one of the most important checking tasks of a watchdog processor is the control flow checking. The watchdog, as a relatively simple coprocessor monitors the checked processor using signatures as compact abstractions of the processor state. The signature based approach reduces the information transfer. Run-time signatures representing the actual control flow are compared with precomputed signatures representing the reference control flow. The result of the comparison is a Go/NoGo type signal which can be used as alarm signal triggering fault treatment and error processing.

The reference signatures can be downloaded into the WP before starting the application to be checked, or can be transferred to the WP during the execution of the checked program. The watchdog processor examines with the help of reference signatures, whether the received

run-time signatures are valid.

Run-time signatures can be derived signatures generated by the WP itself monitoring the instruction bus of the checked processor and compacting the subsequent states of the bus as binary values of the signals. This approach results in a fast WP hardware and the frequency of the signature generation and comparison can be easily modified and controlled.

Considering that new microprocessors use built-in cache and instruction pipeline, the operation of the main processor can not be observed any more by monitoring the external bus cycles and deriving run-time signatures this way. The only way to check the program control-flow graph is to assign signatures explicitly to some states of the program. A preprocessor analyzes the (high-level) source code of the user program, computes signatures associated with software states and inserts the necessary signature transfer operations. During the main program execution these signatures are transferred to the WP uniquely identifying the program location. The WP receives and evaluates run-time signatures concurrently. The signature sequence is accepted independently of the semantic correctness of the branch selections, if it corresponds to an existing path in the program control-flow graph, otherwise it is considered to be the result of an error.

The WP methods can be divided into groups on the basis of how they gain run-time signatures and store the reference control flow. Some typical methods are presented in Table 1.

	Stored reference signature data base	Watchdog reference program	Embedded reference signatures
Derived run-time signatures	ASIS, RMP	WDP	BPSA
Assigned run-time signatures	ESIC	SIC	SEIS

Figure 1: Overview of the control flow checking methods

The Asynchronous Signature Instruction Stream (ASIS) (Eifert, 1984) method and the Roving Monitoring Processor (RMP) (Shen, 1987) evaluate signatures using a WP-internal signature database. The Watchdog Direct Processing (WDP) (Michel, 1991a) method uses a special WP program containing the reference signatures and simple instruction codes according to possible branch instructions of the main program. The Basic Path Signature Analysis (BPSA) (Sridhar, 1982) and its improved versions use reference signatures embedded into the main program (e.g. justifying signatures). The run-time and reference signatures are compared at the end of a branch-free instruction sequence.

The first published method based on assigned signatures, the Signature Integrity Checking (SIC) (Lu, 1982) represents the reference control flow by a reference program consisting of signature receive and compare instructions. This reference program is written in the same high-level programming language as the main program and executed by a general-purpose microprocessor serving as a WP.

The Extended Signature Integrity Checking (ESIC) (Michel, 1991b) method implements a finite deterministic stack automaton. The program control flow is represented as a database

defining the valid successors of each signature, the start and end of the procedures are marked by special signatures. Before the start of the main program the signature database is downloaded into the WP. The signature evaluation is completed by the checking of procedure calls: detecting a procedure call the actual signature is pushed onto a WP-internal stack, in the case of return it is restored.

To use these two methods the WP has to be realized by a general purpose microcomputer, since the reference program and the search and compare engine respectively, require high computing power, programmability and a large memory storing the considerable reference database. Due to this complexity the signature evaluation is relatively slow and the implementation results in a machine comparable in complexity with the main processor intended to be checked.

The method called Signature Encoded Instruction Stream (SEIS) (Pataricza, 1993) tries to avoid these drawbacks assuring high-speed signature evaluation and the elimination of the large reference database. Reference signatures are stored within the application program. Each signature identifies the actual state of the program and the valid successor signatures as well. The actual run-time signatures can be evaluated using only the previous valid signature as a reference. The relatively fast and simple evaluation scheme and the lack of the reference database enables the use of this method even in multitasking systems.

3.1.4 Evaluation of the concurrent error detection methods

The most feasible watchdog method can be selected taking into account the structure and properties of the system to be checked and the hardware, memory and time overhead.

If the processor instruction bus can not be observed and monitored then only the assigned signatures method can be used. The SIC and ESIC methods need a complicated watchdog hardware which can be realized only by an additional processor, so the hardware overhead is high. The SEIS watchdog reduces additional hardware and therefore does not include difficult and unreliable parts into the system. On the other hand, it is able to check multitasking systems as well and can be extended to higher-level checking. The memory and time overhead is similar using these three methods.

The derived signatures methods have the advantage that the error latency time (i.e. the time between the occurrence and detection of a fault) is relatively low, the checker and signature generator hardware is simple and fast. However, sending the justifying signatures to the watchdog means similar overhead like in the case of the assigned signatures. If no justifying signatures are needed (WDP method) then the recompilation of the programs is not necessary and the time overhead and performance loss is practically zero (this is important in critical real-time systems).

At the University of Erlangen-Nurnberg different experiments have been done in order to evaluate the simultaneous use of different concurrent error detection techniques (Dal Cin, 1993). Master-checker configurations as well as watchdog-processor methods have been investigated. It has been found that the main drawback of the master-checker method is that external information (received from the outside world of the MC pair) can not be checked and it can lead to a failure. Only additional methods can check the external information (e.g. faulty instruction opcode or data) affecting both the master and the checker. On the other hand, in order to check the internal world of the MC pair, the system state information stored in the registers and cache has to be regularly copied out in order to compare them

with the similar data of the checker processor. This requirement can lead to the degradation of system performance if an improper update policy is used.

The main drawbacks of the assigned signatures watchdog methods are that the error latency can not be strictly limited (if high-level programming languages are preprocessed) and the signature transfer as external data transfer outside the world of the sophisticated pipeline and cache mechanisms breaks the performance down. The solution of this problem is a lower-level program control-flow graph encoding, taking into account the execution time of the instructions as well (this way a constant signature transfer rate can be achieved).

3.2 Fault diagnosis

3.2.1 Hardware level solutions

The designers and producers of commercial multiprocessor systems try to support diagnosis and provide an easy system maintenance which is important in systems having more hundred processors. As examples the Connection Machine 5 of Thinking Machines and the GigaCube of Parsytec can be mentioned.

The Connection Machine incorporates hardware support for system testing: an independent diagnostic network is established in order to help the localization and isolation of the faulty components. The topology of the network is a binary tree with one or more diagnostic processors placed at the root. The diagnostic network is completely testable and can diagnose itself. All other components are testable when in place, for this purpose boundary scan paths can be used. The diagnostic processors have parallel access to the boundary scan chains, generate test patterns, control the testing procedure and isolate the faulty component.

The GigaCube is a scalable massively parallel multi-transputer system. The hardware architecture is divided into cubes containing 4 clusters (16+1 T9000 transputer per cluster). Each cluster comprises an error detecting hardware: error detecting and correcting codes and memory scrubbing (periodically reading the complete DRAM in order to detect and correct latent bit errors). The interconnection network is a robust one having redundant paths (links): it is guaranteed that there is no complete disconnection even if a routing switch fails. A component transputer itself can detect link and switch errors.

There is a control processor in each cube and a tree-like network connects it with the application processors. The task of the control processor is to:

- detect link and connection failures by an error detecting communication protocol;
- detect processor failures by watchdog timers and periodically received "I-am-alive" messages;
- support fault diagnosis by testing and test result evaluation;
- support error recovery and reconfiguration.

The spare processor located in each cluster can be integrated into the system under the guidance of the control processor if an application processor fails.

3.2.2 High level methods

However in this report we consider fault diagnosis as a part of fault-tolerant activities, it is in general formulated in a much broader sense. Though automated diagnosis is one of the

most important fields of AI, no attempts have been made in the past to apply such results in the development of fault tolerant systems. An exhaustive survey of intelligent diagnostic methods exceeds the limitations of this report, for the sake of further reference some of the most general basic ideas are shortly summarized in the following.

Diagnosis can be defined and characterized according to Reiter's (1987) well known theory based on first principles. Its starting point is that any system can be described by the following information:

- a set C of *components* represented by a finite set of constants;
- *the description of the system* defined as a set of first-order logic sentences about its structure, behaviour, expected symptoms, fault causes etc;
- a set of *observations* given as a set of first order logic statements which describe the actual behaviour of the system.

Reiter introduces the predicate $AB(c_i)$ for expressing that component c_i behaves abnormal. The correct behaviour of the system is then formalized by $\neg AB(c_i)$ for $\forall c_i \in C$. A diagnosis is defined by Reiter as the *minimal set* of those components, which behave abnormal supposing that the selected set of components is consistent with the current set of observations. The diagnostic problem itself is to find one or all diagnosis for a given system and given set of observations.

In this formulation the solution of the diagnostic problem is to find one or all solutions for a system of Boolean equations. Without proof we notice that this is equivalent to the satisfiability problem (SAT), so diagnosis is NP-complete. That is why simple exhaustive generate-and-test algorithms will not work even by systems with moderate complexity.

Reiter proposes an algorithm called *DIAGNOSE* for determining diagnoses more efficiently by constructing and pruning special (so called *HS*) trees and using theorem provers to determine the consistency or inconsistency of generated possible diagnoses.

Reiter's theory gives a mathematically exact foundation for handling diagnosis problems, though in practice efficiency is fundamentally affected by the chosen representation method. First order mathematical logics provide a formal way of description, though they are not appropriate for describing arbitrary kinds of systems with arbitrary kinds of properties (see e.g. the reasons for the introduction of fuzzy logics (Zadeh, 1965)). Another problem is the kind of the applied theorem prover, which in this case is based on *resolution*, which besides other problems does not assure the efficiency required in large scale systems.

It is obvious that to solve diagnosis problems a representation method and a practically efficient search algorithm must be constructed. That is why automated diagnostic systems apply techniques a bit different from the clear mathematical apparatus proposed by Reiter. For the sake of simplicity just two successful solutions are mentioned here: the structure and behaviour based diagnostic system of Davis (1984) and the IPCS real-time diagnostic system presented by Padalkar (1991).

The system of Davis is intended for use in automated troubleshooting of complex digital circuits, while the IPCS system operates in a real-time industrial environment. The basic task is naturally the same as defined by Reiter, though a different formulation is used according to the given practical diagnostic problem.

The operation of both systems are based on a hierarchical multi-level decomposition system model, which describes the diagnosed system from multiple aspects. The modelling is done from two basic views: *structural* (which defines system topology and the kinds of components) and *behavioral* (which defines how fault-free individual components operate).

Hierarchical decomposition is required for being able to cope with system complexity from

either representational and algorithmic points of view. It is a very important property of diagnostic systems, since any structures can be decomposed into hierarchies, like e.g. computer level - board level - chip level - gate level - transistor level. Diagnosis can be performed at any levels, though the higher level a solution is found, the less effort must be spent because of the limited number of components.

Another important property of these methods is modelling from different aspects. E.g. the circuit diagram model can be used to identify logical stuck-at faults while other faults (like e.g. bridging faults) are very unlikely to be explained unless other models like printed circuit board layouts are also known expressing spacial adjacency of components. This way multiple kinds of models can be used to describe the actual diagnostic problem.

According to the general terminology a *shallow model* describes the relation between an abnormal behaviour and its possible causes (the actual diagnosis) in the form of rules (typically having the general *if <precondition> then <consequence>* form). Shallow models are highly appropriate for describing rules of thumb or information about the system like fault-trees or experiences of service personnel. Shallow models can be generally quite efficiently interpreted in practice (e.g. by using different inference procedures developed for rule based systems), though they are "as clever as their rules", so they are not able to diagnose faults, whose symptoms (or fault causes) are not stated by certain rules. *Deep models* rather use exact structural and behavioral properties. Though they may require a larger machinery, they can be used for finding diagnosis for any kinds of faults in a given system.

Davis uses an algorithm called *constraint suspension*. It is based upon the structural and behavioral model of the system. Its basic idea is that, if the abnormal behaviour of the system (its corresponding inputs and outputs) are known, the faulty component can be localized by systematically supposing that in the structural model certain components are faulty and *suspending* the normal behaviour of these assumed faulty components, which means that temporarily no restrictions are considered for the local behaviour of the given component (or in other words it can accept any local value combination in its neighbourhood). The faulty component is localized, when during constraint suspension the abnormal behaviour causes no contradiction inside the circuit, when propagating values from inputs forward, and values from outputs backward.

However the modelling approach presented by Padalkar is in its main ideas similar, the diagnostic algorithm is different. It follows from the basically different application area. In industrial technologies direct measurements of different parameters are performed by sensors and other values can be generated by computing derived parameters from the results of other primary and derived measurements. In this case - although a multi level structural and behavioral model is used - the observations are different, in general they are much more detailed than just observing the outputs of e.g. a computer system by given input patterns.

In IPCS as in general by industrial systems, diagnosis is based upon so called *alarm signals* (Kirschen, 1992). Alarm signals are generated, when current and accepted behaviours differ, by comparing actual system behaviour to different kinds of system models. Since the effects of a single fault propagates over different components, the presence of a single fault generally leads to the appearance of many (in complicated technologies often hundreds of) alarm signals. The diagnostic procedure is able to localize the exact fault cause by using a special kind of models called *fault propagation models* defining possible fault interactions between components, and by traversing a strictly tree-shaped model hierarchy of technology blocks. Since this tree traverse is done in a top-down fashion, this diagnostic procedure is

able to deliver more and more precise diagnostic results as time passes, by localizing smaller and smaller parts of the system laying at a successively deeper layer of hierarchy.

3.3 System recovery

After the error detection and successful diagnosis the fault-free service has to be continued. In the following the error recovery techniques are surveyed for different applications.

3.3.1 Error recovery for uniprocessor applications

If only one processor is available for a uniprocessor application then the fault must be carefully diagnosed. If it is proved to be a temporary fault then it can be masked by restarting the application from the initial or from a stored fault-free state. If the fault is permanent and it prevents the continuation of the application then the system has to be stopped (fail-stop behaviour). If this shutdown results in a safe state then it is called fail-safe behaviour.

If two processors are available, one can be used as a spare of the other. This spare processor can be a cold, warm or hot backup.

- The cold backup is not powered until a failure of the active processor occurs. If a permanent fault is diagnosed then it takes over the control and restarts the application.
- The warm backup periodically receives the complete state of the working processor in the form of checkpoints. After fault detection it restarts the application from the last received checkpoint (the time loss is less than in the case of cold backup).
- The hot backup executes the application program parallel with the working processor. They have to be synchronized and use the same inputs. If a fault is detected then a smooth and fast switchover is needed to the backup.

If three or more processors are available for a uniprocessor application then the N-modular redundancy technique can be applied.

3.3.2 Error recovery for multiprocessor applications

In distributed systems network faults as well as node faults have to be considered. This results in a more complex recovery scheme (Deconinck, 1992). The generally used method is the backward error recovery.

The main problem of the checkpoint generation is how to gain a consistent system state. The four classes of checkpointing are distinguished by the way the checkpoint is taken from the different (communicating) processes:

- *Semi-automatic technique*: In this case the programmer inserts the checkpoint generation instructions into the application using special language constructs or subroutines. The responsibility of the system designer is to take care of the consistency.
- *Message logging*: Processes save their state independently and the interprocess messages are logged.
- *Coordinated checkpointing*: An interacting set (or all) of the processes save their states

together; so called conversations (parts of the processes where they intensively communicate) can be defined. At the begin of the conversation each process saves its state, after it they communicate only with the others taking part in the conversation. At the end of the conversation the new state can be saved.

- *Hybrid technique*: It is the combination of the two previous techniques.

As a case study, the rollback recovery of the MEMSY multiprocessor can be mentioned. In order to get a consistent system state a two-phase commit protocol is used. All of the processes of the system generate its "tentative next" checkpoints autonomously. The tentative next checkpoint becomes a valid one if all of the processes report that the generation of their tentative checkpoint was successful. A high-speed (fibre network) broadcast protocol is used to get this agreement.

The checkpoints are stored in two level. The soft checkpoints are generated often and stored in high-speed local memories. The hard checkpoints are stored in a persistent stable storage using a highly reliable, secure communication protocol. If an error is detected then the system tries to roll back to the last soft checkpoint. If the restart was successful then the soft checkpoint is a proved good one and it can be stored on the stable storage (as a new hard checkpoint). If the restart from the soft checkpoint fails then the hard checkpoint can be used. In this way the rate of the slow hard checkpoint generation is equal with the mean time between failures while the fast soft checkpoints can be generated frequently enough.

3.3.3 Error recovery for distributed real-time applications

There is an essential need for fault-tolerant computing in distributed hard real-time systems. However, applying fault-tolerance techniques the requirements of the timeliness have to be taken into account as well.

As an example for a fault-tolerant distributed real-time system the MARS project (Maintainable Real-time System) can be mentioned (Schwabl, 1989). The special considerations and unconventional techniques implemented in the MARS system in order to guarantee maximum response time even in peak load situations are not described here, only fault-tolerant aspects are considered.

The methods of the application software fault detection take into account the real-time requirements as well:

- During the scheduling phase of the application the worst case execution time is calculated, which becomes the slot of the task. If this time limit is exceeded then it means a severe error.
- Time redundant execution of tasks: In most cases the execution time of a task is smaller than the calculated maximum time slot; in this case the operating system executes the task a second time. To be able to detect faults in the code of the task the two versions are executed using different core images (i.e. registers, address ranges etc.). On the other hand, the code segment is provided with a checksum which is checked in the idle loop of the processor.

The MARS system is based on fail-silent components and active redundancy. The fail-silent property means that the faulty component (a single-board microcomputer) has to be switched automatically off preventing further failures, the results of a redundant component are

available instantly.

Whenever a component fails, it shuts down and extensively tests its processor and peripherals. If the fault is transient, first the code segment of the component is restored reading it from a maintenance unit. After it the state of the working tasks is recovered. In the MARS system the tasks are stateless, which means that at the end of each execution cycle each task puts its inner state into a bus message, at the beginning of the cycle it will be read. In this way the programmer need not take care of recovery phases.

If the fault is permanent then the actual component has to be removed and repaired. The time between the shutdown of a faulty component and the start of the full operation is critical. In order to solve this problem the MARS uses the concept of shadow components. During the normal operation no network time slot is assigned to the shadow component (it does not send messages). In case of failure the slot of the faulty component will be assigned to it. The switchover is made immediately since the internal states are equal. After repairing the faulty component can act as a new spare one.

4 PROBLEM FORMULATION AND THE ELEMENTS OF A REPRESENTATION MODEL

In this chapter the basic problem of maintaining fault tolerance at a general system level is specified and a method is presented, which can be used to describe and interpret system models for achieving system level fault tolerance using intelligent diagnostic centers.

A *distributed system* contains several loosely coupled, communicating nodes, where each node is able to perform autonomous actions and the whole system is working on a common task.

Since very diverse kinds of distributed systems exist, we first examine three important classes of distributed systems' applications (*multi processor computer architectures* (a), *computer networks* (b) and *industrial technologies* (c)) to shortly discuss the potential benefits of system level fault tolerance in them and to point out those common aspects, which make it possible to establish a general framework for handling the problem of fault tolerant operation in them.

From the point of achieving fault tolerant operation the above mentioned three system classes have different properties.

- a) *Multi processor architectures* have predefined, often fixed topology. In such architectures reliability can be estimated at design time and fault tolerance can be achieved at the *hardware level*. On the other hand multi processor architectures appear as a single component at the system level, so we can consider the possibility and the necessity of system level fault tolerance minimal in this case. A further important issue considered with multi processor architectures is, that "traditional" fault tolerance techniques (see sections 2 and 3) are worked out for this domain the best, and they admit the use of fault tolerant design principles.
- b) *Computer networks* are loose nets of computers, with not fully determined structure, which can evolve in time, if the network is extended by additional nodes. If the number of nodes exceeds a given limit, the number of faults within the whole system will increase because of the finite reliability of individual components. Under such circumstances to maintain overall system reliability becomes highly difficult, though it is more and more important. As individual nodes of a computer network are not fault tolerant, and there is no hope for solving this problem by exchanging all nodes by fault tolerant computer architectures, the application of system level fault tolerance for computer networks can be advantageous.
- c) *Industrial technologies* are systems of extraordinary complexity. The design, implementation and operation of industrial technologies is extremely complicated and costly, and their life time (especially in Hungary) can be very long. On the other hand system failures are always expensive and often dangerous. The currently available fault tolerance in industrial technologies is at hardware level and is only present at the lowest control layer in programmable logic controllers or in the form of the duplication of certain components (Johnson, 1988). The application of higher level intelligent diagnostic and advisory systems is not widespread, and such systems do not assure fault tolerance in themselves. We can conclude, that in an industrial environment system level fault tolerance can be essential.

Systems belonging to groups b) and c) have in common that though fault tolerance would be necessary, it cannot be accomplished by merely traditional, hardware level fault tolerance techniques, because of their high overhead and costs. Only those methods are acceptable, which can be realized using the given finite resources of the system by adding just a relatively limited overhead. This can be accomplished by using implicit redundancy, which is always present in complex distributed systems.

4.1 The Basic Problem

In this report a *distributed system* (or simply *system*) means a set of *loosely coupled*, interacting *nodes*. Each node is a *resource* that performs given *services* to achieve the *global service* of the system. The *global service* is considered as satisfying a set of *interrelated requirements*. The notions used in the previous definition are present in *any* distributed systems, so they can serve as a starting point, which is independent from the exact type and properties of the distributed system.

The system operates properly, if any requirement Q_i is fulfilled by an appropriate service S_i and the

interrelations between requirements are also satisfied. If S_i initially satisfies Q_i , and later S_i becomes unable to satisfy Q_i , S_i is called a *fault*³.

This definition is a general *functional fault model*, which appears at a high abstraction level. The criteria for a service to satisfy a requirement is further characterized in section 4.10. A fault is *masked*, if its effects do not cause changes in the global service (or global behaviour) of the system. If a fault cannot be masked, it becomes a *failure*.

The *distributed system* (or *supervised system*) is augmented with a *diagnostic center* which maintains fault tolerant operation by continuously observing the operation of the distributed system, by detecting faults and substituting faulty services with fault free ones by modifying the structure, operating mode or other attributes of the supervised system.

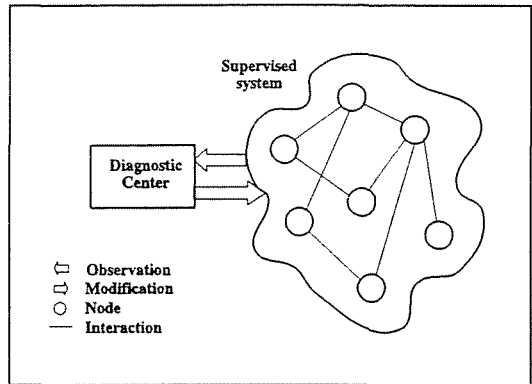


Figure 2: The general model of system level fault tolerance

4.2 General properties of a diagnostic center

Our basic intention is to find a solution which make it possible to achieve general system

³ It is important to notice that according to the terminology introduced in chapter 2 S_i should rather be called an *error*, though in our model fault causes are not localized (see section 4.2 for more details), so there is no need for distinction between a fault and its effects (which are normally called errors). For these reasons in the following only the notion of *fault* is used.

level fault tolerance by creating diagnostic centers and adding them as a relatively small hardware and software overhead to large scale systems (like computer networks or industrial technologies), which require high reliability but which were originally not necessarily designed to be fault tolerant. The successful operation of diagnostic centers is based upon different kinds of redundancy, which is always present in large scale distributed systems.

The supervised system is in general highly complex. To be able to satisfy its goal, the diagnostic center maintains the following basic tasks:

- *Data acquisition and communication*: to continuously collect data about the operation of the supervised system and send information to the supervised system about the required actions, which must take place for masking existing faults.
- *Modelling*: to maintain a model of the supervised system and to match incoming data with that *model*.
- *Checking and recovery*: to check - based upon the actual system model - whether the operation of the supervised system is satisfactory, and in the case of faults to *signal* and *mask* them.

Though all these tasks must be solved for the successful construction of a diagnostic center, this report deals primarily with the questions of *modelling* and *checking/recovery*.

Data acquisition and recovery is also important, though not addressed here due to the following reasons.

In the proposed model *communication* can be simplified to the presence of certain information (e.g. attribute values see section 4.4), which in practice may require extensive data exchange, though in theory it means no further problems to handle it.

The model is capable for hosting different kind of observation methods (like the use of built in software exceptions, memory management units or sensors), but this is no integral part of the model.

These questions will be answered in a later phase of research.

4.3 General characteristics of the model

The model has to assure the possibility of representing and handling information about complex systems. When establishing a modelling method, the following characteristics must be considered:

- The elements of the supervised system can be very diverse.
- It may be required to handle many different kinds of supervised systems in different applications.
- Different system components may require different kinds of modelling techniques to achieve representational and algorithmic efficiency.
- The diagnostic center must continuously match incoming data against the system model and decide what to do. Such decisions have time limits, e.g. the cycle time of data acquisition. Even if no such time limits exist, the interpretation of very complex models containing a large number of elements may require an unacceptably long time (even hours or more).

The main problem is to combine representational and interpretational (algorithmic) efficiency.

Considering that the modelling problem is usually hard to solve using a single method and

a single data structure, the model must be divided into parts, which reflect the different properties of different subsystems. On the other hand the interpretation of complex system models results in complicated algorithmic problems which can hopefully be solved only by decreasing the actual problem sizes. The guideline to solve these is *hierarchical decomposition*. Hierarchy is a general property of "natural" systems and a widely accepted and used design method of man made systems e.g. (Dahl, 1976; Mitra, 1988; Chandrasekaran, 1988). These assure the general applicability of hierarchical decomposition. On the other hand decomposition methods can significantly decrease average algorithmic and memory complexity of hard problems. The following model is proposed.

The decomposition consists of *levels*. Each level represents an abstraction of the *whole* system. Any level is built of one or more *planes*. At each plane the model has a given structure built of *components* (defined in the following sections) and planes are connected to each other in a well defined way.

The highest abstraction is incorporated in the 0th level, where the system appears as a single plane. Higher numbers will indicate lower abstraction levels in this report. Components can be *compound* or *simple*. Compound components are decomposed to more detailed descriptions at a distinct plane of a lower level. Simple components are not decomposed further⁴.

There is a diagnostic center associated with each plane. Every diagnostic center has the observes the operation of the associated plane and maintains the services of the particular plane towards the upper levels. The scope of a diagnostic center is strictly limited to the components contained in its associated plane. Simple components are observed directly and individually (e.g. via sensors in an industrial technology, by exception handlers or memory management units in a computer environment). In the case of compound components (or simply *compounds*) information is delivered by those diagnostic centers, which are associated to the planes containing the detailed decomposition of the particular compounds. Briefly it means that instead of compound components the associated diagnostic centers are seen by an upper level diagnostic center. It also means that any diagnostic center has predefined methods⁵ to have access at its associated plane to data about distinct components.

According to section 4.1. any diagnostic center is familiar with the set of *requirements* that the associated plane must fulfil, it can choose from among the set of *services* which can be executed to fulfil certain requirements, and the components themselves are considered *resources*⁶ able to actually perform the specified services. The diagnostic centers may fulfil given requirements in more possible ways (e.g. there can be more services which can satisfy a given requirement and there may be more resources that can provide the same service). It can be considered as a kind of redundancy, which - as demonstrated later - is present in most complex systems.

When the system operation is started, the system is supposed to be fault free, so by a proper design all the requirements are satisfied at all the planes. If a fault occurs, the

⁴ It does not necessarily mean that such components are "atomic" parts of the system, it rather expresses a decision in the design process of the diagnostic center models.

⁵ In fact these methods are determined by the applied *error detection* methods.

⁶ *Resources* and *components* denote the same thing in this report, the distinction is only necessary to assure better understanding in different contexts.

diagnostic center associated to the faulty component tries to mask the fault by finding another way to satisfy the requirements of the given plane. If the diagnostic center is unable to satisfy a given requirement this way, it signals this fact to the diagnostic center associated with the higher level plane as a failure of the appropriate service. It does not necessarily cause a system level failure, it merely appears as a fault at an upper level. Since faults can be observed in simple components first, this fault masking process starts from the lowermost level and results in a failure only when the diagnostic center of level 0 fails. This solution is advantageous, because the complexity is decreased by the checking and masking of faults being local to single planes, furthermore the fault masking effect is multi level, as a fault must pass through more levels to become a failure.

The diagnostic centers in a real situation can be *distinct physical computers* (which may be advantageous in distributed systems covering large area) or can be *conceptual*, if diagnostic centers of more planes are placed to the same computer (which may be advantageous in smaller scale systems), though it means no difference from the point of view of the introduced concepts. The diagnostic centers are supposed to be *hard cores* (Kime, 1985), so they never get faulty. It means that they must run on reliable hardware architectures. The questions of designing the hardware architecture of diagnostic centers is not addressed in this report, however the hard core requirement can be significantly eased, if there are more physical diagnostic centers present and they are designed to be able to partly or fully substitute each other in some predefined ways.

In the following the elements and structure of such a hierarchical system model and an algorithm for its interpretation is proposed.

4.4 Elements of a plane

At each plane the model contains the following elements:

Requirements:

They define the set of those tasks which the given plane must perform. A simple example is shown in Figure 3⁷.

Each requirement has a name and is associated with a set of (*attribute*, *tolerance range*) pairs, which describe

the requirement and define those values, which the given attributes can take when the requirement is satisfied. Tolerance ranges can be discrete (when they are predefined finite discrete sets of values) or continuous (if the ranges are given by continuous intervals).

The *attribute structure* (Figure 4) describes in the form of relations the *dependencies* between different attributes of a given requirement.

```

You need an egg
((quantity 1)
 (state (row, hard))
 (time [0, 45]min)
 (type (chicken, goose, ostrich))
 (weight [50, 1000]g)
 (temperature [-20, 100]°C)
 )

```

Figure 3: *Elements of a requirement*

⁷ In the following examples are taken from the "world of egg boiling" described in chapter 5. We think that at this point the exact semantics of the examples are less important than the syntactic elements incorporated in different model components and the components themselves. On the other hand boiling eggs should be familiar to anyone. If in spite of these difficulties with understanding would arise, we encourage the reader to refer to section 5 for more details.

Different requirements belonging to the same plane form a structure expressing that requirements (similarly to the steps of recipes for industrial technologies or in a cookbook) cannot be satisfied in an arbitrary order. This structure is described in section 4.6 in more details.

Service types:

are the features, which can be used at the given plane to achieve the given set of requirements (see an example in Figure 5).

A single service type may fulfil more requirements, even the same time. Each service type is associated with a given set of *attributes*. Attributes can be connected by relations expressing dependencies between them.

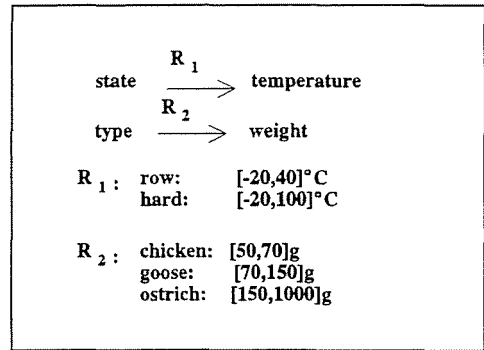


Figure 4: A simple attribute structure

Resources:

are the *components* of the supervised system, which are capable to provide different service types. Resources can be *physical* (like a computer hardware) or *conceptual* (like a software module). Any resource has an associated set of service types and any service type can be associated to more resources. Resources can be considered as real existing parts of the supervised system, while requirements and service types are specifications and models for the operation of resources.

Services:

With any resource R_i , a *service type* S_i can be used to generate a new *service instance* (or simply *service*) $s_i=(S_i,R_i)$, by creating a new service with exactly the same attributes as defined in S_i and assigning values to its attributes by using the attribute values of R_i , which is a resource capable of providing services of the type S_i .

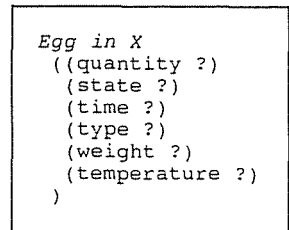


Figure 5: Example for a service type

Attributes:

Attributes do not form a distinct part of the model, but their role is so central that they deserve distinct attention.

Using attributes is the general mean of describing properties of requirements, services and resources. The basic method proposed in this report for maintaining fault tolerant operation is also based on attributes, it is called *attribute mapping*, where attributes of requirements are matched to attributes of available service types to find out which services can fulfil the given requirement and to instantiate new services by matching resource attributes to the given service type. This mechanism is described in section 4.10.

Attributes of requirements, services and resources can be *direct* (when a symbolic or numeric value is associated to the attribute) or *derived* (when the value of the attribute is described as an expression involving other attributes). In general we can say that an attribute is described by a constant function or by a function of other attributes. Furthermore it is possible to associate certain procedures with attributes, which define the way of determining the value for that attribute or consist certain other actions which must

be performed, when the attribute is assigned or referred to (Minsky, 1975).

Diagnostic centers:

For each plane there is an associated *diagnostic center*, which maintains the *global service* of the plane even if certain *requirements cannot be satisfied*. In this case the *diagnostic center* finds those requirements which fail and tries to satisfy them with other services available at the given level.

4.5 Structural properties of the model

The planes in the model can be considered as nodes of an acyclic directed graph. Between two nodes (planes P_i and P_j) there is an edge in this graph directed from P_i to P_j iff P_j is a decomposition of a component in P_i . A plane P_j can be connected to an arbitrary number of P_i through P_iP_j edges, where $i < j$. It also means that P_j can reside at an arbitrarily lower level than P_i , but no edges can be directed towards upper level planes. If the restriction $i=j-1$ would be made (i.e. the decomposition of a node should have to appear at the level directly "under" the level of the node), it would make the model seemingly clearer, though it would make the use of shared planes (see next paragraphs) much more complicated.

The acyclic property is needed, because without it infinite recursive decompositions would be possible, which would complicate the cooperation of diagnostic centers at different levels. On the other hand the cyclic property is not necessary to create hierarchical decompositions of real systems, so it is reasonable to exclude it.

The model allows *shared planes*. A plane P_i is shared if P_j and P_k are two distinct planes and P_iP_j and P_iP_k edges exist ($i > j$, $i > k$, $j \neq k$). If shared planes were not allowed, the structure of any model would be a directed tree.

Shared planes are necessary because of the presence of different shared components in modelled systems. On the other hand tree structures in general offer very nice algorithmic properties (Fujiwara, 1985; Dechter, 1988), so before the introduction of shared planes we had several long discussions. Their result showed that without shared planes the representational power of the modelling method would significantly decrease. On the other hand the model assures locality to each plane, so no traditional extensive graph search is performed upon the decomposition hierarchy, which means that shared planes are no obstacle of algorithmic efficiency.

4.6 The requirement structure

The requirements at a given level are ordered in a directed graph, whose nodes are requirements and whose arcs are causal dependencies between requirements (see Figure 6).

The requirement structure must express the following features:

- a) The kind of dependencies between any given requirements.
- b) The kind of timing which must be fulfilled when fulfilling given requirements.

a)

In the following it is supposed that requirements are uniquely labelled in the requirement structure, i.e. different nodes denote different requirements. The following types of primitive dependencies are needed to express the relation between requirements A , B and C .

- $A \rightarrow B$ A must be satisfied before B can be satisfied. This is a simple ordering. Such a relation holds between e.g. requirements Q_4 and Q_6 in Figure 6. A is called a *predecessor* of B , while B is a *successor* of A .
- A and $B \rightarrow C$ Both A and B must be fulfilled before C can be satisfied. The order of the satisfaction of A and B is arbitrary. Such a relation holds between e.g. requirements Q_4 and Q_5 in Figure 6.
- A or $B \rightarrow C$ A or B must be satisfied (or both of them) before C can be satisfied.

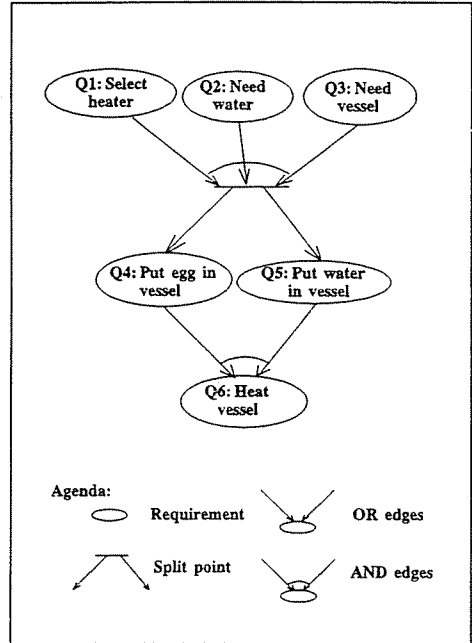


Figure 6: Example for a requirement structure (boiling eggs)

Using these primitive relations compound expressions called *predecessor expressions* can be created. E.g. the predecessor of Q_4 in Figure 6 can be expressed as Q_1 and Q_2 and $Q_3 \rightarrow Q_4$.

A predecessor expression P_j associated with requirement Q_j can be evaluated by considering any Q_i predecessors as a logic variable, assigning the TRUE logic value to it, whenever it is satisfied, and assigning the FALSE logic value to it otherwise. The result of the evaluation of P_j tells, whether the satisfaction of Q_j (independently of available services and resources) is currently possible or not.

According to this a special, predefined requirement can be introduced, called *split point*. A split point is automatically satisfied whenever its associated predecessor expression evaluates to true. E.g. a split point connects requirements Q_1 , Q_2 , Q_3 with requirements Q_4 and Q_5 in Figure 6. Requirement structures could be described without split points, though in this case dummy requirements with no attributes should be introduced, which would unnecessarily complicate the construction of requirement structures.

Requirement structures are interpreted using a simple algorithm called *IRS (Interpret Requirement Structure)*.

Because of the construct of requirement structures and the *IRS* algorithm, no explicit *conditional* structure or *cycle* is needed.

b)

The edges of the requirement structure can be associated with time delays which are actual maximal delays needed to fulfil the given requirement. Such delays can be computed when mapping the requirement structure to such services which have an attribute defining the time spent for solving the given subproblem. Timing requirements are only important, if there is a global timing requirement for the whole requirement structure (i.e. for the *service* which

IRS:

1. $Q_i \leftarrow$ the set of requirements with no predecessors.
2. $Q_{act} \leftarrow$ choose and delete such an element from Q_i , whose satisfaction is allowed by its predecessors.
3. Satisfy Q_{act} .
4. **If** Q_{act} has successors
then add all of them to Q_i ;
 continue by step 2.
else the requirement structure is satisfied, stop.

Figure 7: The IRS algorithm

was decomposed to the given structure). In this case the worst case time of fulfilling the given requirement structure can be easily computed (e.g. by network flow optimization methods on the requirement structure) and the applicability of a given service to satisfy given requirements can be determined.

An important property defines the connection between services and requirements. A *service* at level i can be mapped to a *set of requirements* at level j where level j is a decomposition of level i . Any requirement structure can be derived from a pair $Q=(S_i, R_j)$, where Q is a requirement structure, S_i is a service type and R_j is a resource. It means, that the decomposition of a given service to a requirement structure depends on the type of the service and the resource which is actually used to provide the given service type. On the other hand any level can have more possible requirement structures, even existing parallelly.

4.7 The service structure

The service structure expresses the dependencies between given services of a given level (see Figure 8). The service structure is a directed graph whose nodes are services (service type - resource pairs) and whose edges are relations expressing dependencies between services.

Four kinds of relations can be defined to describe edges of a service structure (supposing that service A is selected before service B is tried to be used):

A **all** B A allows B . If service A is used, service B can be also used. Such dependencies can be expressed as *always true* relations, so no edge is needed for describing them in the service structure. This dependency can be interpreted as the two services are independent. In Figure 8 such a relation holds e.g.

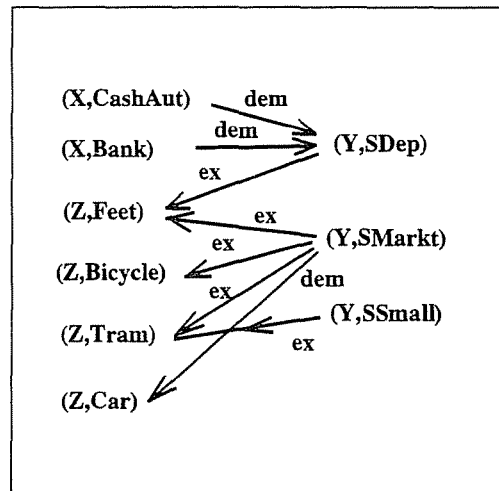


Figure 8: Example for a service-resource structure

as the two services are independent. In Figure 8 such a relation holds e.g.

between services (Y,SDep) and (Z,Bicycle), which means that it is possible to reach a given department store to buy something using a bicycle within a given time limit (for more details see section 4).

- A dem B* *A* demands *B*. If service *A* is used, service *B* must also be used. This is a causal connection between *A* and *B*. In Figure 8 such a relation holds between e.g. (Y,SMarkt) and (Z,Car), which expresses that to reach the distant market within the given time limit, is only possible by car (for more details see section 4).
- A ex B* *A* excludes *B*. Service *B* cannot be used whenever service *A* is used. In Figure 8 such a relation holds between e.g. (Y,SDep) and (Z,Feet), which expresses that it is impossible to reach the department store within the given time limit on foot (for more details see section 4).
- A R B* There is a general relation *R* between *A* and *B* which defines the affect of using *A* on *B*. The typical kinds of such relations are subject of further research.

4.8 The resource structure

From the point of satisfying requirements by certain services it is enough to see an *unstructured resource set*, which can be considered as a simple listing of resources applicable at a given level. If the service structure is constructed properly, no structural information of resources is needed.

4.9 The fault model

A fault can be modelled in this environment as a degradation of certain attributes of given active services.

The degradation of an attribute A_s of a service instance s currently satisfying requirement Q becomes a fault, if the attribute structure of requirement Q cannot be satisfied with the current value of A_s . It can happen because the value of A_s lays outside the tolerance range of A_Q (the attribute of Q mapped to A_s), or in more sophisticated cases the constraints in the attribute structure of Q cannot be satisfied. These explain the necessity of tolerance ranges for attributes.

4.10 The algorithm of the diagnostic center

A *diagnostic center (DC)* is associated to each plane P_i . The DC has the task to satisfy the requirements of P_i by using the services of P_i provided by the resources available at P_i . The DC fulfils this task by continuously monitoring the state (expressed by attribute values) of the services and resources at P_i and if certain attributes change, the DC has to perform *certain actions* to maintain the global service of P_i .

The attributes of *resources* are taken into account as *available services* in *service pools*. When a service of a resource is used, it is considered as an instance of a given service type. Service instances are generated from resources as from service pools, i.e. the resource attributes are modified according to the new service instance. It also means that in general only a finite number of service instances can be generated according to resource attributes.

When a service instance is not required any more for some reasons, the used resource attributes are *given back* to the resource which was used to instantiate the service. More precisely existing attributes are given back. From this aspect there are *consumable attributes* (those, which can be used up as time passes, e.g. fuel, unique messages etc.) or *non consumable attributes* (which do not decrement in time, e.g. memory capacity or certain hardware resources).

-
-
1. If any attribute A_j of service instance $s_i=(S_i, R_i)$ has degraded and if the degradation is outside the tolerable range, continue by step 2, else repeat step 1.
 2. Determine those requirements $Q_k=\{Q_{k,1}, \dots, Q_{k,n}\}$ which were partly or fully satisfied by s_i and for which the degraded attribute is outside the tolerable range.
 3. While Q_k is not empty, repeat the following:
 - 3.1. Q_{act} = select and delete the first element of Q_k .
 - 3.2. s_{act} ← the service which satisfied Q_{act} .
 - 3.3. if $\exists R_j$, such that $s_{act}'=(S_j, R_j)$ and s_{act}' satisfies Q_{act} ,
 then $s_{act} \leftarrow s_{act}'$;
 - 3.4. elseif $\exists S_j$ different from S_i ,
 then
 generate a prototype of S_j by matching attributes of Q_k to S_j ;
 generate s_{act}' by matching the prototype of S_j against an available resource which provides it;
 if s_{act}' could be generated
 then $s_{act} \leftarrow s_{act}'$;
 else continue by step 3.5.
 - 3.5. elseif
 an untried alternative path of Q_{cr} requirements exist,
 which does not contain Q_{act} and can lead to the satisfaction of the requirement structure
 then add all Q_{cr} to Q_k ;
 else report failure for the upper level and go to stand by state (step 5).
 4. Give back all the used resources of all those S_i service instances, which are no more used in satisfying any requirements. Kill all such S_i s. Continue by step 1.
 5. Stand-by state.

Figure 9: *The PRA algorithm*

The generation and retraction of services and the maintenance of resource parameters are performed by the *Plane Recovery Algorithm (PRA)* (Figure 9).

PRA starts from an initial set of service instances which satisfy the initial set of requirements. *PRA* is in an idle state until no degradation is present in the set of active services (existing service instances) or the degradations are inside the allowed tolerance ranges. The operation of *PRA* starts, when a service instance s_i appears, whose degradation is so significant that it lays outside the allowed tolerance ranges; s_i is a *fault*.

Step 3 in Figure 9 explained in some more details in the following. In step 3.4 a *prototype* service is reached after a service type is matched against a requirement and it is found that the given service type can be instantiated to fulfil the given requirement. A service prototype lays between a service type and a service instance. It does not contain exact values, just allowed (required) ranges of attributes prescribed by a given requirement. After a prototype is successfully matched against a resource, a real service instance is generated.

In step 3 *PRA* first tries to find another resource which can provide the same service to fulfil the given requirement. If no such resource exists, it tries to find and instantiate a different service. If no such service is available, it tries to find such an alternative path of satisfying the requirement structure, which does not contain the unsatisfiable requirement. If none of these is possible, the algorithm fails, and signals a failure to the upper level, which will appear as a fault there.

This is an initial version of *PRA* with several open questions. There are no heuristics incorporated to select *optimal* or *good* service types to satisfy a requirement and to determine which resources can be used to provide a given service the best. It seems plausible to associate *priorities* with attributes and create a method which can compute priorities for requirements and services using attribute priorities. In this case the optimality of different choices can be classified and those services can be selected to satisfy a requirement, which in the actual situation seem to be the best.

It is also not characterized, how attributes of service types can be matched against resources and requirements. The basic questions in this aspect are what is the basis of the matching (seemingly attribute names), and what are those acceptable differences between attribute sets of requirements, services and resources, which can be still considered as matching sets.

A further open problem is, how a plane can compute attribute values for the compound resource connected to it at an upper level. Each plane is familiar with its own requirement structure, its services, resources and all the attributes of any kind of plane components. The decomposition method described in the previous sections assures that the diagnostic center of the plane can construct the global service of the plane by satisfying the elements of the requirement structure, but this is a different task from computing *global attribute values* from the values of local attributes.

A possible solution for this problem is to decompose attributes of a service to attributes at a lower level and to define a procedure which is able to determine the global attribute value from its local components, though this needs some more investigation.

Another important question is the meaning of *stand by state* in step 5, when there is a requirement which cannot be satisfied and no alternative requirements are present to substitute it. In stand by state *PRA* of the given plane waits for a change of requirement structure from the upper level (in this case the modified requirement structure may be possible to be satisfied) or the degraded resources may be mended after a while (change in a lower level), which can result again that the requirements can be satisfied.

PRA is an algorithm which tries to satisfy the requirement structure in all possible ways using available service types and resources.

5 AN EXAMPLE FOR THE CONSTRUCTION AND USE OF THE MODEL: THE WORLD OF EGG BOILING

In the following the concepts outlined in the previous sections are demonstrated in a relatively simple example, the "*world of egg boiling*". The task is to boil a hard egg.

It is one of the simplest tasks in everyday life. It may seem strange, though it suits our needs, because human problem solving is in general *fault tolerant*. This means, that humans use their tools, resources and abilities in many possible ways when they fail to solve the problem with given tools and methods. So this is a suitable domain to demonstrate our concepts in practice. On the other hand boiling eggs is familiar to anyone, so the reader can fully concentrate on the details of the presented methods with no efforts of understanding the problem domain itself.

5.1 Elements of the world

To be able to create a model which is *complete* at a certain level, the elements of the world must be strongly restricted. The world of egg boiling, whose model is described in main details in this material contains the following elements.

Kitchen: Your own kitchen (part of your flat, but the flat itself is not part of the world).

Refrigerator: Stores cooled eggs.

Cupboard: Stores room temperature eggs and vessels.

Pan, Glass pan, Jar: vessels.

Table: You can put things on it.

Tap: Supplies water.

Jug: Contains water.

Gas heater: Can heat something, has four burners, requires gas.

Electric heater: Can heat something, has one plate, requires electricity.

Actor: You, i.e. the component which makes things move. The actor is a special component in this world, which - in this case - is the only one to execute the tasks of a diagnostic center either.

Neighbour: The kitchen of your neighbour (part of your neighbour's flat, which also lays outside the world). For the sake of simplicity it has the same components as *Kitchen*. There is a possible modification by introducing:

NActor: Your neighbour, who can do the same things in her own kitchen and even may be asked to substitute you and do whatever is needed in your kitchen. You may also go to her kitchen to perform different tasks.

Environment: The larger physical district of the kitchen laying outside your house.

Vehicles: Means for taking you from a given place to another.

Feet: You can walk.

Bicycle: You can ride it.

Car: You can drive it.

Tram: You can walk to the tram station and take the tram.

Shops: Places, where different props (e.g. eggs, vessels or water) can be bought for

the accomplishment of the task.

SSmall: Small food store in your street.

SDep: Department store. Sells everything. Not quite cheap. Lays not too far. Can be reached by tram.

SMark: Market. Lays far away. Everything is sold. Cheap. Cannot be reached by tram.

Money sources: Places and methods which serve you money.

Purse: Contains some coins and banknotes.

Bank account: Contains a lot of money all saved from your salary.

Cash automar: It gives you cash from another account after you have typed in your id code.

Other money sources (like working, robbery etc.) lay outside the world.

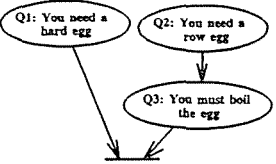
5.2 The model

The following tables summarize the components of the model. There is a distinct table for each plane. Levels and planes are indicated by numbers. For describing requirements, services and resources and defining their attributes, tolerance ranges etc. a Lisp-like syntax is used similar to property lists, where each attribute list consists a list of (*attribute-value/tolerance range*) pairs.

Level 1.:

Requirements	Service types	Resources
<i>Q</i> : You need boiled egg ((quantity 1) (state hard) (time [0,60]min) (type [chicken goose ostrich]) (weight [50,1000]g) (temp [25,60]°C))	<i>S</i> : Boiled egg ((quantity ?) (state ?) (time ?) (type ?) (weight ?) (temperature ?))	<i>R</i> : World of egg boiling ((quantity ∞) (state hard) (time [5, ∞]min) (type (chicken goose ostrich)) (weight [50,1000]g) (temp [2,100]°C))

Level 2.

Requirements	Service types	Resources
<p>Q_1: You need egg ((quantity 1) (state hard) (time [0,45]min) (type (chicken goose ostrich)) (weight [50,1000]g) (temperature [20,100]°C))</p> <p>Q_2: You need egg ((quantity 1) (state row) (time [0,45]min) (type (chicken goose ostrich)) (weight [50,1000]g) (temperature [20,100]°C))</p> <p>Q_3: You must boil the egg ((time [0,15] min) (max_temp 100°C))</p> 	<p>S_1: Egg in X ((quantity ?) (state ?) (time ?) (type ?) (weight ?) (temperature ?))</p> <p>S_2: You can boil it in Y ((time ?) (max_temp ?))</p>	<p>R_1: Kitchen ((bind-to X) (quantity 12) (state row) (time 1min) (type chicken) (weight 60g) (temp (2°C 22°C)))</p> <p>((bind-to Y) (time 7min) (max_temp 100°C))</p> <p>R_2: NKitchen ((bind-to X) (quantity 8) (state row) (time 10min) (type chicken) (weight 60g) (temp (2°C 22°C)))</p> <p>((bind-to Y) (time 25min) (max_temp 100°C))</p> <p>R_3: Environment ((bind-to X) (quantity ∞) (state row) (time [15, ∞]min) (type (chicken goose ostrich)) (weight [50,100]g) (temp 25°C))</p>

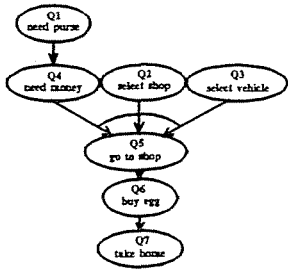
Level 3, plane 1: Taking egg from Kitchen

Requirements	Service types	Resources
<p>Q_1: Open X</p> <p>Q_2: Take egg from X ((quantity 1) (state row) (type chicken) (weight [50,200]g))</p> <p>Q_3: Close X</p> <p>Q_4: Wait until the egg warms up ((temp [22,35]°C))</p>	<p>S_1: Egg in X ((quantity ?) (state ?) (time ?) (type ?) (weight ?) (temp ?))</p> <p>S_2: Y can warm egg. ((temp ?))</p>	<p>R_1: Refrigerator ((bind_to X) (quantity 8) (state row) (time 1min) (type chicken) (weight 60g) (temp 2°C))</p> <p>R_2: Cupboard ((bind_to X) (quantity 4) (state row) (time 1min) (type chicken) (weight 60g) (temp 22°C))</p> <p>R_3: Table ((bind_to Y) (time 10 min) (temp 25°C))</p>

Level 3, plane 2: Taking egg from NKitchen

Requirements	Service types	Resources
<p>Q_1: Go to NActor Q_2: Ask NActor for egg ((quantity 1) (state row) (time [0,5]min) (type chicken) (weight [50,200]g)) Q_3: Take the egg home Q_4: Wait until the egg warms up ((temp [22,35]°C))</p>	<p>S_1: NActor can give egg from X ((quantity ?) (state ?) (time ?) (type ?) (weight ?) (temp ?)) S_2: Y can warm egg. ((temp ?))</p>	<p>R_1: NRefrigerator ((bind-to X) (quantity 6) (state row) (time 10min) (type chicken) (weight 60g) (temp 2°C)) R_2: NCupboard ((bind-to X) (quantity 2) (state row) (time 10min) (type chicken) (weight 60g) (temp 22°C)) R_3: NTable ((bind_to Y) (time 10 min) (temp 25°C))</p>

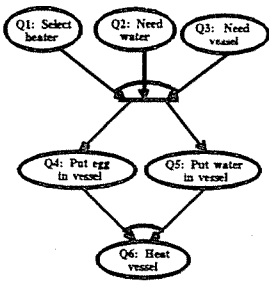
Level 3, plane 3: Taking egg from the environment

Requirements	Service types	Resources
<p>Q_1: You need a purse ((time 1min))</p> <p>Q_2: Select a shop ((type (chicken goose ostrich)))</p> <p>Q_3: Select a vehicle ((*reach [0,15]min))</p> <p>Q_4: You need money in your purse ((quantity [50,∞]Ft) (time [0,10]min))</p> <p>Q_5: Go to shop</p> <p>Q_6: Buy egg ((quantity 1) (state row) (time [0,10]min) (type chicken) (weight [50,200]g) [temp 25°C])</p> <p>Q_7: Take egg home ((time [0,15] min))</p>	<p>S_1: Money in X ((quantity ?) (time ?))</p> <p>S_2: There is a Y shop ((distance ?) (type ?))</p> <p>S_3: There is a Z vehicle ((speed ?) (*reach (/ $Y.distance$ speed)))</p> <p>S_4: You can buy eggs in Y ((quantity ?) (state ?) (type ?) (weight ?) (temp ?))</p>	<p>R_1: Purse ((bind-to X) (quantity 300Ft) (time 1min))</p> <p>R_2: Cash automat ((bind-to X) (quantity 20 000 Ft) (time 10min))</p> <p>R_3: Bank account ((bind-to X) (quantity 200 000 Ft) (time 30min))</p> <p>R_4: Feet ((bind-to Z) (speed 0.1 km/min) (cost zero))</p> <p>R_4: Bicycle ((bind-to Z) (speed 0.5 km/min) (cost low))</p> <p>R_5: Tram ((bind-to Z) (speed 0.3 km/min) (cost medium))</p> <p>R_6: Car ((bind-to Z) (speed 0.7 km/min) (cost high))</p>
 <pre> graph TD Q1((Q1 need purse)) --> Q4((Q4 need money)) Q1 --> Q2((Q2 select shop)) Q1 --> Q3((Q3 select vehicle)) Q4 --> Q5((Q5 go to shop)) Q2 --> Q5 Q3 --> Q5 Q5 --> Q6((Q6 buy egg)) Q6 --> Q7((Q7 take home)) </pre>		

Level 3, plane 3: Taking egg from the environment (continued)

Requirements	Service types	Resources
		<p> R_7: SSmall ((bind-to Y) (distance 0.3 km) (quantity ∞) (state row) (type chicken) (weight 60g) (temp 25°C) </p> <p> R_8: SDep ((bind-to Y) (distance 3 km) (quantity ∞) (state row) (type (chicken goose ostrich)) (weight [50,1000]g) (price [10 50 200]Ft) </p> <p> R_9: SMarkt ((bind-to Y) (distance 10 km) (quantity ∞) (state row) (type (chicken goose)) (weight [50,150]g) (temp 25°C) (price [5,50]Ft) </p>

Level 3, plane 4: Boiling egg in Kitchen

Requirements	Service types	Resources
<p>Q_1: Select a heater ((time 1min) (*can-boil [0,10]min))</p> <p>Q_2: You need water ((quantity [0.5, ∞]l) (time [0,10]min))</p> <p>Q_3: You need a vessel ((capacity [0.5, ∞]l) (max-temp [300, ∞]°C))</p> <p>Q_4: Put egg in vessel</p> <p>Q_5: Put water in vessel</p> <p>Q_6: Heat vessel with heater ((time [3,6]min))</p>	<p>S_1: You have an X heater ((number-of-plates ?) (boil-power ?) (*can-boil (/ Y.capacity boil-power)))</p> <p>S_2: You can take water from Z ((quantity ?) (time ?))</p> <p>S_3: You can take Y vessel from Cupboard ((capacity ?) (max-temp ?))</p>	<p>R_1: Gas heater ((bind-to X) (boil-power 0.2l/min) (time 2min))</p> <p>R_2: Electric heater ((bind-to X) (boil-power 0.1l/min) (time 5min))</p> <p>R_3: Jar ((bind-to Z) (capacity 1.2l) (max-temp 300°C) (time 1min))</p> <p>R_4: Tap ((bind-to Z) (capacity ∞) (time 1min))</p> <p>R_5: Jug ((bind-to Z) (capacity 1l) (time 1min))</p> <p>R_6: Pan ((bind-to Y) (capacity 1l) (max-temp 500°C) (time 1min))</p> <p>R_8: Environment ((bind-to X) (boil-power [0.1,10]l/min) (time 180min)) ((bind-to Y) (capacity [0.1,100]l) (max-temp [50,1200]°C) (time > 50min)) ((bind-to Z) (capacity ∞) (time > 20min))</p>
 <pre> graph TD Q1([Q1: Select heater]) --> Q2([Q2: Need water]) Q1 --> Q3([Q3: Need vessel]) Q2 --> Q4([Q4: Put egg in vessel]) Q2 --> Q5([Q5: Put water in vessel]) Q3 --> Q4 Q3 --> Q5 Q4 --> Q6([Q6: Heat vessel]) Q5 --> Q6 </pre>		

Level 3, plane 4: Boiling egg in Kitchen (continued)

Requirements	Services	Resources
		<p>R_7: Glass pan ((bind-to Y) (capacity 0.8l) (max-temp 350°C) (time 1min))</p> <p>R_8: NKitchen ((bind-to X) (boil-power (0.1 0.2)l/min) (time 20min)) ((bind-to Y) (capacity (0.8 1 1.2)l) (max-temp [300,500]°C) (time > 10min)) ((bind-to Z) (capacity [1, ∞]l) (time > 10min))</p>

Level 3, plane 5 (boiling egg in NKitchen) is similar to 3, plane 4, although the requirement structure is extended by some first additional steps in which the neighbour must be asked to allow us to use her kitchen or to boil us an egg.

The planes of level 4 would describe how to buy heater, vessels or water in the environment. They are similar to level 3, plane 3, which describes how to buy eggs.

Since these planes are not used in the following section in the demonstration example, for the sake of simplicity they are omitted, although they can be easily constructed using the elements of the world and the introduced modelling techniques.

5.3 How the model is used?

In the following the operation of the model is demonstrated by a simple example.

Let us suppose that we like boiled eggs very much and we have one every morning for breakfast. At the beginning there are 12 row eggs in our cupboard (see R_1 at level 2).

Any time we need a boiled egg, the top-level requirement Q (You need boiled egg) is tried to be satisfied at level 1.

At first a prototype service is created (Figure 13) by matching attributes of service type S (Boiled egg) against the attributes of Q and so determining the set of such values for the attributes of S , which can satisfy Q .

```

S: Boiled egg
((quantity 1)
 (state hard)
 (time {0,60}min)
 (type (chicken goose ostrich))
 (weight {50,1000}g)
 (temp {25,60}°C)
 )
  
```

Figure 13: The service prototype generated at level 1.

The next step is the creation of a service s (Figure 14) by binding resource R (the whole

world of egg boiling) to S . s differs from S in the tolerance ranges of certain attributes (e.g. time) and expresses those capabilities, which the system really can provide to satisfy a given requirement.

This attribute mapping sequence is in this case straightforward, because there are no different possibilities to satisfy requirements, there is a single service type and one resource, and these imply that there is only a single instantiation of the level. The importance of the described method (which is in fact the PRA algorithm) will be better demonstrated later.

```
s: (Boiled egg, World of egg boiling)
((quantity 1)
 (state hard)
 (time [5,60]min)
 (type (chicken goose ostrich))
 (weight [50,1000]g)
 (temp [25,60]°C)
 )
```

Figure 14: The service generated from the given prototype

At this point level 2 is activated. In the requirement structure of level 2 there are two alternative possibilities to provide a hard egg: if you have already one (let us say you did not eat it yesterday), you are ready, and if you have only row eggs, you must first boil one and then you get ready. The requirement structure is tried to be satisfied from top down. In the following detailed service prototypes and services will appear only, when understanding would otherwise become very complicated.

Suppose that (because of its apparent simplicity) Q_1 (*You need a hard egg*) is used to generate a prototype of S_1 (*Egg in X*) first. The only important attribute value pair in S_1 for us is (*state hard*), i.e. a hard egg is required. Any of resources R_1 , R_2 or R_3 can provide S_1 , however the "*state*" attribute of all of them has the value "*row*", which cannot be mapped to the value "*hard*" in the prototype of S_1 . It simply means, that we have no hard eggs available.

Next a prototype of S_1 is generated using Q_2 (*You need a row egg*). Service s_1 can be generated creating the (S_1, R_1) service-resource pair by mapping their attributes against each other, where R_1 denotes our own *Kitchen*.

At this point the fault-tolerant nature of the concept can be demonstrated. Let us see what happens when certain resources otherwise providing a given service fail. A simple example for that is the following. Eggs are consumable resources, so each time we use R_1 to get a new egg, the number of eggs decreases by one. If we have used all our eggs in our kitchen, the current value of "*quantity*" in R_1 is 0. Since the prototype of S_1 requires exactly 1 egg, it results that the (S_1, R_1) mapping fails.

Now PRA tries to find another resource, which can provide the service type S_1 , so it will try to use R_2 (*our neighbour's kitchen*) to create service s_1 . This may be successful, if our neighbour has at least one egg. Since to borrow eggs from the neighbourhood is more complicated than just taking some from the refrigerator, it takes more time, that is why it is reasonable to try R_1 first. If R_2 fails we (PRA) can still try to get eggs from the environment, i.e. to go to a shop and buy some.

For the sake of simplicity suppose, that the $s_1 = (S_1, R_1)$ mapping was successful, i.e. there were some eggs to be found in our own kitchen. Since s_1 is a simple service (i.e. it is not decomposed further on lower levels), no other actions must be taken for satisfying Q_2 .

If s_1 is successfully generated, Q_3 (*You must boil the egg*) is used to generate a prototype of S_2 (*You can boil egg in Y*). S_1 is clearly not appropriate for that, since the attribute mapping for (Q_3, S_1) fails because of incompatible attribute names. R_1 be used to generate

service s_2 , which means that our kitchen makes it possible to boil eggs in it. It is important to notice, that R_1 can provide services of types both S_1 and S_2 and these services are *independent*, so it may be possible to boil eggs in our kitchen even if we have currently no eggs and vice versa.

The only compound service is now s_2 , which is decomposed to level 3, plane 4, so this plane must be activated to satisfy Q_3 at a lower level. At level 3, plane 4 the requirement structure is satisfied from top down in the same fashion described by the previous levels. Since the generation of service prototypes and matching them against resources is the same as shown by previous levels, only the requirement-service type-resource associations are given and the satisfaction of the requirement structure is explained.

Requirements Q_1 (*Select a heater*), Q_2 (*You need water*) and Q_3 (*You need a vessel*) can be satisfied in an arbitrary order, though all of them must be satisfied before the satisfaction of any of Q_4 (*Put egg in vessel*) and Q_5 (*Put water in vessel*) can begin. Service type S_1 (*You have an X heater*) is prototyped with Q_1 , then resource R_1 (*Gas heater*) or R_2 (*Electric heater*) can be used to generate service s_1 . In the same manner s_2 is generated from Q_2 by S_2 (*You can take water from Z*) using resources R_3 (*Jar*) or R_4 (*Tap*) or R_5 (*Jug*). Then s_3 is generated from Q_3 by S_3 (*You can take Y vessel*) using resources R_6 (*Pan*) or R_7 (*Glass pan*). When these services are successfully generated, requirements Q_1 , Q_2 and Q_3 are satisfied⁸. The satisfaction of Q_4 (*Put egg in vessel*), Q_5 (*Put water in vessel*) and Q_6 (*heat vessel with heater*) have no attributes, which means that their satisfaction is always possible whenever the *Actor* works properly, or in other words they need no other resources than those previously acquired and the actor itself.

This example demonstrates the *top-down* operation of the method, when a new requirement is initiated at an upper level and it results in a top-down traversal on the decomposition hierarchy finally leading to a working system configuration which satisfies all the requirements at every level or (which we hope will happen very rarely) the failure of the whole system.

The proposed method has however another, *bottom-up* operating mode referred to in this example, when the possibility of missing eggs in our own kitchen was discussed. This is necessary when a working system configuration is set up successfully and must operate continuously for a long time. In such cases it can happen that certain system components fail during operation. This failure is detected at the lowest possible level and tried to be masked. When the masking fails, this fact is forwarded to the upper level as a fault. The upper level tries to resatisfy the requirement structure using *PRA*. This resatisfaction may result in the change of the used services and thus a change in the requirement structure at a lower level.

⁸ Resources R_8 (*the environment*) and R_9 (*the neighbour's kitchen*) can also be used to satisfy requirements Q_1 , Q_2 and Q_3 , though these are compound components, which means that they need further work as their decomposed requirement structures must be also satisfied.

6 RELATIONS TO PREVIOUS WORK

The ideas and results of this report (presented in chapters 4 and 5) stem from two basically different fields of research: fault-tolerant systems and artificial intelligence (AI). If we analyze the properties of these two fields, certain overlappings can be observed between their basic goals, especially at the field of diagnostic and intelligent monitoring systems (see section 3). However there is a huge gap between the two areas. Currently known fault-tolerant solutions are intended for use with digital computer systems and their operation is based upon detailed, though low level information. On the other hand AI methods - though offer much more freedom in representing and handling information - stay at the level of "pure" diagnosis and previous solutions do not tend to cope with fault-tolerant behaviour.

For our current purposes (i.e. to find efficient fault-tolerant solutions for large scale systems, which are not necessarily computer systems) traditional fault-tolerant techniques are not appropriate because of their high overhead and costs and because of the fact that their application is restricted to digital computers (see section 3). From these aspects our approach is novel. For the sake of completeness in the following some preliminary works are mentioned, which - though each of them captures just a particular part of our goals - have in some extent initiated our ideas.

Sztipanovits (1988) generalized the notion of (parametric) adaptivity to *structural adaptivity*. A structurally adaptive system is able to change connections between its components or even incorporate new components (i.e. to change its own operating model and topology) to achieve a better performance when its operating environment changes. Sztipanovits defines the components of structurally adaptive systems and suggests an efficient implementation method based on the object oriented data-flow network concept. The approach shows similarities to *reconfiguration*, so it is highly useful for us, though its original goal is at no means to achieve fault-tolerance and no one has ever tried it to apply for fault-tolerant systems.

Carrasco (1991) presents a hierarchical object oriented language for modelling fault tolerant computer systems. In this article a modelling technique is worked out, which lays at a higher level than traditional techniques. The model is based upon the concept of *components* and *repair teams*, where component states are modified by *failure* and *repair* processes. An explicit construct is used to express *failure propagation*. This approach - just like ours - uses hierarchical decomposition, though its scope is restricted to computer systems and the set of primitives and the modelling approach is basically different from ours.

Crow (1991) extended Reiter's (1987) diagnosis principle to reconfiguration. It is a highly interesting idea, which shows that reconfiguration problems can be generalized through Reiter's theory. The theoretical contents of this article are interesting to us, though this methodology - as Reiter's theory is based upon first order logic - does not assure the representational and algorithmic power, which is required to describe complex systems.

Examples for complex applications near to our goal are the work of Hariri (1992), who presents a framework for architectural support in fault-tolerant distributed systems' design, and the work of Hecht (1991), where a distributed fault-tolerant system for process control is described.

Hariri's method is a design principle for building fault-tolerant open distributed systems, i.e. its scope is restricted to computer networks. This method is based upon *implicit*

redundancy present in large scale distributed systems, a concept to which our approach also strongly relies. The key ideas of this solution are the application of a *two level hierarchical permanent memory structure* to store data about distinct components and a *distributed voting algorithm* to select faulty component candidates and to reconfigure the system. In its methodology Hariri's method is near to the solutions applied in *system level diagnosis* (see section 3 and 4).

Hecht's system is based on an enhanced version of the distributed recovery block method. It has been implemented and integrated into a chemical processing system at the U.S Department of Energy Experimental Breeder Reactor II site at Idaho Falls, ID. This configuration consists of redundant processors interconnected by redundant networks. Under normal circumstances an active node executes a primary version of a control task in parallel with an alternate version executed on a shadow node. This configuration tolerates a broad range of hardware, system software, and application faults, though its operation is based upon extensive hardware redundancy.

7 CONCLUSIONS AND FURTHER WORK

In this report a new modelling method is proposed with the following important potential advantages which distinguish it from previous solutions for achieving fault tolerance:

- it makes it possible to use implicit redundancy in distributed systems to reach fault tolerant operation;
- the amount of information incorporated in diagnostic centers can be kept at an optimal level because of the top-down modelling approach;
- there is no need for performing complete diagnostic procedures, which determine the exact causes of faults and normally are very time consuming, so the proposed procedures can concentrate on the fulfilling of the given set of system requirements and so achieving fault-tolerant operation, and this way high operating speed can be expected;
- it can host different kinds of error detection methods ranging from built in software exceptions and memory management units through sensors or certain kinds of data acquisition components. Since it does not necessitate solutions like master-checkers or watch dog processors, the required hardware overhead can be kept acceptably low.
- it can be applied in systems which were originally not designed to be fault tolerant.

To show that the ideas are viable and the tools for the implementation of such a framework are given, some widespread, efficient and powerful practical methods are mentioned, which exactly fit the requirements of different subparts of the model. A very good summary of these methods is given by Rich (1992).

The basic concept of describing different components of the model is by using sets of attribute-value pairs. Some attributes are direct, some others are derived. Components can be connected to each other through special attributes (see *bind-to* attributes in the example of section 4). These properties can be very well provided by a *semantic network* or *frame system* (Minsky, 1975).

In the model the notion of *attribute structure* and *service-resource* structure appears. The handling of such structures is covered and solved by *constraint satisfaction methods* offering a plenty of different ways for representation and solution methods (Güsgen, 1992).

Attribute mapping, the basic method of satisfying requirements is a special purpose pattern

matching, for which several known methods exist (like indexing, unification (Rich, 1992) or different kinds of the RETE-match algorithm (Forgy, 1982)).

The exact selection and integration of the above mentioned methods is a question of system design which will take place in the following phase of our research project.

The proposed model offers the representational and problem solving power to cope with general problems. Its properties need however further verification by working out the methods in more detail and applying them to practical problems.

The following activities are required:

- The attribute mapping algorithm must be worked out in full details.
- The data exchange between planes must be precisely designed (e.g. how the change of requirement structures is made, and how higher level attribute values are computed from a lower level).
- A classification of typical resources is necessary to determine their properties and nature. It is important to see what kind of autonomous testing and diagnostic tasks can and must be performed.
- Benchmark systems (like simpler industrial technologies or computer systems) must be found to test the abilities of the concepts in real situations and to refine the applied methods according to the gained experiences.
- The algorithmic and representational complexity of the methods should be also estimated.

REFERENCES

- Avizienis, A. (1982):
The four-universe information system model for the study of fault tolerance, *Proceedings of the 12th ISFTC*, 1982, pp.6-13.
- Dahl, O. J. - Dijkstra, E. W. - Hoare, C. A. R. (1976):
Structured programming, *Academic press, London-New York*, 1976.
- Carrasco, J. A. (1991):
Hierarchical object oriented modelling of fault tolerant computer systems, *Proc. of COMPEURO 1991*, pp. 452-456
- Dal Cin, M. - Grygier, A. - Hessenauer, H. - Hildebrand, U. - Pataricza, A. (1993):
Fault Tolerance in Distributed Shared Memory Multiprocessors, in A. Bode, M. Dal Cin (eds.), *Parallel Computer Architectures - Theory, Hardware, Software, Applications, Springer Lecture Notes in Computer Sciences 732*, 1993, pp. 31-48
- Chandrasekaran B. - Punch, W. F. (1988):
Hierarchical classification: its usefulness for diagnosis and sensor validation, *IEEE J. on Selected Areas in Communications*, vol. 6, no. 5, June 1988, pp. 884-891.
- Crow, J. - Rushby, J. (1991):
Model-based Reconfiguration: Toward an Integration with diagnosis, *Proc. of AAAI 1991*, pp. 836-841
- Davis, R. (1984):
Diagnostic reasoning based on structure and behaviour, *Artificial Intelligence* vol. 24, pp. 347-410
- Dechter, R. - Pearl, J. (1988):
Tree-clustering schemes for constraint processing, *IJCAI 1988*, pp. 150-154
- Deconinck, G. - Vounckx, J. - Lauwereins, R. (1992):
Survey of Checkpointing and Rollback Techniques, *Research report in the frame of ESPRIT 6731 "A Practical Approach to Fault-tolerant Massively Parallel Systems"*, Katholieke Universiteit Leuven, 1992.
- Eifert, J. B. - Shen, J. P. (1984):
Processor Monitoring Using Asynchronous Signed Instruction Streams, *Proc. 14th Int. Symposium on Fault Tolerant Computing (FTCS-14)*, pp. 394-399, 1984
- Fabre, J.-C. (1991):
Improving fault-tolerance in distributed systems: the saturation approach, *Proc. of COMPEURO 1991*, pp. 634-641
- Fujiwara, H. (1985):
Logic testing and design for testability, *MIT Press, Cambridge, MA*, 1985.
- Forgy, C. L. (1982):
A fast algorithm for the many pattern/many object pattern matching problem, *Artificial Intelligence* Vol. 19, pp. 17-37
- Güsgen, H. W. - Hertzberg, J. (1992):
A perspective of constraint-based reasoning (An introductory tutorial), *Springer Verlag, Berlin* 1992.
- Hachiga, A. (1993):
The Concepts and Technologies of Dependable and Real-time Computer Systems for

- Shinkansen Train Control". In H. Kopetz, Y. Kakuda (eds): *Responsive Computer Systems*. Springer Verlag, *Dependable Computing and Fault-Tolerant Systems* Vol. 7, pp. 225-252
- Hariri, S. - Choudhary, A. (1992):
Architectural support for designing fault tolerant open distributed systems, *IEEE Computer*, June 1992, pp. 50-62
- Hecht, M. - Agron, J. - Hecht, H. - Kim, K. H. (1991):
A distributed fault-tolerant architecture for nuclear reactor and other critical process control applications, *21th Int. IEEE Symp. on Fault-tolerant Computing Systems* 1991, pp. 462-469.
- Johnson, B. W. (1984):
Fault-tolerant microprocessor based systems, *IEEE Micro*, Vol. 4, No. 6. December 1984, pp. 6-21
- Johnson, B. W. (1988):
Design and Analysis of Fault Tolerant Digital Systems, *Addison-Wesley Publishing Company*, 1988
- Johnson, B. W. (1989):
Design and application of fault-tolerant systems for industrial applications, *Proc. of FTCS 1989*, pp. 57-73
- Kim, K. H. (1989):
Approaches to system level fault tolerance in distributed real-time computer systems, *Proc. of FTCS 1989*, pp. 268-282.
- Kime, C. R. (1985):
System Diagnosis. In *Fault-Tolerant Computing: Theory and Techniques*, Prentice-Hall, New York, 1985, pp. 577-623
- Kirschen, D. S. - Wollenberg, B. F. (1992):
Intelligent alarm processing in power systems, *Proc. of the IEEE*, Vol. 80, No. 5, May 1992, pp. 663-672
- Kohari, Z. (1978):
Switching and finite automata theory, *McGraw-Hill New York*, 1978.
- Laprie, J. C. (1985):
Dependable Computing and Fault Tolerance: Concepts and Terminology, *Proceedings of the 15th ISFTC*, 1985, pp.2-11.
- Laprie, J. C. et al (1990):
Dependability: Basic Concepts and Terminology, *IFIP WG Dependable Computing and Fault Tolerance*, October 1990
- Lee, P. - Anderson, T. (1990):
Fault tolerance. Principles and practice, *Springer Verlag, Wien*, 1990.
- Lu, D. J. (1982):
Watchdog Processors and Structural Integrity Checking, *IEEE Trans. on Comp.*, Vol. 31/7, pp. 681-685
- Mahmood, A. - McCluskey, E. J. (1988):
Concurrent Error Detection Using Watchdog Processors - A Survey, *IEEE Trans. on Comp.*, Vol. 37/2, pp. 160-174
- Malek, M. (1991a):
Responsive systems: a marriage between real-time and fault tolerance, *Proc. of FTCS 1991*, pp. 1-17

- Michel, T. - Leveugle, R. - Saucier, G. (1991b):
A New Approach to Control Flow Checking Without Program Modification, *Proc. 21st Int. Symposium on Fault Tolerant Computing (FTCS-21)*, pp. 334-341
- Michel, E. - Hohl, W. (1991):
Concurrent Error Detection Using Watchdog Processors in the Multiprocessor System MEMSY, *Informatik-Fachberichte 283, Fault Tolerant Computing Systems, Springer Verlag Berlin*, 1991, pp. 54-64
- Mittra, S. S. (1988):
Structured techniques of system analysis, design and implementation, *John Wiley and Sons, New York*, 1988
- Minsky, M. (1975):
A framework for representing knowledge, *The Psychology of Computer Vision, P. Winston (ed.), McGraw-Hill, New York*, 1975
- Padalkar, S. - Karsai, G. - Biegl, C. - Sztipanovits J. (1991):
Real-time fault diagnostics, *IEEE Expert*, June, 1991, pp. 75-85
- Pataricza, A. - Majzik, I. - Hohl, W. - Hoenig, J. (1993):
Watchdog Processors in Parallel Systems, *EUROMICRO'93, 19th Symposium on Microprocessing and Microprogramming, Barcelona*, 1993
- Patel, J. H. - Fung, L. Y. (1982):
Concurrent error detection in ALUs by recomputing with shifted operands, *IEEE Trans. on Computing*, Vol. C-31, No. 7, July 1982, pp. 589-595
- Reiter, R. (1987):
A theory of diagnosis from first principles, *Artificial Intelligence*, Vol. 32, pp. 57-95
- Reynolds, D. A. - Metz, G. (1978):
Fault detection capabilities of alternating logic, *IEEE Trans. on Computers*, Vol. C-27, No. 12, December 1978, pp. 1093-1098
- Rich, A. (1992):
Artificial intelligence, *McGraw Hill*, 1992
- Shen, J. P. - Tomas, S. P. (1987):
A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems, *Microprocessing and Microprogramming 20, North-Holland*, pp. 249-269
- Sridhar, T. - Thatte, S. M. (1982):
Concurrent Checking of Program Flow in VLSI Processors, *Proc. 1982 Int. Test Conf.*, pp. 191-199
- Schwabl, W. - Reisinger, J. - Grünsteidl, G. (1989):
A Survey of MARS, *Research Report Nr. 16/89, Technische Universität Wien*
- Sztipanovits, J. (1988):
Toward structural adaptivity, *Proc. of ISCAS 1988*, Vol. III, pp. 2359-2362
- Zadeh, L. A. (1965):
Fuzzy sets, *Information and Control*, Vol. 8, pp. 338-353

