Efficient System-Level Fault Diagnosis of Large Multiprocessor Systems

by

Tamás Bartha
M.Sc., Budapest University of Technology and Economics, 1993

A thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Faculty of Electrical Engineering and Computer Science
Budapest University of Technology and Economics

Abstract

Efficient System-Level Fault Diagnosis of
Large Multiprocessor Systems

Tamás Bartha, M.Sc.
Budapest University of Technology and Economics, 2000

Supervisor: Prof. Endre Selényi, D.Sc.

This document presents new results in different areas of multiprocessor fault tolerance. The first main topic of research studied is fault diagnosis of large parallel computers. Both the centralized and distributed approach to fault diagnosis are considered. In the framework of centralized diagnosis a novel probabilistic fault diagnostic methodology, called *local information diagnosis* (LID), is introduced. The LID approach combines the diagnostic power of the generalized test invalidation model with the time efficiency of probabilistic methods. The developed methodology is applied to create three classes of diagnostic algorithms: *limited inference*, *limited information*, and *scalar* methods. The representation and handling of the diagnostic information is different in the above classes, providing a diagnostic accuracy/time complexity trade-off.

LID diagnostic algorithms employ the *generalized test invalidation model* to handle a wide range of testing arrangements and failure behavior. They use parameterized inference rules to obtain one-step diagnostic implications conforming to the fault hypotheses and the actual syndrome. Scalar methods utilize only the one-step implications, thus achieving low time and space complexity together with good diagnostic accuracy. Limited inference and limited information methods generate multiple-step diagnostic implications by transitively propagating one-step implications. They provide significantly better diagnostic accuracy than either scalar or other existing probabilistic algorithms. Naturally, they have higher time and space complexity than scalar methods, but their execution time and memory requirements are still quite adequate for practical application. Different techniques of using the diagnostic implications in the fault classification process are studied in detail. For the purpose of fault classification based on diagnostic implications three heuristic methods having different characteristics are developed and analyzed.

The thesis presents five new probabilistic fault diagnostic algorithms created on the basis of the local information diagnosis methodology. The diagnostic and computational performance of the algorithms are compared using both theoretical analysis and measurement techniques. The measurement result confirm that the new algorithms have better diagnostic accuracy than existing methods, while their time complexity is acceptable for the application in large-scale computing systems. The results also demonstrate that the developed methods are

- flexible
- applicable in non-symmetric and heterogeneous test invalidation situations,
- work in arbitrary system topology, and
- can improve the accuracy of the diagnosis by iteratively extending the amount of diagnostic information extracted from the syndrome.

The thesis also introduces new results in the area of distributed fault diagnosis. An innovative distributed diagnostic algorithm is described, which lays special emphasis on the low induced diagnostic message load during normal operation. The algorithm schedules the higher communication load during the initialization of the system, when the performance requirements are not strict. During the *initial* stage a system-wide local diagnostic image is generated in each processing element. In the later *working* stage this basic diagnosis is updated incrementally. This diagnostic strategy also supports the *event-oriented* dissemination of the testing information, therefore new messages are only transmitted if they contain relevant information. The diagnostic procedures are supplemented by two communication protocols, tailored to the characteristics of the two stages of diagnosis.

The third topic of research presented is a backward error recovery scheme developed for a practical system: the APEmille parallel computer. The created recovery scheme unifies the error recovery mechanisms selected for the different components on the basis of a component-level failure semantics model. It is argued that the two main architectural units require distinct recovery approaches, and the main features that determine the applicable solutions are explained in detail. The selection of component-level recovery mechanisms extends to single and multiple failure occurrences in the considered resource. As part of the recovery scheme a prospective implementation of a stable storage is given, which fits into the architecture of the APEmille computer and does not require the inclusion of external hardware components.

Dedication

To my parents, for their love and support

Acknowledgements

This thesis would never have been completed without the support and encouragement of my advisor, Endre Selényi. His insightful comments and constructive criticism taught me to concentrate on the essential aspects of the presented work and helped me to overcome my constant urge of exploring my new ideas without investigating into the depth of old ones.

I am also indebted to András Pataricza, who is a guiding influence of my professional and personal life. I thank him for the possibility to have research opportunities abroad, which helped me to get acquainted with new approaches and to discover new directions.

I express my gratitude to Professor József Bokor for providing me with the ideal conditions to do my work. I would like to thank Professor Piero Maestrini, and Professor Mario Dal Cin for hosting me in their research institutes and making it possible for me to participate in the research projects.

I am grateful to all of my colleagues who collaborated me on various topics or simply shared their opinion of my work with me. In particular, I would like to mention István Majzik, Jörn Altmann, and Stefano Chessa, whose friendship is one of the most important results of the years spent with preparing this thesis.

Contents

List of Tables

List of Figures

List of Algorithms

Motivation

The rapid development of electronic device manufacturing technology influences the design methodology of high-performance computing systems to a large extent. The present tendency in the price to performance ratio of complex, highly integrated components (like processors, memory chips, and controllers) expedites the use of parallel processing as the primary architectural method for increasing the computational capacity. As a consequence, distributed multiprocessor systems appeared in all application areas of computer technology from networking to financial transaction processing. *Massively parallel systems*, capable of incorporating as much as several thousand processing elements, represent the most comprehensive utilization of distributed computing. Nowadays *workstation clusters* start to gain increasing popularity especially in universities and research institutes. These systems require only a high-speed local area network, several low-cost commercial desktop computers, and some (public domain or self-developed) support software which are usually already available in these institutes. Thereby the supercomputer performance of large parallel machines becomes accessible to a growing set of users.

At the same time the requirements of computer system dependability have also increased accordingly, particularly in those application areas, where the continuous error-free operation has serious financial and moral consequences. Such areas are space engineering and research, military technology, medicine, and energetics. While electronic circuits became more reliable with the advances in manufacturing technology, the probability of error occurrence remained significant due to the large amount of built-in components. Since faults will not ever be completely avoidable, computing environments must be equipped with the ability to retain their integrity as experienced by the user in spite of low-level hardware failures. The assurance of system availability is the task of *fault tolerance.*

There are two main approaches to provide a reliable computing service. A widely used fault tolerance technique is *error compensation.* This approach uses redundant functional units (duplicates, triple-modular redundancy, etc.) to compensate for the loss of the failed devices. The advantage of the approach is its relative simplicity and effectiveness: faults are tolerated transparently, without system down-time or performance degradation. On the other hand, error masking is very expensive: the production cost is at least doubled, not mentioning other important factors like physical dimensions, power consumption, and heat dissipation that also grow accordingly.

Rather than depending on spare units, the other approach called *error recovery* bases on the inherent redundancy of multiprocessor systems. The underlying idea of this approach is to (physically or logically) remove the failed resources and redistribute the user tasks from the failed units to the remaining fault-free processing elements. Naturally, the decrease of the system resources and the increase of the executed tasks per processor result in a higher overall load and thus computational performance drops. Yet, system availability can be maintained with only a small set of inexpensive additional devices (error detection hardware, switches to isolate the faulty components). Therefore, error recovery is more complex, but less cost-intensive fault tolerance technique than error masking. *System-level fault diagnosis* is an important part of the error recovery process. Its purpose is to identify the failed components using the results of high-level tests.

The theory of system-level fault diagnosis was founded in 1967 by the introduction of the centralized deterministic fault diagnosis concept [1] of Preparata et al. The name of the concept originates from the high-level modeling of the units constituting the multiprocessor system and their relationship. The diagnostic model comprises processing elements and test connections between them. The testing job is accomplished by the processors of the distributed system themselves, without additional or external tester devices. The fault state of the modeled devices is typically of pass/fail or GO/NO GO type. This simplified view, however, is not disadvantageous within the field of application, as the purpose of diagnosis is localization of faults in the component level and not a detailed report of the underlying hardware phenomenon that caused the failures.

The methodology of centralized deterministic fault diagnosis remained the dominant research area for more than ten years, but the advance of computer architecture created new requirements also for diagnostic algorithms. Changes in the field necessitated the gradual extension of the Preparata model, so new diagnostic approaches appeared to replace the obsolete concepts. New *test invalidation models* have been developed to describe different relationships between faulty and fault-free units. Parallel to the progress in system organization and complexity increase machines became *heterogeneous*, therefore only a *generalized test invalidation* model could capture the new features. The limitations on system structure and admissible fault sets required for complete (classifying every unit) and correct (reflecting the real fault situation) diagnosis made the early algorithms inapplicable in cases where these conditions could not be fulfilled.

The *probabilistic diagnostic* approach was designed to function in large multiprocessor systems even when the limits of deterministic diagnosability are exceeded. The expansion of the applicability of system-level diagnosis is at the cost of diagnostic accuracy: probabilistic algorithms cannot guarantee the correct and complete classification of each system component, diagnostic mistakes have a non-zero—albeit quite low—probability. Likewise the extensive distributed environments proved that the centralized control of diagnosis causes potential concerns in availability and performance. This problem can be solved by implementing diagnosis with decentralized control, the topic is thoroughly investigated by the theory of *distributed system-level diagnosis.* Of course numerous other additions and improvements have been devised in the framework of system-level fault diagnosis, the above mentioned approaches are studied in detail within this thesis.

In general the methods existing in the literature contain only a certain aspect of the mentioned extensions. There exists a deterministic algorithm that can handle heterogeneous systems on the basis of the generalized test invalidation model, but that incorporates the limiting conditions of deterministic diagnosability, and is not appropriate for use in massively parallel computers due to its complexity. There exist simple and effective probabilistic algorithms that use only a small portion of the total diagnostic information available in the syndrome and leave the differences of components out of consideration, thus do not estimate the fault state of units very precisely.

Nevertheless, large computing environments usually use many highly integrated communication, control, and monitoring devices. These supplemental devices have a complexity comparable to that of processing elements, therefore it is necessary to involve them in the diagnostic process. However, in such cases the system structure is no longer homogeneous, the diagnostic model must handle variety in the behavior of faulty units. The employed techniques must have small time complexity and low resource usage to handle the huge set of test results collected from the numerous processing elements. The traditional diagnostic t-limit and similar restrictive conditions required for deterministic diagnosis are unmaintainable due to the large dimensions but low local complexity of these systems.

A special class of multiprocessor systems, the massively parallel computers have other important properties that can (and need to) be utilized in diagnosis. These machines have regular physical and interconnection structure in order to retain scalability. The system complexity is small and constant in the local environment of the processing elements. At the same time the major part of relevant diagnostic information originates in this local area. The computational requirements of the diagnostic algorithm can therefore be significantly reduced by examining only the limited environment of a certain unit during its fault classification, instead of the whole system. This is the basic idea of *local information diagnosis.*

Centralized fault diagnosis is an integral part of a larger procedure, the *error removal* type of fault tolerance approach. Its results are utilized by each subsequent steps: error confinement, damage assessment, the reconfiguration of the available resources and task redistribution, as well as the recovery from errors by the restoration of a consistent valid global system state. These steps depend on each other, therefore they must be designed and implemented in a unified framework. Although the individual elements of the error recovery process can be realized with the help of methods and ideas taken from the literature, the structure of the recovery process itself is completely application-specific, and so it requires customized methodology. By participating in the APEmille project which aims at the development of a parallel supercomputer, the author was given the opportunity to formulate such an error recovery scheme.

Although the architecture of the APEmille computer is particularly suitable for the application of centralized fault diagnosis, other machines designed according to different requirements are not capable of employing centralized methods. Typically the completely distributed large scale systems containing many autonomous units belong to this category. The use of central units or resources conflicts with the objectives of these computers. Consequently, the need for adapting the diagnostic procedure to the distributed structure of the system came up already in the eighties. The methodology of *distributed fault tolerance* by Kuhl and Reddy assigned the task of fault diagnosis to the processing elements of the distributed system themselves. The purpose of the diagnostic algorithm is to guarantee the uniformity and validity of the created diagnostic image in the case of fault-free units. The isolation of the faulty processors is also achieved in a distributed manner: the fault-free units refuse to cooperate with the faulty ones based on the common diagnostic image, thus boycotting them.

In the recent years many distributed fault diagnostic algorithms have been formulated, indicating the importance of this research area. These methods periodically execute tests in a static or adaptive assignment, then disseminate the local test results to every unit by reliable *multicast* protocols. The processors receive every test result and perform the analysis of the collected global syndrome individually. Existing methods, however, do not give enough consideration to the message volume caused by the distribution of local test results and the actualization of the adaptive testing arrangement. The possibility of improvement is motivated by the technique of *incremental checkpointing*. Following its basic mechanism the processors can create an initial global diagnostic image and then update only the differences caused by the occurring new diagnostic events (failure, repair). This way the transmission of the diagnostic messages can also be scheduled in an event-oriented way. Therefore only the creation of the initial diagnostic image generates high message volume, which occurs during the start-up phase of the system and thus does not imply additional load during the execution of user applications.

Contribution

The thesis presents new research results in three main areas of multiprocessor fault tolerance: *centralized fault diagnosis*, *distributed fault diagnosis*, and *backward error recovery* of parallel computing systems. This section summarizes the new results of our work and refers to those parts of the document where the results are described in detail.

Contribution 1. *We have developed a new probabilistic diagnostic methodology based on the counting of one-step diagnostic implications in the implication chains supporting a given fault hypothesis. The new approach employs the generalized test invalidation model (in contrast with the probabilistic algorithms known in the literature). This way it is able to identify the failed system components handling the additional diagnostic information inherent to systems having homogeneous non-symmetric and heterogeneous test invalidation [2]. We have defined probabilistic diagnostic algorithms according to the new methodology [3]. We proved, that the developed algorithms have linear time and space complexity, therefore they are particularly suitable for application in large scale systems [4]. The diagnostic methodology and the corresponding new results are introduced in Sections 2.1 and 2.1.3.*

(a) We have defined the *inference graph*, which represents the one- (and multiple-) step implications extracted from the syndrome on the basis of the generalized model of test invalidation. We gave a probabilistic decision procedure for the classification of the unit fault states [5]. The decision is based on the number of edges in the directed edge sequences ending at a certain node of the inference graph. This quantity is equivalent to the number of diagnostic implications supporting the fault hypothesis corresponding to the given node. The decision making method assumes that fault-free tester units dominate the local environment of the diagnosed unit. Contrasted to existing probabilistic algorithms this new diagnostic approach considers the information originated not only the direct neighbors, but (to a limited depth) at the farther units as well. The utilization of this additional information reflects in improved diagnostic accuracy. (The definition of the inference graph can be found in Section 1.2.4, while the different variants of the probabilistic decision mechanism are described in Sections 2.1.2 and 2.1.3.)

(b) We have developed an algorithm for the efficient (although distorted) estimation of the basic quantities of the above probabilistic decision making method. The algorithm uses two implication counters to compute the number of elements in the implication chains supporting the fault-free/faulty states of units. Therefore, it is referred to as the

CIP-2 (*Count Inference Paths*) algorithm [3]. The counter values are determined in multiple iteration step, using only the counters of the neighbor units, thus achieving linear time and space complexity. (The CIP-2 algorithm is one of the scalar LID methods introduced in Section 2.1.3. The theoretical values of time complexity can be found in Section 2.2, while the practical evaluation is presented in Section C.2.)

(c) We have refined the CIP-2 algorithm using four implication counters. The new algorithm, called CIP-4, can differentiate the implication chains not only by their endpoints, but also by their start points. We have modified the probabilistic decision making procedure to consider only implications drawn from the fault-free states of units. This way a better estimation of the fault hypothesis likelihood was obtained and diagnostic accuracy has been improved compared to the CIP-2 algorithm. (The CIP-4 algorithm is the second scalar LID method described in Section 2.1.3. The comparison of the two CIP algorithms can be found in Section C.2.)

The underlying idea of the developed *local information diagnostic methodology* is that in the case of practical fault set types there is enough information available in the local environment of units to determine the quantities used in the probabilistic decision making procedure with sufficient accuracy. Existing *deterministic* algorithms typically classify the fault states taking into consideration all of the units in the system. On the other hand, probabilistic algorithms examine only the direct neighbors of the diagnosed unit. Local information diagnosis creates a compromise between these two approaches. The information used in the LID diagnostic process is gathered from a limited, but wider range environment than the direct neighborhood. Therefore, we could derive algorithms with smaller time and space complexity than the existing deterministic methods, while achieving better diagnostic accuracy than existing probabilistic techniques.

For statistical measurement of the CIP algorithm properties we implemented a general-purpose simulation environment. The simulation environment is capable of the verification of each centralized probabilistic diagnostic algorithms developed in the framework of this thesis, as well as the existing centralized algorithms known in the literature. (The description of the simulation environment is not covered by this document due to the many options and detailed measurement results the simulator provides, a brief user's guide is contained in [6].) Both the CIP-2 and CIP-4 algorithms have been implemented within the simulation environment, and their diagnostic properties were studied. We have confirmed that the CIP-4 algorithm provides better diagnostic accuracy than the CIP-2 algorithm [4]. Both algorithms were proved to correctly identify the fault state of those units which can be surely classified based on the syndrome (provided they are run in an adequate number of iterations) [3]. (These theoretical results are explained in Section 2.2, the correctness proof is provided by Theorem 2.2.9.)

Contribution 2. *We have developed a new probabilistic diagnostic method based on the limited transitive propagation of one-step diagnostic implications. The method consists of two subsequent phases. In the first phase the one-step diagnostic implications are collected* *and they are transitively propagated in a limited depth [7]. The employed data representation includes the entire set of implications, therefore the method is able to extract the complete diagnostic information contained in the syndrome (provided it is executed in an appropriate number of iterations). In the second phase the generated implication set is processed using heuristic fault classification rules [8]. Procedures for both phases of the new diagnostic method have been devised. Using the created procedures we have formulated probabilistic diagnostic algorithms. The developed algorithms were confirmed to have better diagnostic properties than the CIP-2 and CIP-4 algorithms, as well as the probabilistic algorithms known in the literature. The improvement of the diagnostic accuracy comes on the expense of the increase in time and space complexity [4]. The description of the new diagnostic method and the procedures devised for the different phases can be found in Sections 2.1.1 and 2.1.2.*

(a) We have devised a method for the limited-length transitive propagation of diagnostic implications based on matrix multiplication. The method, called the LMIM (*Limited Multiplication of Inference Matrix*) algorithm, obtains the multiple-step implication set used in probabilistic diagnosis by computing a given power of the implication matrix [9]. The time complexity of the method is $O(n^3)$ in the function of the system size, which is determined by the employed matrix multiplication. (The LMIM algorithm is one of the limited inference methods defined in Section 2.1.1.)

(b) We have developed a new method for limited-length transitive propagation of diagnostic implications, which generates a diagnostically equivalent implication set to the matrix multiplication. The new method, called the DIL (*Distribution of Inference Lists*) algorithm, computes the partial implication set used in local information diagnosis with less time complexity ($O(n^2)$ instead of $O(n^3)$) than the LMIM algorithm [2]. The reduction of the time complexity is achieved by realizing the transitive propagation in multiple iterations, using solely the implication sets of the direct neighbor units. (The the DIL algorithm is the other limited inference method described in Section 2.1.1.)

(c) We have developed a new method for limited-length transitive propagation of diagnostic implications based on the transitive closure of diagnostic implications in a reduced environment. The method has linear time and space complexity in the function of system size. The time complexity depends on the extent of local environment considered in diagnosis at a rate of $O(\nu_k^3)$. In large systems and for small considered environments the new method, called the LTC (*Local Transitive Closure*) algorithm, computes the partial implication set used in local information diagnosis more efficiently than the algorithms known in the literature [2]. (The LTC algorithm is a limited information method presented in Section 2.1.1.)

(d) We have formulated three new heuristic methods for the diagnostic analysis of the generated partial implication set. The *Majority* heuristic classifies the fault state of the diagnosed unit according to a majority decision on the implications drawn from the fault-free states of the neighbor units. The *Election* heuristic performs the diagnostic task sequentially, by selecting the most likely faulty unit in each iteration step. The Clique heuristic groups units according to the identical and opposite relationship between unit fault hypotheses. Then, it selects the units classified as fault-free by finding the largest group of units having identical and consistent fault states [8]. (The three fault classification heuristics are defined in Section 2.1.2.)

The one-step diagnostic implications obtained from the system based on the actual syndrome can be extended into multiple-step implications utilizing the transitive property. The adaptation of the local information diagnostic methodology in this framework means the limitation of the transitive propagation. We have given two different approaches to solving the problem of limited propagation of diagnostic inferences. The *limited inference* (LMIM, DIL) approach performs the transitive propagation of the whole implication set in a restricted depth. The *limited information* (LTC) approach executes the transitive closure of an implication set generated from a reduced environment [2].

We have shown that the limited inference and limited information approaches generate identical implication sets (though possibly in a different number of iterations). Thus, they are equivalent from the *diagnostic viewpoint.* (This statement is supported by Theorems 2.2.2, 2.2.4, and 2.2.7 given in Section 2.2.) We have demonstrated by simulation experiments, that the computation of the transitively extended diagnostic implication set by the matrix multiplication technique (known in the literature and employed by the LMIM algorithm) performs many redundant operations in large, regular topology multiprocessor systems. The devised alternative methods (DIL, LTC) have much better suited time complexity for practical applications [4].

The formulated heuristic fault classification methods are essentially the applications of the probabilistic decision making procedure defined in Contribution 1. We have created new probabilistic diagnostic algorithms combining the LMIM, DIL, and LTC implication propagation methods and the Majority, Election, and Clique fault classification heuristics. We have appointed the characteristics affecting the diagnostic accuracy of heuristics. With the help of the simulation environment the developed fault classification heuristics were compared to each other, and to several probabilistic diagnostic algorithms taken from the literature [8]. The measurements have confirmed that the new algorithms are better than the existing methods in terms of the selected characteristics up to one order of magnitude. The improvement is particularly large in systems with non-symmetric and heterogeneous test invalidation [4]. (The measurement activities and the obtained results for the developed and existing algorithms are described in detail in Appendix C.)

Contribution 3. *We have developed a new, general-purpose distributed diagnostic algorithm. The new algorithm combines the static and event-oriented diagnostic approaches existing separately in known algorithms, in order to reduce the induced message load [10]. The algorithm consists of two main stages. The initial stage is executed in the startup period shortly after the system was switched on. Its main purpose is to determine the actual fault state of the whole system. The second, working stage is active during the execution of user* *applications. Its objective is to handle the occurring failure events with smaller diagnostic message overhead compared to the methods known in the literature [11]. The developed distributed fault diagnostic algorithm, including the two operating stages, is presented in Section 3.2.*

(a) We have defined a diagnostic procedure for the *initial stage* which can quickly produce a consistent global diagnostic image in each fault-free processor. The procedure generates a relatively high message load, however, in the startup period the performance requirements are not strict. On the other hand, the time complexity of the procedure is not adversely affected by the existence of a large number of faulty units (contrasted to the known event-oriented algorithms) [12].

(b) We have defined a diagnostic procedure for the *working stage* which analyzes only the changes in the syndrome incrementally, compared to the global diagnostic image created in the initial stage. The distribution of the testing information is accomplished by an event-oriented mechanism in order to decrease the transmitted message volume. Diagnostic messages are started only in the case of detected new diagnostic events (failure, repair), therefore the algorithm influences the system operation during user application execution at a minimum level [12].

(c) We have determined those situations, in which the incremental approach used in the working stage may lead to an inconsistent diagnostic image. Such situations are handled by returning to the initial stage of the algorithm [11]. (The operation of the initial and working stages, and the conditions controlling the alternation between them are detailed in Section 3.2.1.)

Due to the machine-specific nature of distributed diagnosis the message communication protocols constitute an integral part of the diagnostic algorithm. The two stages of the developed distributed algorithm require different protocols. We have defined an effective, but tightly scheduled synchronous communication protocol for use in the initial stage. (The synchronous communication protocol is specified in Section D.1.) For the purposes of the working stage we have developed an asynchronous communication protocol in order to avoid deadlocks and scheduling problems. The asynchronous protocol has been adopted to the two-dimensional mesh topology of the Parsytec GCel machine. This refined protocol generates nearly optimum number of messages even in the presence of faulty units in the system. (The asynchronous communication protocol is outlined in Section D.2.) The operation of the developed distributed diagnostic algorithm and the employed communication protocols was verified by an experimental implementation in the Parsytec GCel system. (Some details of the practical implementation are covered by Section 3.2.2. Other particulars, not included in this document for the sake of conciseness, can be found in [13].)

We have studied the possibility and requirements of the application of local information diagnostic algorithms in distributed environments were studied. We gave methods for integration of the limited transitive propagation based probabilistic diagnostic algorithms into the developed distributed algorithm. We have examined the possibility of further decreasing the transmitted message volume by limiting the area of message dissemination and showed the effect of the reduced testing information on diagnostic accuracy. (The adaptation of LID algorithms to the distributed environment, and the influence of testing information reduction are presented in Section 3.3.)

Contribution 4. *APEmille is a Single Instruction Multiple Data parallel computer prepared as a joint collaboration of several European research institutes. We have developed a comprehensive backward error recovery mechanism that forms a part of the fault-tolerant architecture of the APEmille machine [14]. We have created a component-level failure model of the system, and devised the state saving and state restoration procedures for each component based on this model. In the procedures separate methods were formulated for the case of single and multiple failures [15]. The results concerning the developed recovery mechanism are summarized in Sections 4.3 and 4.4.*

(a) The two main functional units of the APEmille system, the *processing engine* (SIMD part) and the *supervisor computers* (MIMD part), must be treated differently with regard to the recovery mechanism. We have devised separate recovery schemes for both parts, taking into consideration the employed diagnostic algorithm and the planned maintenance strategy [16]. (The recovery scheme for the SIMD part is given in Section 4.4.1, while for the MIMD part it can be found in Section 4.4.2.)

(b) We have developed a prospective implementation of a stable storage, used in the recovery mechanism for reliable recording of state information [17]. The proposed implementation utilizes only the available hardware resources of the system. (The structure of the developed stable storage is described in Appendix E.)

The results corresponding to the contribution are the first steps in the realization of the APEmille backward error recovery mechanism. As this work is not preceded by other research within the framework of the APEmille project, the author had to solve problems outside of the scope of recovery, like the formulation of a simplified, component-level failure semantics model for APEmille (which is included in Section 4.3). The work on the recovery mechanism started at an advanced stage of the project, when it was no longer feasible to suggest modifications in the hardware design. Therefore, one of the requirements during the development of the state saving and state restoration procedures was to employ only the available hardware resources.

Although the diagnostic algorithm, which constitutes a part of the recovery mechanism, has already been completed by the time we started working on the topic of recovery, the APEmille computer is also an appropriate platform for the application of local information diagnosis. Thus, we examined the possibility and conditions of integration of the LID probabilistic diagnostic methods in the fault tolerance of APEmille. (The integration problems and solutions are explained in Section 4.5.)

Chapter 1

Introduction

This chapter introduces the principles and means of fault-tolerant computing with respect to multiprocessor fault tolerance. The basic notions and definitions are presented through the *dependability concept* of Laprie [18]. A set of selected centralized and distributed fault diagnostic algorithms are outlined to illustrate the main problems and existing solutions to the task of fault diagnosis. Finally, the concept of generalized test invalidation and its use for system-level fault diagnosis is described.

1.1 Overview of multiprocessor fault tolerance

Computer system *dependability* is the quality of the delivered service such that reliance can justifiably be placed on this service. The delivered service is an abstraction of the system behavior as it is perceived by other systems (its users) interacting with the considered system. This abstraction is highly dependent on the application, for example *time* plays an important role in the abstraction, as the time granularity for the system and its users are generally different.

1.1.1 Dependability impairments

The *impairments* of dependability are undesired circumstances causing or resulting the lack of dependability, whereby reliance cannot be justifiably placed on the service any more. Reasons, causing the loss of dependability manifest in the forms of faults, errors, and failures:

1. a *fault* is the cause (in its phenomenological sense) of an error,
2. an *error* is the part of the system state which is responsible for the manifested failure (i.e., the delivery of a service not complying with the specification), and
3. a *failure* occurs when the delivered service differs from the expected service, where the expected service is described in the service specification. The failure occurred because the system was erroneous.

Upon occurrence, a fault creates a latent error, which becomes an effective error upon activation. When the effective error modifies the delivered service, a failure occurs. For example, a short circuit occurring in an integrated circuit is a fault, the consequence (modification of

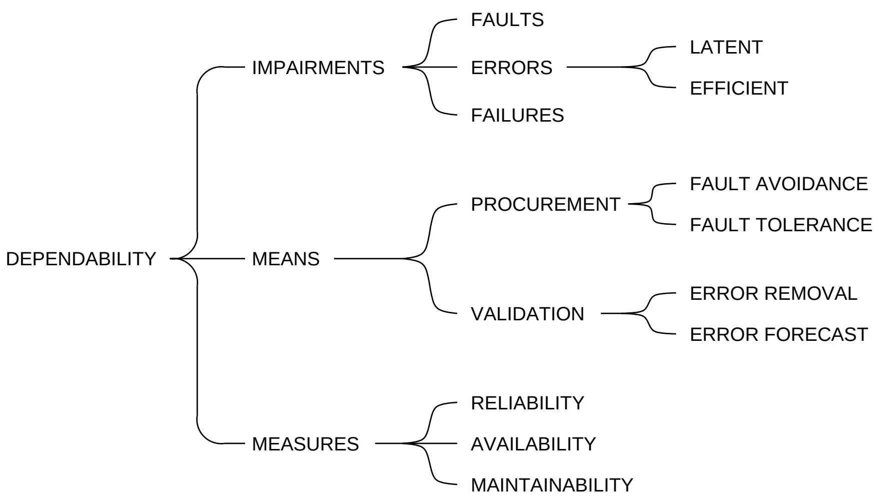


Figure 1.1: System dependability and related concepts

the circuit function) is a latent error. When the module where the error resides is activated (an appropriate input pattern selects the erroneous function) the error becomes effective. This effective error may produce incorrect data (value of results or timing of their delivery) which may affects the delivered service creating a failure. Definitions of fault, error and failure can be stated in other terms as: an error is the manifestation in the system of a fault, and a failure is the manifestation on the service of an error. The granularity of a particular dependability model always stops at the cause which is intended to be avoided or tolerated. Otherwise, a faults would become nothing else than a failure of a system having interacted with the considered system.

Faults can be categorized into the classes of *physical faults*: adverse physical phenomena, either internal (threshold changes, short circuits) or external (environmental effects), and *human-made* faults: imperfections, which may be either design faults (committed during the system design and modifications) or interaction faults (violations of operating or maintenance procedures).

The life of a system is perceived by its users as an alternation between *service accomplishment* (the service is delivered as specified) and *service interruption* (the delivered and specified services differ). The events causing transition between the above two system states are the *failure* and *restoration* (also referred to as repair). Quantification of the characteristics of this accomplishment/interruption alternation leads to three possible measures of dependability. *Reliability* measures the continuous service accomplishment (or equivalently the time to failure) from a reference initial instant. *Availability* rates the service completion with respect to the alteration of accomplishment and interruption. *Maintainability* quantifies the continuous service interruption (or equivalently the time to restoration). These characteristics depend on non-deterministic, stochastic processes, so they can be represented mathematically as probabilities.

The error latency duration may vary considerably, depending upon the fault, the system utilization and other circumstances. Whether or not an error will lead to a failure is determined by the activation conditions that cause a latent error to become effective, the system construction with the regard of available redundancy, and the definition of a failure from the users point of view (for example, voltage threshold levels in logical circuits imply the notion of "error acceptance regions"). Available redundancy can be classified as

1. *explicit redundancy*, built-in the system in order to ensure fault-tolerance, i.e., it is directly employed to prevent an error from leading to a failure, and
2. *implicit redundancy*, the level of redundancy residing in every system, which may play the same role as explicit redundancy. It is in fact difficult to build a system without any form of redundancy.

1.1.2 Means of dependability

Constructing a dependable computing system requires the combined utilization of a set of methods, which can be either *fault-avoiding*, preventing the occurrence of faults by appropriate construction of the system, or *fault-tolerant*, utilizing the implicit redundancy in a computing system to deliver the specified service in spite of that faults have occurred, *error-removing*, trying to minimize the presence of latent errors so they cannot be activated, or *error-forecasting*, estimating the presence, creation and consequences of errors by evaluation. The former two groups may be regarded as constituting *dependability procurement*, attempting to provide the system with the ability to deliver service complying with its specification, while the latter two groups may be considered as constituting *dependability validation*, aiming at reaching confidence in the system ability to deliver the specified service.

Fault-tolerance is carried out by *error processing*, which may be automatic or operator assisted. The purpose of error processing is the preservation of data integrity. Two constituent phases can be recognized: (1) *effective error processing*, bringing the effective error back to a latent state (if possible, back to the state right before the occurrence of the failure), (2) *latent error processing*, ensuring that a latent error does not become effective (again). This is carried out by making the error passive and reconfiguring the system to make it capable of delivering the specified service. Effective error processing can be realized either as *error compensation*, where the fault affected system component contains enough redundancy to fulfill the delivery of an error-free service despite the erroneous internal state, or *error recovery*, where an error-free state is substituted for the erroneous state.

Prior to error processing the erroneous state must be recognized (as early as possible), which is the task of *error detection*. On the other hand, error compensation may be applied systematically even in the absence of effective errors. This systematic application is called *error masking*, and it ensures that any effective error (that was taken into account during the design process) is brought back to a latent state. As an example, a logical AND gate can be protected of a single stuck-at-1 fault by a wired OR connection of duplicated AND gates. This way either of the two AND gates can independently pull down the output and realize the logical function. However, error masking can at the same time result an unregarded decrease of redundancy. This is well demonstrated by our example, as there is no information if none, one, or both of the duplicated AND gates became faulty during the operation of the system. Therefore, practical implementations of error masking generally involve error detection. This way error compensation can be applied again when an effective error has been detected, in order to check if latent error processing has to be executed.

The substitution of an error-free state for the erroneous state may be carried out in two directions: backward or forward. *Backward error recovery* transforms the erroneous system state into a previously occurred valid system state prior to the error becoming effective. *Forward error recovery* finds a new valid system state instead of the erroneous system state, which has never occurred or which at least has not occurred since the same erroneous state. In forward error recovery the damage caused by a detected error must be assessed. Damage assessment can be omitted in the case of backward recovery, provided that the mechanisms enabling the replacement of the erroneous state to a valid state have not been affected by the present faults.

Error processing is generally achieved by maintenance, which is aimed at removing latent errors. Maintenance actions can be *corrective*, removing the latent errors that became effective and have been processed, and *preventive*, eliminating latent errors before they become effective in the considered system. These latent errors can be the results of either physical faults occurred since the last maintenance action, or design faults led to effective errors in other similar systems. The classes of faults which can be tolerated depend on the fault hypothesis considered in the design process, and thus rely on the independence of the implicit or explicit redundancy from the fault creation and activation processes. The mechanisms being liable for fault-tolerance must be protected against the faults affecting them (e.g., voter replication, self-checking observers, stable memory for recovery programs and data [19]). Some system properties, like structural regularity, may limit the necessary amount of redundancy [20]. These properties directly affect the fault model of the system, and so indirectly the set of tolerated faults will also be influenced by them.

The necessary operational time overhead for effective error processing significantly differs in the two main prevention concepts. In error masking, the time overhead is constant, and in practice the duration of error compensation is much shorter than error recovery, due to the larger amount of redundancy. In error detection and recovery, the time overhead is longer when an error becomes effective than when it was latent (it is related to the preparation of recovery points). These characteristics of error processing constrain the applicable fault tolerance methods with respect to the user time granularity. On the other hand, they introduce a relation between operational time overhead and structural redundancy, in its most simple form: "the more the redundancy in the system structure, the less the operational time overhead incurred."

1.1.3 Model and elements of system-level fault diagnosis

System-level fault diagnosis uses a simplified system model composed of units, connections, faults, and tests. The system is built of *units* called *processing elements* (PEs) connected by interconnection links. The units execute independent tasks and cooperate with other units using messages. A unit can either be fault-free or faulty. *Faults* in the system are uncovered using system-level *tests* performed by the units on adjacent units or neighbors, directly accessible via an interconnection link. The collection of system-level test results is

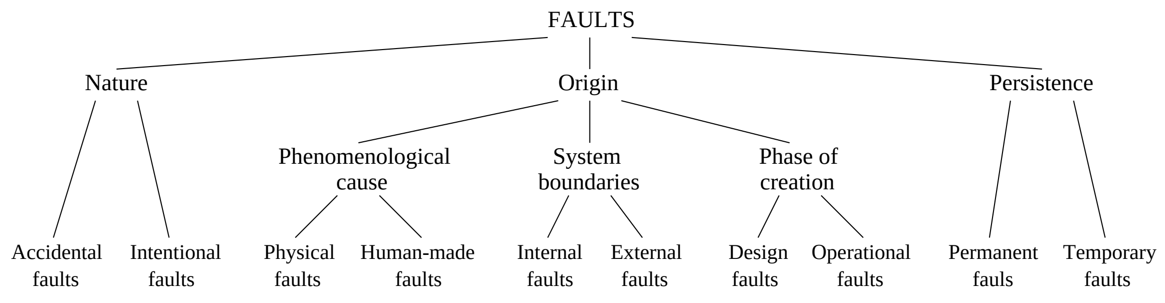


Figure 1.2: The classification of faults

the *syndrome*. The diagnostic process is the analysis of the syndrome in order to identify the faulty units.

Faults. Faults are undesirable—but preferably not unexpected—circumstances negatively affecting the service a system delivers. The sources of faults may be classified according to three main viewpoints: their *nature*, their *origin*, and their *persistence*. The main fault classes according to the three viewpoints [21] are presented in Figure 1.2.

System-level diagnosis distinguishes the following main fault categories:

Permanent faults. Permanent faults remain to exist after their occurrence until the related faulty component is repaired, or replaced. Their presence is not affected by point-wise conditions, thus they always produce errors when they are fully exercised. Internal physical or design faults belong to this category.

Intermittent faults. Intermittent faults are temporary internal faults. These faults are essentially permanent faults, but their activation depends on rare combinations of conditions. By their nature, intermittent faults do not occur consistently, which makes their detection a probabilistic event over time. Recurrent faults, whose activating conditions can be reproduced are also considered as intermittent faults in system-level diagnosis.

Transient faults. Transient faults are temporary external faults. They originate from the effects of the physical environment. Transient faults only affect the operation temporarily, usually without causing a permanent fault in the corresponding component. This fault class is hard to deal with in system-level diagnosis. Transient faults cannot be distinguished from intermittent faults just by simply observing the system. However, the difference is very important: the affected component cannot be declared faulty if a transient fault occurred (although a restart or a temporary shutdown might be necessary to restore the integrity of the system).

Hybrid-fault model. Transient and intermittent faults are the major source of errors in systems. Recent studies have shown, that generally more than 80 percent of the field failures were caused by these fault types [22]. Therefore, Mallela and Masson [23] introduced the concept of a *hybrid-fault* situation. The hybrid-fault model specifies bounded combinations of permanently faulty and intermittently faulty components in

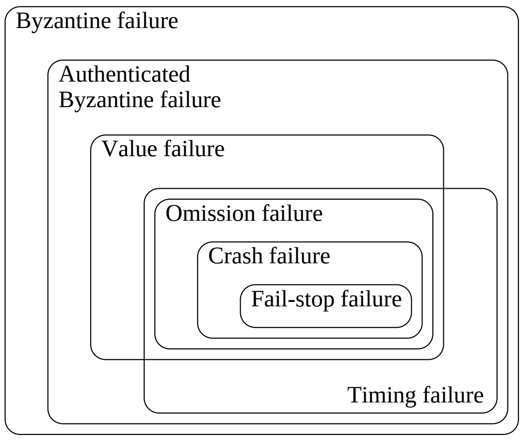


Figure 1.3: The relationship of failure classes

the system. The concept includes as special cases the case where each faulty unit may only be either permanently or intermittently faulty.

A system generally does not always fail in the same way. The ways a system can fail are specified as its *failure modes* [24]. They also may be categorized according to different viewpoints. The two main viewpoints significant in system-level diagnosis are the *failure domain* viewpoint, and the *failure perception* viewpoint [21]. From the failure domain viewpoint one can distinguish: *value failures*, when the value of the delivered service does not comply with the specification, and *timing failures*, if the service is not delivered within the specified response time frame. The latter class is sometimes called a *performance failure*. Early and late timing failures can be defined depending on whether the service was delivered before or after the specification. A class of failures relating to both value and timing failures are the *stopping failures*: the system activity is not any more perceptible and a constant value service is delivered. The special case of stopping failures when no service is delivered is called *omission failures*. If the omission of service delivery is persistent, a *crash failure* has occurred. The relationship of these failure classes is illustrated in Figure 1.3.

When a failed service is delivered to multiple users, the failure perception leads one to distinguish *consistent failures*, if all users have the same perception of the failure, or *inconsistent failures*, when the system users may have different perception of the same failure. Inconsistent failures are usually termed as *Byzantine failures*, because in this failure class a faulty processor may send different messages to its neighbors during the broadcast of a given piece of information [25]. Since any fault is possible within the Byzantine model, this failure class can be considered the universal fault set. *Authenticated Byzantine failures* constitute a subclass of Byzantine failures, where messages sent by processors are authenticated and even a failed sender unit cannot imperceptibly alter an authenticated message.

Tests. Testing is a *dynamic verification* of the system. A test, in a general sense, can be imagined as the application of some sequence of input patterns and the observation of one or more outputs for one or more time frames [26]. Tests are aimed at uncovering active faults by detecting the errors they caused. The generation of the test inputs can be either deterministic or probabilistic:

Table 1.1: Non-equivalent testing models according to [27]

Notation	Name	Reference	Test outcome			
			$\odot \to \odot$	$\odot \to \otimes$	$\otimes \to \odot$	$\otimes \to \otimes$
T_{00}	0-fail-safe tester		0	1	0	0
T_{01}	perfect tester		0	1	0	1
T_{0X}	sequential tester		0	1	0	X
T_{11}	irreflexive invalidation (1-fail-safe tester)	HK2 [28]	0	1	1	1
T_{1X}	reflexive invalidation	HK1 [28]	0	1	1	X
T_{X1}	asymmetric invalidation	BGM [29]	0	1	X	1
T_{XX}	symmetric invalidation	PMC [1]	0	1	X	X
T_{PT}	partial tester		0	X	X	X
$T_{\emptyset}$	0-information tester	Blount [30]	X	X	X	X

$\odot$ fault-free unit, $\otimes$ faulty unit, $\to$ test assignment

Deterministic testing. Test patterns are predetermined according to the adopted criteria and the given fault model. Test generation can be *functional* or *structural*, depending on whether the selection process is related to the function or the structure of the system.

Random, or statistical testing. Test patterns are selected on the basis of a defined probability distribution on the input domain. The distribution and the number of input patterns are determined according to the adopted criteria and the given fault model.

A test is said to be *complete* for a certain fault if it always fails when the fault is present, and passes when all faults are absent. On the system level, a test performed by unit u_i on unit u_j is complete if it detects all low-level faults that can cause a failure in unit u_j. Yet, for highly complex components such as processors exhaustive testing can take a lot of time and processing capacity, and still not produce accurate results. If the t_i test detects only the subset $F^d \subset \mathcal{F}, |F^d| = \phi^d$ of faults from the set of all possible faults $\mathcal{F}, |\mathcal{F}| = n$, it is called *incomplete*. The performance of t_i can be characterized by the *test coverage*, defined as $\text{cov}(t_i) = \phi^d/n$. Naturally, complete tests have 100 percent test coverage. A second problem that reduces the effectiveness of tests is *test invalidation*. Ideally, the outcome of a test on a fault is not influenced by the presence or absence of other faults. A complete test is called *invalid* if some faults can make it erroneously pass when the fault (for which the test is complete) exists, or can make it erroneously fail when the same fault does not exist.

The phenomena of testing impairment (incompleteness, invalidation) is determined by the behavior of the (faulty) tester unit (i.e., its *failure semantic* [24]) and the relationship of the tester and tested units. Table 1.1 presents a complete tabulation of the non-equivalent testing models. The table lists the test result in the four possible configurations, respective to the fault state of the tester and tested units. The test result is denoted by 0 if it has passed, 1 if it has failed, and $X \in \{0, 1\}$ if the test result may be arbitrarily pass or fail in the same fault situation.

Test invalidation models are also strongly dependant on the applied testing mechanisms.

The most commonly used testing mechanisms are:

Direct testing. In *direct testing*, the units of the system execute tests on each other. Typically, unit u_i tests unit u_j by giving it certain input sequences and comparing the resulting output sequences with the set of correct responses. Usually only the normal communication facilities can be used for testing purposes. Since units are able to execute the tests individually, these tests can be invalidated only by a fault in the tester unit. This is also known as *single invalidation per test* (SIPT).

Self-testing. Testing can also be performed by each processing element on itself in a series of *self-tests*. Kuhl and Reddy proposed a hierarchical system of built-in fault tolerance techniques and monitoring devices [31]. These include hardware or software checkers, watchdog timers and processors, error-checking codes, or redundancy with voting. Thus, the testing of a unit becomes a simple request to the built-in self-test component for the actual fault status of the tested unit.

Comparison testing. A practical method of detecting faults in components is performed by *comparison*. Two or more units are assigned the same task and the output of the task execution of each unit are compared to each other. A difference in comparing the results of two units signifies an error in one or both of them, but there is no way to tell which one of the units is/are faulty. Using comparison testing and *n-multiple redundancy* (NMR) techniques up to $\lceil n/2 \rceil - 1$ errors can be tolerated.

While direct testing is mostly performed periodically in dedicated testing sessions between normal system activities, comparison tests can take place concurrently to productive tasks. An error is detected immediately upon propagation. Comparison tests can be made more compact by creating and comparing *signatures* of the results, such as checksums or *cyclic redundancy checking* (CRC) codes.

Multiple invalidation per test. A test may consist of a large number of stimuli to obtain a good test coverage. Testing can be done by a single unit, but as another possibility, it can be the result of cooperation among multiple testing units. In the latter case a fault in any of the participating units can invalidate the result. If the failure of more than one independent modules can invalidate a test, the test invalidation model is follows a *multiple invalidation per test* (MIPT) scheme.

Time domain testing. A large portion of the potential failures manifests in the time domain (see Figure 1.3). Therefore, it is necessary to also test the behavior of system components with respect to *time*. When a unit fails to complete a task by a certain deadline, or a message is not received within a certain time frame, a timing failure occurs. Errors causing timing failures can be detected using time stamps and watchdog timers. The consistency of time domain testing usually requires the presence of a global set of synchronized clocks. A common time domain testing technique is the use of $<$ I'm alive $>$ or "heartbeat" messages. In this technique units periodically send $<$ I'm alive $>$ messages to their neighbors, and validate the incoming messages by time-outs. When an $<$ I'm alive $>$ message is late or is omitted, the origin of the message is declared as faulty.

Note, that the time domain testing model is orthogonal to the value domain testing methods described earlier. Timing errors may remain undetected in the value domain as well as value failures may stay uncovered in the time domain.

Direct testing is the oldest and most thoroughly studied testing method. Many invalidation schemes have been devised over the years, most of them have already been mentioned in Table 1.1. The leading ideas behind the main invalidation schemes of direct testing are the following:

Symmetric test invalidation. This model was introduced in the late sixties by Preparata et al. [1]. Since then it became the most widely used test invalidation model in system-level diagnosis. The main feature of *symmetric invalidation* is that a faulty tester unit generates an arbitrary test result, i.e., the test may pass or fail independent on the state of the tested unit. The main advantage of the model lies in its generality: all other complete test SIPT invalidation models produce syndromes compatible with it. Thus, the algorithms designed for symmetric test invalidation are applicable to a wide range of systems, even if they will not necessarily provide optimal diagnostic performance.

Asymmetric test invalidation. Barsi et al. proposed a less restrictive invalidation model in [29]. In their concept the system components are complex devices having many different failure modes, so testing units must apply a large number of test stimuli for a complete test. They assumed that two distinct units fail identically with a negligible probability. Therefore, at least one mismatch occurs between the test outputs and expected results when the tested unit is faulty. However, a faulty tester unit testing a fault-free tested unit can still produce an arbitrary test result. The above model is called *asymmetric invalidation*. The limitation of the possible syndromes make the decoding of syndrome easier: all tests with outcome $a_{ij} = 0$ unambiguously identify a fault-free tested unit. The $a_{ij} = 1$ test results have the same interpretation as for symmetric invalidation.

HK models. Later, Kreutzer and Hakimi [28] used the same assumptions to extend the BGM model. They suggested, that a test on a fault-free tested unit by faulty tester unit should always fail. Two mechanisms comply with this situation: in *reflexive invalidation* arbitrary test outcomes are possible when both the tester and tested units are faulty, while in *irreflexive invalidation* a faulty unit tester always generates a failed test result. They also convey more diagnostic information than the original Preparata model: for reflexive invalidation, the $a_{ij} = 0$ test outcome means that both units associated with the test have the same fault state; for irreflexive invalidation it even implies that they are unambiguously fault-free.

General test invalidation models. Somani et al. presented a generalized characterization theorem for system-level diagnosis [32]. They considered permanent and intermittent faults, and symmetric and asymmetric test invalidation in their model. For

Table 1.2: Comparison testing model

Comparator	**Compared unit u_i**	**Compared unit u_j**	**Comparison result**
fault-free	fault-free	fault-free	0
fault-free	fault-free	faulty	1
fault-free	faulty	fault-free	1
fault-free	faulty	faulty	1
faulty	both states	both states	X

the model they derived the necessary and sufficient conditions for any fault pattern of any size to be uniquely diagnosable.

Selényi introduced a unified framework for complete test SIPT invalidation schemes [33]. His work includes the diagnosability analysis of all the nine possible invalidation mechanisms, as well as the optimal testing arrangements for them. He also developed a diagnosis algorithm for the unified model. The algorithm can handle heterogeneous systems consisting of various components following different test invalidation models.

Malek [23] introduced the comparison approach for system-level diagnosis and gave a method to assign comparison edges in the testing graph. A syndrome of comparison results is created by labeling an edge with a "0" if it connects two units that agree, and with a "1" if it connect two disagreeing units. Maeng and Malek [34] suggested that a third unit should be employed to compare the results, thus decentralizing the arbitration of comparisons. Their testing model is shown on Table 1.2.

The theory of multiple invalidation per test was introduced by Russel and Kime [35, 36, 37]. Their model is formulated of faults, tests, and their relationships. The model assumes, that at least one complete test exists for each possible fault in the system. The tests are complete for one and only one fault, there is a *single unit per test* (SUPT). Tests can be invalidated by any number of faults, that is *hard-core* tests (not invalidated at all) are also included in the model. The authors defined two different types of systems (the notation $T(F^i)$ represents the set of tests invalidated by the presence of fault pattern F^i):

1. *semimorphic* systems: $T(F^i \cup F^j) \subseteq T(F^i) \cup T(F^j)$, and
2. *morphic* systems: $T(F^i \cup F^j) = T(F^i) \cup T(F^j)$

Kime [35, 26] and later Maheswari and Hakimi [38] extended the model by allowing tests that are complete for more than one fault. More precisely, a valid test t_i will fail if one or more faults in $F(t_i)$ are present, and will pass if all faults are absent ($F(t_i)$ denotes the set of faults t_i is complete for). This testing scheme is called *multiple units per test* (MUPT). As Gupta and Ramakrishnan [39] pointed out, multiple invalidation can sometimes may result in loss of diagnostic information. They considered systems, where several units conduct multiple independent tests on a unit. Each of the tests probes a different part of the unit under test, and the results are combined into one complete test. Under these conditions, a faulty testing unit does not necessarily make the combined results unreliable.

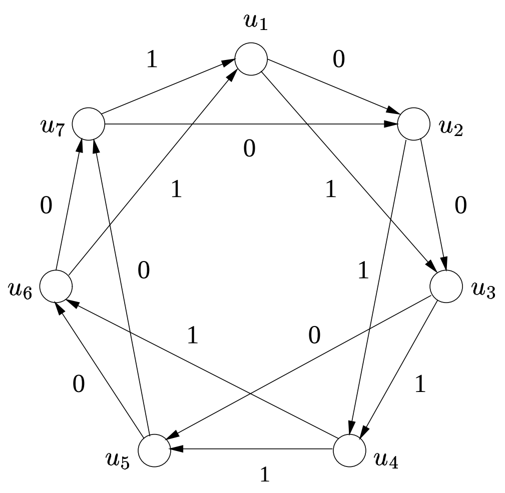


Figure 1.4: An example diagnostic situation

1.2 System-level fault diagnosis

To get acquainted with the problem of system-level fault diagnosis and the possible solution methods, let us consider a small example taken from [33] of a fault-tolerant multiprocessor system consisting of 7 units. Each unit is capable of testing two other units, as illustrated in Figure 1.4. The directed arcs represent tester-tested unit relationships, the arrow points from the tester unit to the unit under test. In other words, Figure 1.4 is a 7-node, directed $D_{1,2}$ testing graph. Every unit conforms to the symmetric test invalidation model.

In centralized diagnosis first each test is carried out and the results are collected by the central supervising module. The complete set of test results is then analyzed. In our example, the outcome of the tests is written on the arrows (passed tests are denoted by a 0, failed test by a 1). From the syndrome we can draw the following conclusions on a purely logical bases without any particular knowledge of system-level diagnosis:

- The system contains at least one faulty unit, otherwise all of the test would pass following from the complete-test assumption. This conclusion provides a *fault-detecting* information.

- Unit u_1 is surely faulty. Let us suppose that u_i is fault-free. In this case, units u_2 and u_3 must also be fault-free, since they were tested by fault-free units and the tests passed. However, at the same time u_3 is faulty according to u_1. Since there is a contradiction among the conclusions, we must invert our original hypothesis. This conclusion provides a *fault-localizing* information.

- The fault state of units u_2, u_3, u_5, u_6, and u_7 is the same. The reason is that these units are involved in a closed loop of 0-links (passed tests). Suppose one of these units is fault-free. Then the unit tested by the fault-free one must also be fault-free. On the other hand, if one of the units is faulty, then its tester must be faulty as well. The same rule applies for every situation where units are part of a closed loop (or a strongly-connected subgraph) composed only of 0-links.

- A possible diagnosis is that all of the units in the system are faulty. Since faulty units can generate any test outcome independent on the state of the tested unit, every syndrome is is compatible with this situation, including the one shown in Figure 1.4.

The last conclusion is not very useful in practice, however it cannot be left out of consideration without some *a priori* information. Assume we can guarantee (with an acceptable probability) that at most two units can be faulty at the same time—based on the thorough knowledge of the system. In this case units u_2, u_3, u_5, u_6, and u_7 are surely fault-free. This can be inferred, because these units have the same fault state and the existence of 6 faulty units would exceed our assumed limit. Consequently, u_4 is faulty, according to units u_2, u_3. We managed to classify all of the units in the system! Our classification is contradiction-free and compatible with the syndrome, thus it is another possible diagnosis.

Further experimenting with our example, we can state the following:

- No diagnosis exists if no more than one unit can be faulty.
- When at most 2 to 5 units can be faulty the above described diagnosis is the only solution.
- When at most 6 units can be faulty, there are two possible diagnoses: the one described above, and the "only unit u_4 is fault-free" solution.

The fundamental model for multiprocessor system-level fault diagnosis was developed by Preparata, Metze, and Chien [1]. This model formed a framework for much of the research in the area. We present the model as a basis for all main issues in further discussion.

Preparata et al. made several assumptions considering *diagnosable* systems. They assumed that a large computing system is composed of many small processing units. These units can execute test on each other to determine the condition of other units. A test outcome is evaluated as "pass" or "fail" on the basis of the test responses. The tester unit classifies the tested unit as fault-free or faulty. Units are *intelligent*: they are able to perform testing task individually, independent on other units. A fault-free unit is always able to identify accurately the fault state of the tested unit (this is the complete test assumption). A faulty unit produces unreliable test results, independent on the fault state of the tested unit (i.e., they behave according to the symmetric test invalidation model as mentioned in Table 1.1). All faults are hard or permanent faults. Tests are executed periodically by each unit on its neighbors according to a predefined schedule. The local test results are sent from the units to a central diagnostic device. The diagnostic algorithm running on the central device analyzes the syndrome and creates a global diagnostic image containing the fault state of each unit in the system.

Three important problems arise in connection with the Preparata, Metze, and Chien (PMC) model:

Characterization problem. Characterization aims at finding the necessary and sufficient conditions for the test connection assignment of a system to achieve a given level of diagnosability under a certain fault model and an allowable family of fault sets.

Diagnosability problem. Diagnosability means determining the necessary and sufficient conditions for a group of fault sets in a given test connection assignment and fault model to be uniquely identifiable.

Diagnosis problem. The goal of diagnosis is to determine a fault set present in a systems based on a given syndrome, test connection assignment, and fault model assuming that the set of faults present belongs to valid group of fault sets.

Note, that the characterization problem and the diagnosability problem are not the same. The former is a theoretical problem while the latter is an algorithmic problem. Thus, a good answer to the characterization problem may result an efficient algorithm for the diagnosability problem.

Originally, Preparata et al. were interested in the so-called t-diagnosable systems. These systems allow correct and complete diagnosis under the presented conditions if at most t units can become simultaneously faulty. A diagnosis is said to be *complete* if all faulty units can be identified from a syndrome resulted in the presence of this fault set. A diagnosis is *correct* when no fault-free units are improperly classified to be faulty.

Two different diagnostic strategies can be applied to t-diagnosable systems: one-step diagnosis and sequential diagnosis. In *one-step diagnosis* or *diagnosis without repair* any possible fault set can be diagnosed correctly and completely based on a single syndrome produced by the system. This is the highest quality of diagnosis, but at the same time it has the strongest conditions. In *sequential diagnosis* or *diagnosis with repair* the location of faulty units is determined in more phases. At first only a subset of the faults present is diagnosed. These faulty units are replaced or repaired. Then the set of tests in the system is executed again, and the same process is repeated until no more errors are detected. To succeed, the diagnosis algorithm has to identify at least one faulty unit decoding every single syndrome.

The necessary conditions of one-step t diagnosability for a system consisting of n units under the PMC model were given by the authors:

$$\begin{aligned} n &\geq 2t+1 && (1.1)\\ |\Gamma^{-1}(u_i)| &\geq t, \forall u_i \text{ such that } u_i \in U && (1.2) \end{aligned}$$

Hakimi and Amin showed that for the special case when mutual tests between units are not allowed, the necessary conditions are also sufficient [40]. To achieve a general characterization of one-step t-diagnosable systems they also added a third condition:

$$\begin{aligned} |\Gamma^{-1}(U_k)| &\geq p, \forall p \text{ such that } 0 < p \leq t, \text{ and}\\ &\quad \forall U_k \text{ such that } U_k \subset U \text{ and } |U_k| = n - 2t + p && (1.3) \end{aligned}$$

The most significant result for sequential t-fault diagnosability deals with single-loop systems (i.e., systems in which there is exactly one test on each unit and one test performed by each unit) [1]. A single loop system is sequentially t-diagnosable if and only if n, the size of the loop, satisfies

$$n \geq \left\lfloor \left(\frac{t+2}{2}\right)^2 \right\rfloor + 1 \qquad (1.4)$$

The diagnosability problem was solved by Sullivan [41]. He used *network flow* to construct an $O(|E|n^{2.5})$ algorithm that computes the t-diagnosability of a given test connection assignment under the PMC model. Recently, Raghavan and Tripathi improved the efficiency of the algorithm to $O(nt^{2.5})$ [42]. They also proved that calculating the sequential diagnosability of reparable systems is co-NP complete [43].

The problem of finding a minimal cardinality set of faulty units in whose presence a system can produce a given syndrome (i.e., the diagnosis problem) is NP complete according to Fujiwara and Kinoshita [44]. Still, much work has been done in restricted situations. For t-diagnosable systems under the PMC model Kameda, Toida, and Allan gave an $O(t|E|)$ algorithm (KTA) of one-step diagnosis. Later, Sullivan improved the KTA solution and extended it by employing weighted digraphs in his system model [45]. The algorithm has $O(t^3 + |E|)$ time complexity, which is the best known solution in the practically relevant case when t is small compared to n.

The best method in terms of worst case efficiency was given by Dahbura and Masson [46, 47]. Their algorithm first calculates the union of implied faulty sets for all units, represented by the graph G_{L} called the *L-graph.* Then, the set of faulty units is identified as the *unique minimum vertex cover set* of G_{L}. The authors proved, that for the class of graphs that include G_{L} the problem to find a unique minimum vertex cover set can be solved in polynomial time. The total time complexity of the diagnosis algorithm is $O(n^{2.5})$.

1.2.1 Classification of system-level diagnostic algorithms

Practice has quickly proven that the original system model and diagnostic goals formulated by Preparata et al. make a number of assumptions that do not directly conform to existing hardware. Much research have been therefore conducted toward more widely applicable diagnostic models. The addressed shortcomings of the PMC approach and the proposed solutions are the following:

Complete tests. Algorithms based on the PMC model assume, that the result of a test performed by a fault-free tester unit always reflects the real fault state of the tested unit. However, in most practical cases it is impossible to set up tests that fully probe all of the complex components of a unit. Thus, tests usually do not have 100 percent coverage. The *comparison testing* technique can improve on this situation, but it still does not assure the test to be complete. *Probabilistic methods* attempt to provide the most probable diagnosis to cope with imperfect tests. The most general attempt in this direction is Blount's *0-information tester* model, although that approach could never gain any practical significance despite of all its generality.

Permanent faults. The only fault class the PMC model considers is the class of permanent faults. It was also assumed, that fault-free and faulty units do not change their fault states (i.e., do not fail or get repaired) during the whole testing session. Fault injection based experiments [22] and practical experience [48] has shown that the majority of occurring faults are either transient or intermittent. Many different fault models were proposed to include *intermittent* and *hybrid fault* situations in system-level diagnosis. Most of the diagnosis methods based on these models can handle not only intermittent faults but also incomplete tests. This is not surprising since the fault to syndrome correspondence is very similar in both cases.

Incomplete knowledge of failure modes. The Preparata model has a very simplified and pessimistic view of the behavior of faulty units. This general approach may be applicable to many systems, but more efficient algorithms can be formulated if there are additional information available on faulty units. *Asymmetric invalidation* by Barsi et al. placed less restrictive requirements on faulty units [29], later Hakimi and Amin suggested different invalidation mechanisms [40]. Still, these models assume that the system consist of identical components, i.e., it is homogeneous. The *generalized test invalidation* model [33] integrates all of the possible SIPT/SUPT mechanisms and at the same time can handle heterogeneous systems. Recently, a constraint-based representation was introduced in the framework of generalized invalidation, further extending its modeling power [49, 50].

Intelligent tester units. The majority of testing mechanisms require the tester unit to carry out the testing task individually. A test may be so complex that it requires the cooperation of several units. Unfortunately, this also means that the test is invalidated by multiple faults. Only *MIPT diagnostic models* are able to represent these situations. Another disadvantage of assuming intelligent tester units is that it excludes passive components not participating in the diagnosis (like communication devices, mathematical coprocessors, etc.) from the scope of the diagnosis. Liaw et al. suggested a solution to the problem by introducing a heterogeneous model composed of *testing* and *non-testing* units [51]. The authors also gave a classification and a distributed diagnosis algorithm for their model.

The number of faults is bounded. Preparata et al. proved that $n \geq 2t + 1$ must hold for the number of units in one-step t-fault diagnosable systems. Since n in most cases does not grow during run-time, the above condition is essentially an upper bound on the number of faults. In practice, if the system was not built specifically to fulfill the above condition then t may exceed the static limit. There are two approaches that handle larger amounts of faulty units: *sequential diagnosis* and *probabilistic methods.* Sequential diagnosis concentrates on locating and repairing only one fault at a time [52]. If the repair-rate is larger than the fault-rate, this strategy will eventually lead to a completely fault-free system. Probabilistic diagnostic methods try to the fault state of each unit, but the correctness and/or completeness of the resulting diagnostic image is not guaranteed.

Centralized diagnosis. Often it has been assumed that a hard-core central supervising module exists in the system to analyze syndromes. Every unit transfers its local test results to this module via a direct and reliable communication link. This assumption violates two important design principles. The first principle states that a fault-tolerant system must be free of single-point failures;in other words it cannot have hard-core components. According to the second principle reliable massively parallel systems should be scalable and gracefully degrading. The diagnostic algorithm itself must be *distributed* to avoid these pitfalls.

Test organization. Tests are organized in space (testing graph) and in time (test schedule). The testing graph defines the tester—tested unit relationships, and so it mainly depends on the interconnection topology of the system. Many diagnostic algorithms is restricted to a specific topology. *General topology* algorithms have no such restrictions, thus they can be applied to all kinds of 'topologically static' systems. Sometimes the interconnection topology is also subject to change (e.g., due to failures in communication devices). In this case *adaptive* algorithms—which automatically accommodate to the dynamically changing circumstances and use an optimal amount of tests—can be used.

Many diagnostic algorithms ignore the timing requirements of tests: the test schedule. They treat the syndrome as a static 'snapshot' of the system. Thus, all of the units are supposed to be tested simultaneously, and there cannot be a new fault occurrence during testing and diagnosis. In reality, the execution of tests and the distribution of test results have a significant execution time. Units must perform the testing in a coordinated way to avoid deadlocks and performance bottlenecks. They must also be able to reconstruct the temporal order of the tests, no matter how the communication delays and the lack of synchronization among units may have distorted it. The time spent with communication should be kept minimal. These issues are addressed by *event-driven* algorithms.

Reliable communication. Communication is a vital issue in system-level diagnosis. Since testing is performed locally at processing elements, the test results must be transferred to the central supervising module (centralized diagnosis) or broadcast to a set of other units (distributed diagnosis). Usually the algorithms leave the failures of the interconnection links and devices out of consideration. In this case the underlying hardware must guarantee the reliable message transport using techniques as broken line detection, error detection and correction (EDAC) codes, message authentication, etc. Adaptive algorithms can operate in weaker conditions: it is enough if the communication has fail-stop or crash failure semantics. Finally, the scope of diagnosis can be broadened to include the location of faults in the interconnection links and devices as well. It is most advantageous in fine-grain large distributed systems, where proper diagnostic support can reduce the time and cost of maintenance.

Diagnostic goals. A natural expectation of a diagnostic algorithm is to find every faulty unit at once. However, this *one-step* diagnosis is usually quite a time and resource consuming task. Other methods achieve less precise results, but have much lesser requirements. Sequential diagnosis finds only one fault at a time. *Set diagnosis* [53] aims at finding a set of units, containing all of the faulty ones and potentially some fault-free ones as well. This can be acceptable when either single-unit diagnosability is not practical, or the system granularity is so coarse, that replacing a set of units can be done by replacing a single *field-replaceable unit* (FRU).

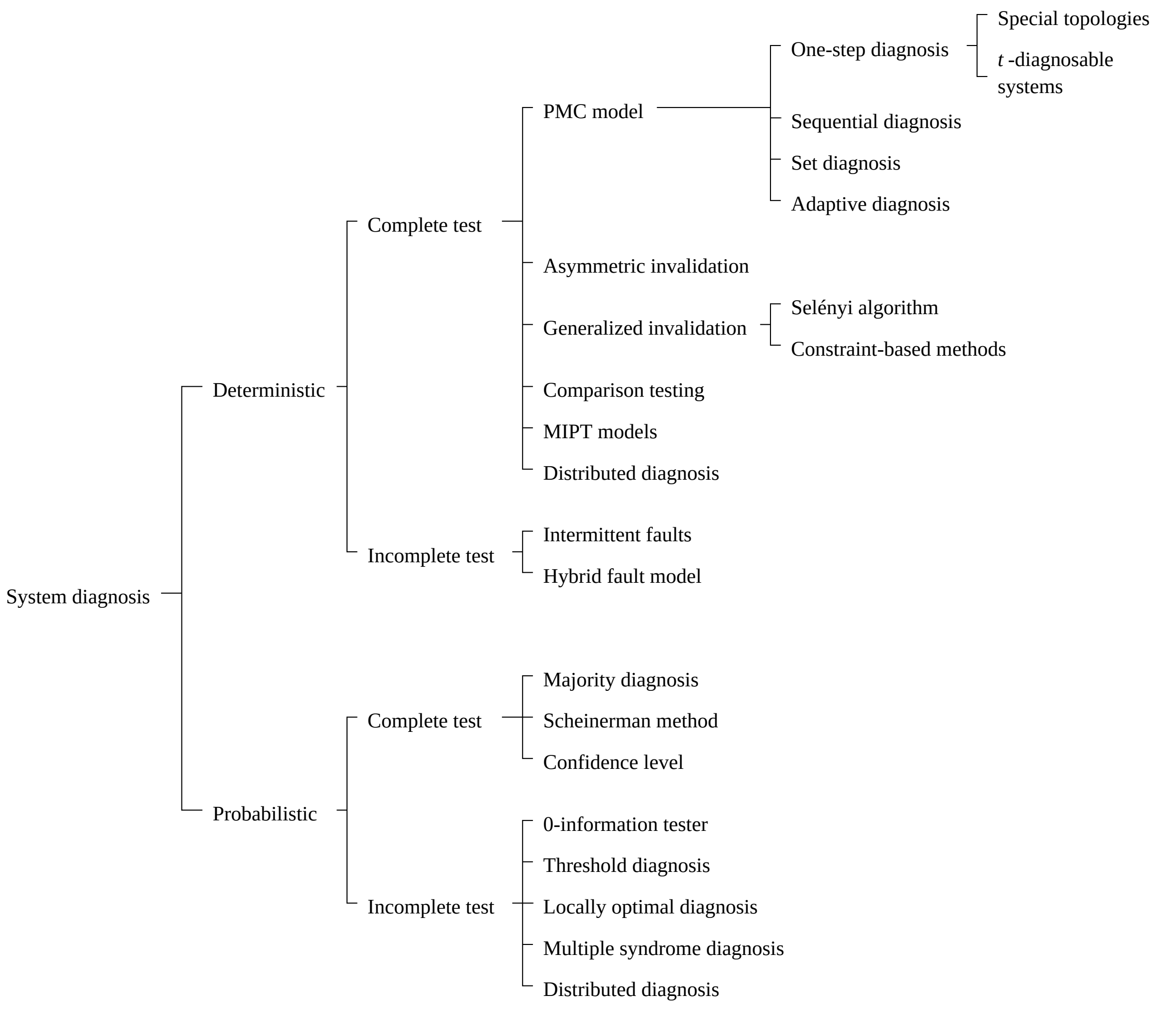


Figure 1.5: Classification of main research areas of system-level diagnosis

System-level diagnosis aims at solving fault classification problems equivalent or similar to the example in the beginning of this section *in the form of a general algorithm*. As Figure 1.5 points out many approaches to generalize the diagnostic task were developed over the years. They are too numerous to be described even briefly, therefore we chose a different approach. Instead of a global and outlined overview, only those works are mentioned which either illustrate a basic concept or have some direct correspondence to the diagnostic algorithms subject to this thesis. For further discussion of the field the reader is referred to a survey of system-level diagnostic algorithms [54] by the author, or a broader view of the consensus and diagnostic problems [55] by Barborak et al..

1.2.2 Centralized diagnostic algorithms

This section briefly outlines a selected set of centralized diagnostic algorithms. In the first part deterministic algorithms are desribed, then the existing probability models and the most notable probabilistic diagnostic approaches are introduced.

Deterministic centralized diagnosis

One of the first methods presented for one-step t-fault diagnosable systems was the KTA algorithm by Kameda, et al. [56]. They treated the syndrome decoding as a state-space search. Their state-space consists of the different unique fault state to unit assignments. The search terminates when a complete unit classification is achieved. The control algorithm assures that the generated diagnosis is unique, free of contradictions, and consistent with the syndrome.

At the beginning an arbitrary unit is selected and assigned to be fault-free $U^0 \leftarrow U^0 \cup u_i$ (explicit assignment). Then two sets, the 'implied fault-free' $L^0(u_i)$ and 'implied faulty' $L^1(u_i)$ sets are calculated. The sets are initialized to $L^0(u_i) \leftarrow u_i$ and $L^1(u_i) \leftarrow \emptyset$. New elements are appended to the sets using the following implication rules:

Unit	**Test outcome**	**Implication**
$u_j \in L^0(u_i)$	$a_{jk} = 0$	$L^0(u_i) \leftarrow L^0(u_i) \cup u_k$
$u_j \in L^0(u_i)$	$a_{jk} = 1$	$L^1(u_i) \leftarrow L^1(u_i) \cup u_k$
$u_j \in L^0(u_i)$	$a_{kj} = 1$	$L^1(u_i) \leftarrow L^1(u_i) \cup u_k$
$u_j \in L^1(u_i)$	$a_{kj} = 0$	$L^1(u_i) \leftarrow L^1(u_i) \cup u_k$

The algorithm encounters a contradiction when the $L^0(u_i) \cap L^1(u_i) \neq \emptyset$ condition is met during the generation of the sets. In this case the initial explicit assignment was not consistent with the syndrome, so it is changed to faulty: $U^0 \leftarrow U^0 - u_i, U^1 \leftarrow U^1 \cup u_i$ (forced assignment), $L^0(u_i) \leftarrow \emptyset$ and the $L^1(u_i)$ set is recomputed. Another condition to evaluate is the number of the faulty units found so far: $|U^1| = |L^1(u_i) \cup L^1_j \cup \cdots \cup L^1_k|$. If this number is below the diagnostic t limit, the units in $L^0(u_i)$ and $L^1(u_i)$ are tentatively labeled as fault-free and faulty, respectively. If it exceeds the limit, again the last explicit assignment is changed to faulty, $L^0(u_i) \leftarrow \emptyset$ and $L^1(u_i)$ is recomputed.

Forced assignments are sure classifications and cannot be changed. Normal explicit assignments, however, are just fault hypotheses and eventually they may need to be changed. When such an assignment is changed its consequences must be cancelled (backtracking step). Therefore the algorithm maintains a stack structure to store the explicit assignments and the corresponding $L^0(u_i)$ and $L^1(u_i)$ implied sets. Due to the above 'branch and bound' technique, the algorithm traverses the state space in a depth-first manner. The search terminates when either all of the units were assigned a fault state (diagnosis is complete), or a backtracking would be necessary and the stack is empty (diagnosis is impossible).

Meyer and Masson published a much simpler algorithm for the same problem in [57]. They store the explicit assignments and their consequences in a matrix representation to avoid backtracking. Forced assignments are not used, the conditions of one-step t-diagnosability guarantee the solution to be unique anyway. Their algorithm also requires a specific topology to work: the testing graph is a $D_{1,t}$ interconnection design. However, the algorithm can be easily adopted to other testing structures.

The consequence matrix $\mathbf{B}$ is a binary matrix: $\forall i, j : b_{ij} \in \{0, 1\}$. It is initially empty. The B_i row vector contains the implications drawn from the explicit assignment $U^0 \leftarrow U^0 \cup u_i$. First the b_{ii} elements in the main diagonal of $\mathbf{B}$ are set to 0. Then the row vectors are filled one-by-one. Let us imagine that the $\mathbf{B}_1, \ldots, \mathbf{B}_{i-1}$ vectors are already complete.

The b_{ii} element of B_i is 0, reflecting that unit u_i is supposed to be fault-free. Take the a_{ij} test results local to unit u_i, where $u_j \in \Gamma(u_i), j = i+1, i+2, \ldots, i+t$. Let $b_{ij} \Leftarrow a_{ij}$ including the last passed test, say a_{ik}, until $a_{i,k+1} = 1$. Now take the a_{kl} test results local to u_k, that is $u_l \in \Gamma(u_k), l = k+1, k+2, \ldots, k+t$ and repeat the previous procedure. The fill process is terminated when either the row vector is completely full, or the number of ones in the row (the fuond failed test results) reaches t. In the latter case the remaining row vector elements are set to 0.

The completed **B** consequence matrix has the following two properties: (1) the row vectors corresponding to fault-free units are identical, i.e., the b_{ij} element of these vectors reflect the correct fault state of unit u_j, and (2) the elements of row vectors corresponding to faulty units contain arbitrary values. Now it is easy to understand the final, classification step of the algorithm. Since it is known that the fault-free units are in majority in the system (due to the conditions of one-step t-fault diagnosability, since there are $n \geq 2t+1$ units and only at most t of them might be faulty), we can conclude that at least $t+1$ B_i row vectors are filled out correctly. Then, a unit u_k can be classified as faulty iff in the corresponding column the b_{jk} elements ($j = 1, 2, \ldots, n$) contain at least $t+1$ ones.

Dahbura and Masson approached the diagnostic problem from a purely graph-theoretical viewpoint [46, 47]. Their solution is composed of three main steps. In the first step they generate the so called 'implied faulty' graph. Then a maximum matching of this graph is computed. Finally, a labeling method is applied to the graph, where the labels represent the classification of the corresponding units.

The 'implied faulty' G^{01} graph is an undirected graph, which contains an edge from each node u_i to the nodes corresponding to units in the 'implied faulty' $L^1(u_i)$ set of unit u_i. The 'implied faulty' set can be created from the syndrome using the following equation:

$$L^1(u_i) = \Delta_1(D_0(u_i)) \cup A_0(\Delta_1(D_0(u_i))).$$

The authors propose a more effective method to obtain the information required to construct the G^{01} graph. Let $\mathbf{X}$ be an $n \times n$ matrix such that $x_{ij} = 1$ if and only if $a_{ij} = 0$ or $i = j$, and 0 otherwise. It follows that $\hat{\mathbf{X}}$, the transitive closure of $\mathbf{X}$ is an $n \times n$ matrix in which $x_{ij} = 1$ if and only if $u_j \in D_0(u_i)$. Let $\mathbf{Y}$ be an $n \times n$ matrix such that $y_{ij} = 1$ if and only if $a_{ij} = 1$ or $a_{ji} = 1$, and 0 otherwise. Then the matrix $\mathbf{Z} = \hat{\mathbf{X}}\mathbf{Y}\hat{\mathbf{X}}^T$ is an $n \times n$ matrix where $z_{ij} > 0$ if and only if $u_j \in L^1(u_i)$. The so obtained $\mathbf{Z}$ matrix is the adjacency matrix of the graph G^{01}.

The 'implied faulty' graph is a bipartite graph due to the rule of symmetry: for $u_i, u_j \in U, u_j \in L^1(u_i)$ if and only if $u_i \in L^1_j$. The authors proved that this graph contains a maximum matching which saturates each node corresponding to faulty units in a one-step t-diagnosable system. The minimum vertex set of this maximum matching is the set of faulty units. The algorithm employs a simple labeling method to determine the minimum vertex set. Those units not saturated by the maximum matching can be immediately labeled as fault-free, and placed into a queue. One of these units, say unit u_i is then removed from the queue. If u_i is labeled as fault-free, then the units adjacent to it are labeled as faulty and placed into the queue. If u_i is faulty, then the unit adjacent to u_i in the maximum matching is labeled as fault-free and placed into the queue. This procedure is repeated until the queue is empty, that is the labeling is complete. The generated node labeling constitutes the diagnosis.

In traditional centralized fault diagnostic algorithms it is explicitly assumed that one first chooses a set of test connections T, then obtains the results of these tests, and finally analyzes the test results to identify the faulty units (assuming that the number of faulty units does not exceed the t-limit). Nakajima [58] was the first to suggest a departure from this practice. He proposed an *adaptive* scheme to choose the tests to be executed and to examine their results until at least one fault-free unit can be identified. Then, one can use this unit as a tester and to locate all of the faulty units. Nakajima showed that one can identify a fault-free unit by using the results of some $t(t+1)$ tests. This implied that $(n-1)+t(t+1)$ tests are sufficient to find every faulty unit.

Later, Hakimi and Nakajima improved on this result [59]. Their method requires only $2t-1$ tests to find a fault-free unit. Therefore, every faulty unit can be located after the application of at most $n+2t-2$ tests. The authors also gave an optimal adaptive diagnostic algorithm for the asymmetric test invalidation model, and examined the possible application of their methods to fault diagnosis in computer networks. The algorithm for identifying one fault-free unit is based on two observations. First, if u_i and u_j are two adjacent units in the system and $a_{ji}=0$, then it is not possible that u_j is fault-free while u_i is faulty. Second, if $a_{ji}=0$, then it is not possible that both of u_i and u_j are fault-free.

Two sets of units are generated by the algorithm: U^0 denotes the set of possibly fault-free units, and U^1 denotes the set of accused faulty units. At the beginning both sets are empty. First an arbitrary unit $u_i \in U$ is selected and placed in U^0. Then, one of the units $u_j \in \Gamma^{-1}(u_i)$ is instructed to test u_i. If $a_{ji}=0$, then u_j is appended to set U^0. Otherwise, in case of $a_{ij}=1$, u_i is removed from U^0 and both units u_i and u_j are placed in set U^1. Note, that the units in U^0 at a given time are either all fault-free or all faulty. On the other hand, at least one of each pair of units added to U^1 is surely faulty (but we do not know which one). Therefore, the above process must continue until $|U^0|+|U^1|/2=t+1$. Then, the latest member of U^0 is surely fault-free.

Probabilistic centralized diagnosis

The deterministic diagnostic methods presented in the previous section identify the entire fault set (or a predefined subset of all faults) from the syndrome, provided that certain restrictions on the testing structure and the behavior of the units are satisfied. By contrast, probabilistic diagnosis methods try to locate the faulty units *with high probability* and require no restrictive assumption on the set of faulty units or the structure of the testing assignments (i.e. the concept of diagnosability is not applicable to probabilistic diagnosis). On the other side, probabilistic methods cannot guarantee that a correct and complete diagnosis is made in every situation.

In probabilistic diagnosis a probability model is used to model the operation of fault-free and faulty units. The existing probability models are summarized in Appendix B. On the basis of the chosen probability model, a procedure—which may be heuristic—is used to diagnose a set of units as faulty. The important issues in probabilistic methods are the complexity of the diagnosis procedure and the quality of the diagnoses generated. Many of these methods use probabilistic parameters to describe the behavior of units and to evaluate diagnosis performance. However, the way of obtaining these parameter values in real computing systems is still an important unsolved problem [60]. Another limitation of probabilistic diagnosis algorithms is that they assume the outcome of test performed by different units to be statistically independent. Statistically dependent test results do not make these diagnosis algorithms useless, but they may lower the level of the diagnostic accuracy.

It is clear from the previous sections that if 100 percent test coverage is assumed, then there are numerous deterministic methods available to solve the diagnostic problem. Note, that in particular cases—like one-step t-diagnosable systems—even a unique set of faulty units can be identified. The reason for the introduction of probabilistic fault diagnostic algorithms in the complete-test model is to extract diagnostic information from situations with more than t faulty units and for general testing graph structures.

Scheinerman presented a probabilistic approach with desirable asymptotic properties [61]. In his algorithm, first a core group of non-faulty units is identified. This is accomplished by finding a strongly connected subgraph of G in which all edges are labeled by a 0 test outcome and in which the more than the half of the total units is included. Since in this component each unit has the same fault state (all of them are involved in a circle of 0 edges), they are surely fault-free provided that only the minority of the system units may fail. Other units accessible from the core group via a path of 0 edges are then added to the set of fault-free units. All other units are finally classified as faulty.

The author showed that his algorithm produces asymptotically correct and complete diagnosis in random graphs in which a unit is connected to another unit with a probability $(c \log n)/n$, where $c > 1/(1-p_f)$. A significant weakness of the algorithm that it does not work if a fault-free core group cannot be found. Such situations can easily occur even for a moderate number of faults (e.g. when the testing graph is "cut up" into more than two strongly connected subgraphs by "chains" of faulty units). Scheinerman's work is important, however, since it provided the first proof of asymptotically correct and complete diagnosis.

Using Blough's probability model, Blough et al. presented a complete-test probabilistic diagnostic method in which each unit u_i simply evaluates itself to be fault-free or faulty based on the majority opinion of its testers [62]. Thus, u_i is diagnosed as faulty if and only if

$$|\{u_j : u_j \in \Gamma^{-1}(u_i), a_{ji} = 1\}| > \left|\frac{\Gamma^{-1}(u_i)}{2}\right|$$

The authors were able to demonstrate that asymptotically 100 percent accurate diagnoses can be obtained for hypercube testing topologies, as the size of the system approaches infinity, if $p_f < 0.067$. Additionally, it was shown that asymptotically correct and complete diagnosis can be generated for a special class of testing graphs with $n\omega(n)$, where $\omega(n)$ is any function that grows to infinity, albeit arbitrarily slowly.

Somani and Agarwal introduced three more complex algorithms for the same problem [63, 64]. These algorithms first assign *confidence levels* to indicate how likely a certain processor can be considered as faulty. There are two confidence levels: units are classified as either 'likely good' or 'likely faulty'. The decision between the two classes is based on a majority voting of the testers, just like in Blough's algorithm. Then, the found potentially fault-free units are used to locate at least one definitely fault-free unit (again using a majority voting, this time only taking into account the votes of likely good units). Finally, test results of the definitely fault-free units are propagated and the other units are diagnosed iteratively. The iteration continues until no further diagnosis is required or possible. At the end the algorithms check the consistency of the generated diagnostic image with the symmetric test invalidation model.

The three algorithms differ in the voting technique they use. The first algorithm LDA1 employs majority voting in both phases. The second algorithm LDA2 differs from the previous one only in that unanimous voting among the likely good units is required to identify a faulty processor. The third algorithm LDA3 uses unanimous voting to classify likely good and likely faulty units, and also to find definitely fault-free units. The LDA2 and LDA3 algorithms are meant to be gradually simpler methods than the LDA1 algorithm. Somani and Agarwal described the conditions under which their diagnostic algorithms produce correct and/or complete diagnosis. They also present several examples to show that a good diagnostic accuracy can be obtained with the LDA1 algorithm in $\sqrt{n} \times \sqrt{n}$ mesh topologies.

Probabilistic diagnosis is most useful in situations where the units can be intermittently faulty or the tests have significantly less than 100 percent coverage. Although, probabilistic methods by definition cannot guarantee that the produced diagnosis is accurate, neither can deterministic methods make such a claim in the case of intermittent faults or incomplete tests. The best possible probabilistic method in terms of diagnostic accuracy is the one that outputs the most probable set of faulty units given a syndrome. Blount gave an early approach for the solution of this problem [30]. He described the 0-information tester model and used the common probability model to define a mapping from syndromes to fault patterns. The diagnostic process uses a lookup table, which contains the fault set F that has the largest probability $P(S|F)$ for a given syndrome S. Whenever an observed syndrome needs to be decoded, the lookup table is consulted to find the most probable fault set. Though straightforward, this method is very resource consuming. Given $|E|$ edges and n units, there are $2^{|E|}$ possible syndromes and 2^n possible fault sets. Thus, to create the lookup table, $O(2^{n+|E|})$ calculations and $O(2^{|E|})$ memory locations are required.

Lee used the partial tester and common probability models to produce a more efficient method for finding the most probable diagnosis given a syndrome [65]. In his method, he introduced several heuristics to speed up the search for the most likely fault set by bounding the search tree as early as possible. Although the average behavior of this algorithm is fairly good, its worst-case computational complexity is $O(2^{|F|})$, where F is the set of faulty units identified.

Realizing the limitations of finding the most probable diagnosis, Blough et al. used Blough's probability model to develop an $O(|E|)$ algorithm which is able to achieve asymptotically correct and complete diagnosis in systems of $O(n \log n)$ testing connections [66]. In the algorithm, the number of failed tests on a unit $|\Delta_{\text{in}}|$ is compared to an *a priori* threshold value k. The unit u_i for which $|\Delta_{\text{in}}(u_i)| - k_i$ is maximal (i.e. the unit most exceeding its threshold) is selected and placed in the set of faulty units. Next, all of the test results generated by u_i are set to 1 (to imitate that each of them has failed). The difference between the number of failed test and the threshold is the re-calculated at the units $u_j \in \Gamma(u_i)$. Again, the process looks for the maximal difference to place another unit in the set of located faults. This iteration terminates when there are no more u_k units for which $|\Delta_{\text{in}}(u_i)| > k_i$. The rest of the system is classified as fault-free.

The authors proved that asymptotically correct and complete diagnosis can be obtained if the k_i threshold values local to unit u_i are chosen as follows:

$$k_i = \left\lfloor \frac{p_f + p_{ij}(1-p_f)}{2} |\Gamma^{-1}(u_i)| \right\rfloor .$$

They also describe a heuristic method of choosing better threshold values, based on the following lower-bound estimate of the correct diagnosis:

$$\begin{aligned} P_{\text{correct}} &\geq 1 - n(1-p_f) \sum_{j=k+1}^{N} \binom{N}{j} p_f^j (1-p_f)^{N-j} - \\ &- np_f \sum_{j=0}^{N} \left[\binom{N}{j} (1-p_f)^j p_f^{N-j} \sum_{l=0}^{\min(j,k)} \binom{j}{l} (1-p_d)^l p_d^{j-l} \right], \end{aligned} \tag{1.5}$$

when a common threshold $k_i = k$, failure probability p_f, and detection probability $p_{ij} = p_d$ is used, and $N = \max_{u_i \in U} |\Gamma^{-1}(u_i)|$. To find the best common threshold, the above expression is evaluated for all possible values of k and the threshold resulting the highest probability of correct diagnosis is chosen. Note, that the threshold selected by this method is not necessarily an optimal one.

Dahbura et al. gave a comparison-testing based probabilistic diagnostic algorithm of $O(n^2)$ time complexity [67, 68]. It repeatedly selects the unit that is incident to the largest number of 1-links (failed test), i.e. $\max_{u_i \in U} |\Delta_{\text{in}}(u_i)|$. This unit is placed in the set of diagnosed faults and excluded from the testing graph, as well as its local test results are removed from the syndrome. This procedure is iterated until there are no 1-links in the testing graph. Using an assumed upper bound on the number of faulty units, it was shown that the probability of incorrect diagnosis is extremely small given a completely connected testing assignment. Later, Lee [65] was able to show that this algorithm has the same asymptotic properties as the method from Blough et al. [66] while performing significantly better for several different topologies and probability parameter values. This demonstrates that asymptotic accuracy is not sufficient in itself to indicate the usefulness of a probabilistic diagnostic algorithm.

Lee and Shin presented another set of diagnostic algorithms [69]. They observed that many of the approaches to that time used only partial syndrome information in classifying the fault state of each unit. The authors defines several categories of methods respective to the type of partial syndrome information they used. Then, Lee's probability model was used to calculate *a posteriori* fault probability values for the individual units in each categories of diagnosis. Faulty unit were identified on the basis of these *a posteriori* fault probability calculations. All of the algorithms developed was shown to have the same asymptotic properties as Blough's algorithm. The main advantage of Lee and Shin's work is that their methods find the most probable fault set given a particular type of syndrome information, resulting in a *locally optimal* diagnosis. Their most significant disadvantage is that the syndrome decoding process is relying on the use of probability parameter values.

Fussell and Rangarajan introduced an entirely different type of diagnostic methodology [70]. Given that the testing graph is the subgraph of the processor interconnection graph, the algorithms described so far require each unit to be tested by at least $\log n$ other units for asymptotically correct and complete diagnosis. Fussel and Rangarajan suggested to conduct multiple mutual tests in each pair of units. If the number of these tests on all of the units grows faster than $\log n$, then the same asymptotic properties can be obtained in systems with lower connectivity (e.g., meshes or rings), thus improving on the connectivity bound. Their algorithm uses several stages of comparison testing. In testing stage i, all units are assumed to execute the same test task, then the local test results of each unit are compared to the results of all adjacent units. The testing link between two units u_i and u_j is labeled by $c_{ij} = c_{ji} = 1$ if the tests t_{ij} and t_{ji} gave identical results. In this manner, r independent syndromes are collected at the end of the testing stages. Two thresholds s_v (tester threshold) and k_v (test threshold) are used to classify the units. The algorithm assigns labels to the vertices of the testing graph in two steps. First, a *super label* of value 1 is associated with units for those adjacent to 1-links exceeding the tester threshold s_v. Given r syndromes, r super labels are created. In the second step, a *decision label* of value 1 is associated with units for which the sum of super labels exceeds the test threshold k_v. These units are diagnosed as fault-free, all others are faulty.

The authors chose $k_v = 1$ and a range of values was indicated as being acceptable for s_v. In later works [71, 72], they derive better k_v and s_v threshold values and discuss a hierarchical version of the above algorithm. The hierarchical method conducts testing and diagnosis in *clusters.* A third threshold h_v is used to compute a *hierarchy label* which has similar role in the diagnostic process as the previous two ones. The new method thus preserves the topological flexibility of the previous algorithm, while allowing the number of tests each tester must perform to be tailored to the requirements of the interconnection topology.

Referring to the use of multiple testing stages and multiple syndromes as a *multiple-syndrome* diagnosis method, Lee and Shin derived a *locally optimal* multiple-syndrome diagnosis algorithm using their own probability model [65]. The only change introduced by the authors is the way in which the thresholds k_v and s_v are chosen, but in the other aspects the algorithm is the same as Fussel and Rangarajan's algorithm. Calculations of the *a posteriori* fault probabilities help to derive optimal values for k_v and s_v. Due to the similarities and the optimal thresholds, the algorithm shares the same desirable asymptotic properties as Fussell and Rangarajan's method.

1.2.3 Distributed diagnostic algorithms

In massively parallel computing systems several problems arise employing diagnosis with a centralized control:

- The *space and time complexity* of the diagnosis algorithm increases. The processors cannot transmit their local test results to the central observer simultaneously, thus the communication phase becomes longer. The diagnosis can take place only after the complete syndrome has been collected by the central observer. The computing power of the parallel processors is not utilized during diagnosis (the central observer executes the diagnostic algorithm alone). Yet, the system cannot proceed with its normal tasks, since diagnostic results are necessary to validate the correctness of the computation completed up to that point.

- The *information storage and processing capacity* requirements of the central unit also increase. The duration of processing the larger diagnostic image becomes longer as well. The amount of memory in the central observer is finite, and the time available for the diagnosis process is also bounded (since diagnosis is almost always a real-time task). The existing solutions to these limitations have contradicting effects. Data compression can be applied to reduce the necessary amount of memory space, however, data compression algorithms are fairly complex, computation-intensive methods. On the other hand, large, fast access memory can be used to speed-up diagnosis. A good compromise of these solutions is hard to find [26, 73].

- Due to the finite amount of resources the central observer tends to become *overloaded* and hence diagnosis might be inefficient or even impossible.

- The central observer is exclusively responsible for locking out faulty components, i.e., it is a *diagnostic hard-core.* A failure in this hard-core causes a single-point failure which is inadmissible in a fault-tolerant system.

Since massively parallel systems are distributed, there is an opportunity to use the *distributed observer* approach, where no global monitoring and control is present. There exists no central facility in the coordination or mediation among the processors. The distributed nature of such a diagnostic process requires that the units are able to make local decisions regarding certain tasks like scheduling, resource allocation, etc. These decisions can be based only on the information obtained via normal, unit-to-unit communication links. Moreover, any action taken in response to the detection and location of a fault must be local, and may again involve only normal communication.

The *distributed fault-tolerance* concept was proposed by Kuhl and Reddy [31] to avoid the problems with the centralized observer. As its name implies, distributed fault-tolerance allows diagnosis and subsequent isolation of faulty processing elements in a way that does not centralize the failure handling devices in the system. The requirements for fully distributed fault-tolerance are as follows: each fault-free unit must be able to independently achieve correct diagnosis of all other units in the system. Once a unit has recognized the failure of another one, it can "discriminate" against the faulty processor simply by refraining from further cooperation. Since all fault-free units locate the fault(s) and refuse to interact with the faulty component(s), the erroneous part is in effect isolated from the rest of the system.

A processing element is capable of testing only neighboring units. The local test results must be broadcast throughout the whole system to enable the diagnosis of remote units. All of the messages are forwarded through the interconnection network. Units may rely only on indirect information passed via a neighbor in order to obtain the condition of non-neighbors. Since the model allows a faulty processing element to transmit incorrect test results, or alter and even destroy the information routed through it, fault-free units may only trust data received from neighbors they "personally" tested and found to be fault-free. Cooperation and communication with faulty units is forbidden. For this reason, fault-free processors perform a test before transmitting and after receiving a message to a certain neighbor. If the neighbor was found to be faulty, the message is not sent or the received message is ignored. Due to this mechanism, all of the messages are distributed over a path of fault-free processors to assure their reliability.

Kuhl and Reddy introduced a new diagnosability measure for distributed multiprocessor systems: *t-fault self-diagnosability.* A system is considered to be t-fault self-diagnosable if and only if each fault-free processing element can correctly identify all faulty (and fault-free) units in the system, provided that no more than t faults are present. Faulty units block the inter-processor communication and thus hinder the reliable dissemination of the syndrome among fault-free units. If the system is cut up into isolated areas by faulty units that cannot exchange information, then it is impossible to generate a system-wide diagnostic image. Therefore a multiprocessor system with a testing graph $G_T = (V_T, E_T)$ is t-fault self-diagnosable on the basis of the distributed fault-tolerance concept if and only if $\kappa(G_T) > t$.

The algorithm SELF presented by Kuhl and Reddy in [31] conducts the testing in the system in rounds. A round of testing consists of the execution of each test represented by the testing graph. It is assumed, that no failures occur in the system from starting a round of testing till every fault-free processor completes its diagnosis. This assumption means that the diagnostic algorithm analyzes a static "snapshot" of the status of different components at a fixed point of time.

In the algorithm SELF each unit u_i computes a fault vector containing its conclusions about the condition of each other unit. Initially, the local test results are obtained. The units then label themselves as 'good' and all other units as 'unknown' in their fault vector. They examine and record the local test results in the fault vector. Every unit u_i prepares a message containing the local test results for broadcasting to each $u_j \in \Gamma^{-1}(u_i)$. A unit u_j records any new test results received from a neighbor that it tested and found to be fault-free in the fault vector, and forwards these test results to each $u_k \in \Gamma^{-1}(u_j)$. The messages sent by units tested and found to be faulty are ignored. Note, that diagnostic information travels backward over the directed arcs in the testing graph via only fault-free units. This process iterates at each unit until either their fault vector is complete or contains indications of t faults. In the latter case, the remaining unspecified entries in the fault vector are filled with fault-free classification since at most t faults can coexist in the system.

The SELF algorithm developed for the distributed fault-tolerance concept had some imperfections. The algorithm did not take into consideration faults in the communication links and reintegration of repaired, replaced, or simply recovered faulty components was not supported. An improved algorithm called NEW_SELF dealing with these problems was introduced in [74, 75]. The new algorithm is based on the same theoretical concept, thus the same assumptions and notion presented above are used.

The validation of diagnostic messages possibly altered, destroyed or erroneously issued by faulty units has been changed. A unit u_i accepts a diagnostic message from a neighbor unit u_j only if $u_j \in \Gamma(u_j)$ and it has ascertained that unit u_j is fault-free. This is accomplished by the following mechanism: when unit u_j sends a diagnostic message, unit u_i temporarily stores this message in a buffer, provided u_i has found u_j to be fault-free last time it tested. Upon the next test on u_j, if the result indicates that it is still fault-free, unit u_i confirms all pending messages waiting in the buffer. Of course, this strategy requires the assumption that a unit cannot fail (undetected) and recover from the fault in the interval between two consecutive tests on it. If the self-testing processing elements are equipped with latch circuitry for storing error indications, then even transient faults can be detected by acquiring the condition of these latches.

The possibility of reentry of previously faulty units and/or communication links, or entry of new units adds considerable complexity to the diagnostic process. This complexity is due to the necessity of keeping track of temporal ordering among different messages traveling over the interconnection network. To deal with this problem each unit contains a local clock, and messages incorporate a time stamp. This mechanism is similar to that commonly used in distributed systems for message sequencing and maintaining consistency in distributed databases, but it is complicated by the fact that a failure may cause the unit to lose its old clock value.

The algorithm maintains at each unit u_i two arrays: ACCUSERS$_i$ and ENTRY$_i$. The elements of both arrays are sets of messages stored by u_i for diagnostic purposes. The j-th element of these arrays is associated with the unit u_j. At any time, a fault-free unit u_i will have in ACCUSERS$_i[j]$ a number of messages representing that unit u_k started a message at its local time τ_k indicating that u_j was faulty. The element ENTRY$_i[j]$ will contain messages indicating that unit u_l tested u_j at its local time τ_l and found it to be fault-free. In order to prevent the ACCUSERS and ENTRY arrays from growing boundlessly, messages are added to these arrays only if they represent new and useful information. Any messages superseded by newer information are removed. A unit uses similar criteria to determine whether it should or should not forward a message to its testers, thus ensuring finite message lifetimes (a message never passes through a unit more than once).

A unit u_i tests a neighbor $u_j \in \Gamma(u_i)$ according to some locally determined schedule. In addition to periodically scheduled tests, u_i may test in response to some suspicion concerning unit u_j (e.g., not having communication with u_j for an abnormally long time) or in response to a request for an entry message from unit u_j (which is possibly added to the network or was repaired, replaced, or recovered from a transient failure). If u_i finds u_j to be faulty, it composes a message including the local clock value τ_i of the time of test, and broadcasts it to each $u_k \in \Gamma^{-1}(u_i)$. This is the initiation of the reliable dissemination of the message throughout the network backward along paths in the testing graph. After each test performance u_i increments τ_i. When u_i finds u_j to be fault-free and it has earlier received any message indicating that u_k found u_j to be faulty at its local time τ_k, then it composes and broadcasts an announcement that u_j has recovered from the failure detected by u_k.

The actual diagnosis is accomplished by examining the ENTRY$_i$ array. The presence of a message in ACCUSERS$_i[j]$ does not necessarily imply that u_j is faulty. It may also represent an old failure from which u_j has recovered, but for which no entry message will ever reach u_i. Similarly, the presence of a message in ENTRY$_i[j]$ does not imply that u_j is fault-free, because it is possible that the test u_k unit has failed after sending the message, and subsequent to this u_j has also failed. However, u_i can discover which units are fault-free by performing a simple computation on the ENTRY$_i$ array. First, u_i can directly certify the condition of the units it has tested. Then, for any unit $u_k \in \Gamma(u_i)$ found to be fault-free, if there is a message indicating that u_k has tested u_j and found it to be fault-free, then u_j can be accepted as being fault-free. Also, for any message reporting a successful test on unit u_l by u_k or u_j, then u_l must be fault-free, and so on. The diagnosis is achieved by a simple breadth-first like search of the ENTRY$_i$ array. Upon completion all of the fault-free units generate a correct and complete system-wide diagnostic image, provided the testing graph remains strongly connected when the vertices and edges corresponding to faulty units are removed.

The NEW_SELF algorithm also allows the classification of interconnection links. In order to take the inter-processor communication devices and their potential faults into consideration, the notion of a *link failure domain* was incorporated in the diagnostic model of the algorithm. A link failure domain of a distributed computing system is any single source of failure in the inter-processor communication devices. The *link failure domain set* is the set of all link failure domains for the network. The detection and location of interconnection link faults are accomplished in one of two ways:

1. If a test on a unit u_j performed by unit u_i fails, then it may be due to an error in u_j or in the communication path between u_i and u_j, or both. Therefore, the failure message of u_j must be interpreted by the units receiving it as being possibly indicative of any of these situations.

2. A unit u_i may have difficulties accomplishing normal, non-diagnostic communication with a neighbor u_j due to a faulty inter-processor link. In such a case, if $u_i \in \Gamma^{-1}(u_j)$ it can broadcast a message identical to any other failure message. If $u_i \notin \Gamma^{-1}(u_j)$, then the generated message has the special form of a link failure report.

A processor u_k, receiving a failure report initially treats the message as indicative of a fault in processor u_j, and adds it to ACCUSERS$_k[j]$. If, in fact, u_j is fault-free and only the communication path between u_i and u_j is faulty, then u_k should eventually receive an entry message of another tester of u_j reporting a successful test. Upon receiving this message u_k removes the entry indicating the failure of u_j from ACCUSERS$_k[j]$, and adds a new entry to the array FAULTY_LINKS$_k$.

An adaptive diagnosis approach based on the Meyer and Masson algorithm [57] was introduced by Bianchini and Buskens in [76, 77]. The algorithm, called *Adaptive DSD*, is similar to the centralized adaptive methods in that the testing assignment is determined by the actual fault pattern present in the system. The algorithm is only applicable for systems with a fully connected interconnection topology (these include bus-oriented systems or networked workstations). Unit failures and repairs are considered; link failures are not. The number of faulty units in the fault set is not bounded, the fault-free processing elements correctly diagnose the fault state of all units in the system.

In the algorithm each unit u_x maintains a local array called TESTED_UP$_x$. The array contains n elements, indexed by the unit identifier. Each element of the TESTED_UP$_x$ array contains the identifier of a certified fault-free unit. That is, the entry TESTED_UP$_x[i] = j$ indicates that u_x has received diagnostic information from some fault-free u_i unit confirming that he has tested u_j and found it to be fault-free. The basic mechanism of the Adaptive DSD algorithm is that each unit first tries to identify another fault-free unit. Assume that units are listed in sequential order: $u_0, \ldots, u_{n-1}$. Unit u_x identifies the first fault-free unit subsequent to itself in this ordered list. This is accomplished by testing consecutively units $u_{x+1}, \ldots, u_{x+n-1}$ (modulo n) until a fault-free unit, say unit u_y, is found. Then u_x sends a request to u_y to transmit back the contents of its local TESTED_UP$_y$ array.

A test on the unit u_y is assumed to pass, if u_y has remained fault-free since the last test performed by u_x including the period required for unit u_y to send the TESTED_UP$_y$ array. This guarantees that the diagnostic information represented by TESTED_UP$_y$ is accurate. This information is utilized to update the local data structures of unit u_x in the following manner:

1. the entry TESTED_UP$_x[x]$ is set to y, specifying that unit u_x has tested unit u_y and determined it to be fault-free,

2. all other elements of TESTED_UP$_x$ are updated to the values of corresponding entries in TESTED_UP$_y$.

Note, that the diagnostic information in the TESTED_UP arrays are communicated among units in the reverse direction of tests.

Diagnosis is achieved at unit u_x by following the fault-free paths from u_x to other fault-free units. Results of the analysis of TESTED_UP$_x$ are recorded in array STATE$_x$, where the entry STATE$_x[i]$ represents the diagnosed fault state of unit u_i. Initially, all units are classified as faulty in STATE$_x$. a variable called *node_pointer* (denoted by NP) is set to x, the identifier of the unit performing the diagnosis. The algorithm traverses the forward fault-free paths in the test set, labeling each of the units visited as fault-free. This is accomplished by marking the unit represented by STATE$_x[NP]$ as fault-free. Then setting NP is set to TESTED_UP$_x[NP]$, which is the sequentially next fault-free unit in the ordered list. This procedure is repeated until NP is set to every fault-free unit and returns to x.

Later the same authors applied the Adaptive DSD algorithm for systems using a comparison based unit validation scheme. They omitted explicit tests as being impractical to detect arbitrary failures. Instead, a unit monitors the communication of two other units to decide whether they are reliable or not. Two new models for the behavior of faulty units were defined. On-line distributed diagnostic algorithms that operate correctly under the new failure semantics were also given.

The standard "black box" method for testing consists of applying the input test sequences at the control points and observing the outputs of the tested device. The test passes when the outputs conform to the specification of the system given its inputs. In the model of Buskens and Bianchini, the input of each unit corresponds to diagnostic messages the unit receives, and its outputs are the diagnostic messages the unit generates. Unit u_i attempts to validate the operation of unit u_j by observing its input and output messages.

However, u_i may not able to directly observe the input messages of u_j. In this case it is required to request this information from any unit that sends messages to u_j (e.g., unit u_k). Thus, u_k transmits messages to two different units. There is no guarantee that the two messages will be identical, unit u_k may send message m_1 to u_j and message m_2 to u_i. There are two classes of faulty units depending on the relation of m_1 to m_2: *consistent liars* always send incorrect diagnostic information on response to a request, while *inconsistent liars* arbitrary disseminate correct and incorrect diagnostic information. Faulty units that distribute correct diagnostic information are indistinguishable from fault-free units. Explicit tests are not used to detect faults, redundant message passing provides evidence of faulty behavior instead.

There are two variations of the so called *Robust* algorithms for systems with consistent and inconsistent liars. Every message transmitted by fault-free units during the operation of the algorithms is assumed to be delivered correctly. The receiver of a message can identify the sender of the message. The absence of a message can be detected. Every unit can communicate with every other unit (fully connected topology).

The version of Robust for systems with only consistent liars is very similar to the Adaptive DSD algorithm. If there is only one faulty unit, then the algorithm tries to identify a unique fault-free unit and update the local Syndromes array with the information received from that unit. (The Syndromes array is equivalent to the TESTED_UP array of the Adaptive DSD algorithm.) Units are listed in sequential order: $u_0, \ldots, u_{n-1}$. Unit u_x requests Syndromes information from the next two consecutive units in the ordered list: u_y, u_z. If u_y does not validate with u_z, then u_x requests the Syndromes information from a subsequent unit u_w and identifies the first unit that validates with u_w as fault-free. Diagnostic information from the validated unit is then used to update the local Syndromes array. By this mechanism, the contents of the Syndromes arrays at different nodes will be identical to the TESTED_UP arrays of the Adaptive DSD algorithm for symmetric test invalidation. Consequently, processing of the Syndromes array to achieve diagnosis of each unit is done with the same method of the TESTED_UP array described before.

Diagnosis of inconsistent liars is complicated by the fact that an inconsistent liar may forward valid information in addition to corrupt information. Therefore, in systems with inconsistent liars it is not always possible to uniquely identify a fault-free unit. To ensure that diagnosis is always correct in such situations, multiple units may be validated by a single unit. Therefore, each fault-free unit validates with at least one other fault-free unit. Suppose, there is no more than one faulty unit in the system. The version of Robust for systems with only inconsistent liars first attempts to identify a unique fault-free unit. If it fails, it evaluates two units as fault-free. In either case, the Syndromes array is updated from the found fault-free unit(s).

The algorithm can be generalized to handle at most t inconsistent liars similarly to the case of consistent liars. A validation path of length t must be found to guarantee that at least one fault-free unit is identified. Faulty units may appear on the path if they do not lie conclusively. As such, fault-free units may exist that are not on the validation path. To ensure that fault-free units are not incorrectly diagnosed, every unit validating with the farthest unit on the validation path must be classified as fault-free.

Stahl, Buskens, and Bianchini developed an on-line diagnosis algorithm called *Adapt* on the same theoretical background as the algorithm of Bagchi and Hakimi [78]. The testing assignment is *adaptive*. The diagnostic process is composed of two main steps. In the first step, test connections are established locally by each tester unit to ensure the strong connectivity of the testing graph. Execution of this step occurs in parallel at different PEs to minimize diagnosis latency. Redundant tests are removed in the second step. The algorithm locates any number of faulty units provided that the testing graph of the whole system remains strongly connected (i.e., there are sufficient number of available reliable units in the first step of the algorithm).

The Adapt algorithm does not classify the interconnection links, but it handles blocked communication paths due to its adaptive nature. The tests in the system are periodic. A unit may not fail and subsequently recover undetected during the interval between two consecutive tests by the same tester unit. A message validation scheme, similar to that of the NEW_SELF algorithm is used to ensure that messages received from faulty units are ignored. Both unit failures and repairs are considered. The algorithm consists of three phases. Immediately after a fault event occurs, the algorithm enters into the *Search phase* to reconstruct a strongly connected testing graph. Once the Search phase completes, the algorithm begins the *Destroy phase* to remove redundant tests (i.e., to minimize the number of test connections). Finally, the *Inform phase* is invoked to notify all units about the most recent testing assignment. When the testing assignment is updated, units execute the same diagnosis procedure as described at the Adaptive DSD algorithm.

A unit begins the Search phase when it detects the failure or repair of a neighbor unit. It generates a *Search packet* containing information about the diagnostic event, then it broadcasts the packet in the system. All units receiving a Search packet enter the Search phase and forward the Search packet to other units. Every tester unit entering the Search phase updates its local representation of the testing graph and verifies if it remained strongly connected after the inclusion of the new diagnostic event. If a certain unit cannot find a path to any other system unit, it will attempt to add new tests to the testing graph, and updates the forwarded Search packet is to reflect the changes. At the end of the search procedure the resulting testing graph is guaranteed to be strongly connected. The Search phase is finished when a single Search packet completely traverses the whole network, indicating that all units have accomplished the search procedure.

Completion of a Search packet signals the start of the Destroy phase. During this phase a single *Destroy packet* is circulated to all units of the network, causing each unit to execute a destroy procedure on the testing graph. A unit excludes a test if it can remove the test and still finds a directed path to all other units in the testing graph. A strongly connected testing graph containing no redundant edges exists at the completion of the Destroy phase. Finally, the algorithm begins the Inform phase. In this last phase an Inform packet is circulated to distribute the new testing graph to each unit.

The Search procedure can be executed on several units in parallel. The Destroy procedure, however, must run sequentially to assure that a unit does not remove tests required by another unit for connectivity. Parallelism occurs in the Search phase when multiple Search packets are created for the same fault event by two or more units. Since several Search packets can simultaneously exist in the network, coordination is necessary for correct operation. *Packet dominance* defines a competitive relationship between two packets. Packets are compared on the basis of *syndrome time stamps* generated by the units upon entering the Search phase, included in every Search pocket. Given two packets A and B, packet A is said to be *dominant* (denoted by $A > B$), if either all time stamps in packet A are greater than the corresponding time stamps in packet B, or the time stamps are equal but the sender identification code (also included in the packets) of packet A is greater than of packet B. Two packets A and B are *bidominant* (denoted by $A \leftrightarrow B$) if some of the syndromes in packet A have greater time stamps than the corresponding syndromes in packet B and some of the time stamps in packet B are greater than in packet A.

The dominance and collision rules assure that if multiple Search packets were generated in the network, then only a single Search packet completes. Therefore only one Destroy packet is created, ensuring the destroy procedure is executed sequentially. The collision rules are modified for a Destroy or Inform packet to provide that the Destroy or Inform phase is terminated when a new Search phase begins. If a Destroy or Inform packet is bidominant with the information stored at a certain unit, then a (new) search packet must be created.

In the distributed fault-tolerance concept processing elements were considered to be identical from the diagnostic viewpoint, i.e., the system was homogeneous. A more general, non-homogeneous diagnosis model was presented by Liaw, Su and Malaiya [51]. A processing element in the model can be either a *testing unit* capable of performing tests on neighbors, and a *non-testing unit* only tested by others. Accordingly, interconnection links also have different types. A *test-and-transfer edge* implies a message transfer capability in the direction indicated by the corresponding directed arc in the testing graph, and a testing capability in the opposite direction. A *transfer-only edge* implies a message transfer capability in the given direction, but no test is made via the link represented by this arc. A *test-only edge* indicates that the corresponding link is not capable of transferring data, yet a test can be performed via the link in the direction opposite to the respective directed arc in the testing graph. If there exists both a transfer-only and a test-only edge between two units in the same direction, then they can be logically replaced by a single test-and-transfer edge. In the testing graph, there is at most one edge from a unit to another.

The tests in the model are assumed to be complete, that is the test covers all faults in the tested unit and the interconnection link. If the testing unit is fault-free, but a fault exists in either in the tested unit or the interconnection link between the two units, then the testing unit obtains a failed test result. Otherwise the test passes. The inter-processor communication network is considered to be a passive component having a simple and easy-to-test structure. As long as both of its terminal units are fault-free, the receiving unit can verify the correct operation of an interconnection link with a simple test. A failure of a unit or a link involved in the communication can corrupt any message passed through. In order to protect essential information from unnoticed alterations during transmission, error-correcting codes are proposed. Each message will include a *private identification code* of the sending unit. Each non-testing unit can decode and detect errors in codes, so any information with valid identification code received by a non-testing unit will very likely be correct.

The authors also developed a new diagnosability measure for their extended model. They defined a system to be *unweighted* if all of its elements have equal reliability. When different components are not equally reliable, a positive integer weight may be assigned to each component to reflect their relative reliability. The weight assignment should satisfy the following condition: a set of faulty units with smaller total weight is (assumed to be) more probable to fail than other fault sets with larger total weight. Such a system is called a *weighted* system. A weighted system is *w-weight self-diagnosable* if each fault-free unit can correctly identify all faulty (and fault-free) units in the system, provided that the weight of the fault set present is not greater than w. A *testing path* is a sequence of testing edges in the same direction. The first edge may either be a test-and-transfer or a test-only edge, but all the subsequent edges must be test-and-transfer edges. A *message path* is a sequence of test-and-transfer and transfer-only edges in the same direction, that cover only testing units except a non-testing unit at the receiving end. A set of paths sharing sending and receiving units are *blocked* by a fault set (not including the two terminal units) if none of the paths remain completely connected after the removal of faulty components. The following conditions must be fulfilled by a weighted system to be w-weight self-diagnosable:

- if w_l is the total weight of testing units, then $w \leq \lfloor \frac{w_l+1}{2} \rfloor$,
- from every unit to a testing unit there exists either a test-and-transfer or a test-only edge, or a set of testing paths that cannot be blocked by any fault set with a weight less than w, and
- from every testing unit to a non-testing unit there exists either a test-and-transfer or a transfer-only edge, or a set of message paths that cannot be blocked by any fault set with a weight not greater than w.

Self-diagnosis of a weighted, non-homogeneous system incorporating both testing and non-testing units is achieved in three steps. First, diagnosis of individual units via test links is performed. Testing units record test outcomes in the local syndrome. Then the testing units exchange diagnostic information among each other to complete their diagnosis about the whole system. Finally, testing units broadcast information to non-testing units and let them achieve their diagnosis.

In the first step of the algorithm all of the tests included in the testing graph are executed by testing units. Non-testing units can also verify the transfer-only edges connecting them to testing units. The second step is very similar to the SELF algorithm with some differences: diagnostic messages are only broadcast over test-and-transfer edges, and the process is iterated until either the fault vector is fully specified, or the weight of the identified fault set is reaches or exceeds w (unclassified units are labeled as fault-free). In the third step each testing unit generates a message for each fault-free non-testing unit, containing the completed fault vector and the personal identification code. Testing units only start those messages that can be forwarded over a chain of testing units they diagnosed to be fault-free. Upon receiving a message, non-testing units first decode the identification code to verify the integrity of the information. Valid messages are stored in a stack structure, invalid

Table 1.3: Generalized model of test invalidation

Tester unit	**Tested unit**	**Test result**
fault-free	fault-free	pass
fault-free	faulty	fail
faulty	fault-free	$C \in$ {pass, fail, or arbitrary}
faulty	faulty	$D \in$ {pass, fail, or arbitrary}

messages are ignored. Since faulty testing units may transmit a diagnostic message more than once, such messages are counted and are taken into consideration only when received for the first time. The weight of subsequently received fault vectors is added to the weight of the identical fault vector already recorded in the message stack. If the weight of a certain fault vector on the stack becomes greater than or equal to $w \leq \lfloor \frac{w_l+1}{2} \rfloor$, then the procedure is terminated and the fault vector is accepted. The fault vector selecting process is similar to the algorithm of Meyer and Masson [57].

1.2.4 Generalized test invalidation

In *heterogeneous* systems consisting of various functional units, test invalidation will likely be heterogeneous as well. The *generalized test invalidation* scheme provides a unified framework to handle the differences of the invalidation models of system components [33]. The model is described in Table 1.3. Due to the complete test assumption fault-free units always test other units correctly. Test results of faulty tester unit can have three outcomes: always pass, always fail, or arbitrarily pass/fail independent on the fault state of the tested unit. These results correspond to the constants 0, 1, and X. Nine possible test invalidation models are encompassed by the generalized scheme, denoted by the respective C and D values. For example, symmetric invalidation is referred to as the $\mathrm{T_{XX}}$, and asymmetric invalidation as the $\mathrm{T_{X1}}$ test invalidation model.

The relationship between tester and tested units encapsulated by generalized invalidation can be used to derive *parameterized inference rules.* An inference rule is affected by three main parameters: (1) the test invalidation of the tester unit, (2) the (supposed) fault state of the tester/tested unit, and (3) the actual test outcome. Parameterized inference rules are useful to extract one-step implications from the syndrome in the form of "fault state a of unit u_i implies the fault state b of unit u_j" (denoted by $f_i^a \rightarrow f_j^b$).

Four types of one-step implications exist: tautology, forward and backward implication, duality, and contradiction. A *tautology* implication simply repeats the value of a logical variable: $f_i^x \rightarrow f_i^x$, where $x = \{0, 1\}$. Although they seem to be unnecessary, tautology implications play an important role in the transitive extension of diagnostic knowledge. *Forward* and *backward* one-step implications are derived from the parameterized inference rules, expressing the relationship of two adjacent units. Duality rules are the formal negations of forward and backward implications: if $f_i^x \rightarrow f_j^y$ exists, then $f_j^{\neg y} \rightarrow f_i^{\neg x}$ is also a valid implication. A contradiction provides a *sure* inference. It expresses that either the fault-free or the faulty state of a certain unit is *incompatible* with the syndrome: $f_i^0 \rightarrow f_i^1$ (u_i is surely faulty) or $f_i^1 \rightarrow f_i^0$ (u_i is surely fault-free). The complete set of parameter-

Table 1.4: Parameterized inference rules

		Tester unit		**Tested unit**	**Test result**	**Implication**
Name	**Type**	**fault state**	**invalidation**	**fault state**		
[a.1]	tautology	fault-free				$f_i^0 \rightarrow f_i^0$
[a.2]	forward	fault-free			pass	$f_i^0 \rightarrow f_j^0$
[a.3]	backward		T_{1D}	fault-free	pass	$f_j^0 \rightarrow f_i^0$
[b.1]	forward	fault-free			fail	$f_i^0 \rightarrow f_j^1$
[b.2]	backward		T_{1D}, T_{XD}	fault-free	fail	$f_j^0 \rightarrow f_i^1$
[b.3]	contradiction		T_{0D}	fault-free	fail	$f_j^0 \rightarrow f_j^1$
[c.1]	backward		T_{C0}	faulty	fail	$f_j^1 \rightarrow f_i^0$
[c.2]	forward	faulty	T_{10}, T_{X0}		fail	$f_i^1 \rightarrow f_j^0$
[c.3]	forward	faulty	T_{01}, T_{X1}		pass	$f_i^1 \rightarrow f_j^0$
[c.4]	contradiction		T_{C0}	faulty	pass	$f_j^1 \rightarrow f_j^0$
[c.5]	contradiction	faulty	T_{00}		fail	$f_i^1 \rightarrow f_i^0$
[c.6]	contradiction	faulty	T_{11}		pass	$f_i^1 \rightarrow f_i^0$
[d.1]	tautology	faulty				$f_i^1 \rightarrow f_i^1$
[d.2]	backward		T_{C0}, T_{CX}	faulty	pass	$f_j^1 \rightarrow f_i^1$
[d.3]	forward	faulty	T_{00}, T_{0X}		fail	$f_i^1 \rightarrow f_j^1$
[d.4]	forward	faulty	T_{10}, T_{1X}		pass	$f_i^1 \rightarrow f_j^1$

ized one-step inference rules derived from the general test invalidation model is shown in Table 1.4.

Diagnostic implications can be drawn in a graphical form using the notion of the *inference graph*, as introduced by Definition 1.2.1.

Definition 1.2.1. The inference graph is a directed $G_I = (V, V_I, E_I)$ graph composed of the V partitions symbolizing the U set of units, the set of $v_i^0, v_i^1 \in V_I$ vertices corresponding to the $f_i^0, f_i^1 \in \mathcal{F}$ possible fault states, and the set of $(v_i^x, v_j^y) \in E_I$ directed edges representing the $p_{ij} \in \mathcal{P}$ possible one-step implications between the $u_i, u_j \in U$ units. The V partitioning function assigns a v_i partition to each pair of v_i^0 and v_i^1 vertices: $\{v_i^0, v_i^1\} \mapsto v_i$. A directed edge exists between (v_i^x, v_j^y) iff $f_i^x \rightarrow f_j^y$ is a valid one-step implication extracted from the actual syndrome.

In the graphical representation partitions, nodes and edges correspond to boxes, circles and arrows (see Figure 1.6).

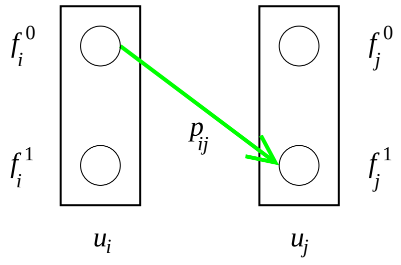
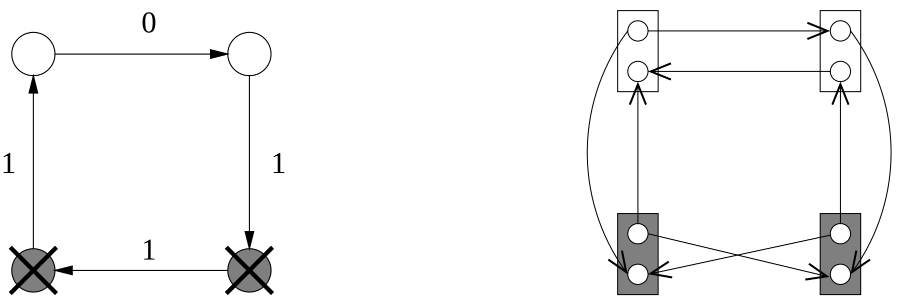


Figure 1.6: Components of the inference graph

Figure 1.7 demonstrates both the application of the inference graph, and the extraction of the diagnostic information from a given syndrome. In Figure 1.7(a) a testing assignment



(a) Example testing graph with syndrome (b) Corresponding inference graph

Figure 1.7: Inference graph creation

with four units (two fault-free and two faulty ones) and four tests can be seen, each unit conforms to the symmetric test invalidation. After applying the parameterized inference rules, the one-step implications displayed in Figure 1.7(b) could be extracted from the four test results.

Two one-step implications can be combined into a two-step implication using the transitive property: if $f_i^a \to f_j^b$, and $f_j^b \to f_k^c$ are two valid one-step implications, then they imply $f_i^a \to f_k^c$. The set of all one-step and multiple-step implications obtained by repeated application of the transitive property is the *transitive closure*. It contains all information that can be extracted from the syndrome.

The use of the transitive closure in the fault classification procedure was first introduced by Selényi [33]. In his algorithm one-step implications are stored in a $2n \times 2n$ $\mathbf{M}$ hypermatrix, called the *inference matrix*. The $\mathbf{M}$ matrix consists of four $n \times n$ binary minor matrices: $\mathbf{M}^{00}$, $\mathbf{M}^{01}$, $\mathbf{M}^{10}$, and $\mathbf{M}^{11}$. The $\mathrm{m}^{xy}[i,j]$ element of the $\mathbf{M}^{xy}$ minor matrix can be expressed as

$$\mathrm{m}^{xy}[i,j] = \begin{cases} 1 & \text{if } f_i^0 \to f_j^0 \text{ is a valid one-step implication,} \\ 0 & \text{otherwise.} \end{cases}$$

The inference matrix has the following three structural properties: the $\mathbf{M}^{00}$ and $\mathbf{M}^{11}$ minor matrices contain the identity matrix ($\mathrm{m}^{00}[i,i] = 1$ and $\mathrm{m}^{00}[i,i] = 1$) due to the tautology implications; and $\mathbf{M}^{00}$ equals to the transposed $(\mathbf{M}^{11})^{\mathrm{T}}$ matrix, while $\mathbf{M}^{01}$ and $\mathbf{M}^{10}$ are symmetric minor matrices, both due to the duality implications. The structure of the inference matrix is illustrated in Figure 1.8.

Transitive closure of the one-step implication set can be computed by the logical closure of the $\mathbf{M}$ matrix. This is achieved by the repeated application of the $\mathbf{M}^{(k+1)} \leftarrow \mathbf{M}^{(k)} \cdot \mathbf{M}^{(k)}$ (the contents of $\mathbf{M}$ in the $(k+1)$-th step is the square of $\mathbf{M}$ in the (k)-th step) iteration. The transitive propagation stops when no new non-zero elements appear in the subsequent iteration steps. In the resulting $\hat{\mathbf{M}}$ transitively closed matrix the non-zero elements in the main diagonal of $\mathbf{M}^{01}$ and $\mathbf{M}^{10}$ indicate a contradiction, and so the corresponding units can be surely classified.

After the identification of the surely diagnosable units, the algorithm employs the diagnostic t-limit to classify the fault state of the remaining processors. For the purpose an undirected graph, which contains an edge from each node u_i to the nodes corresponding to

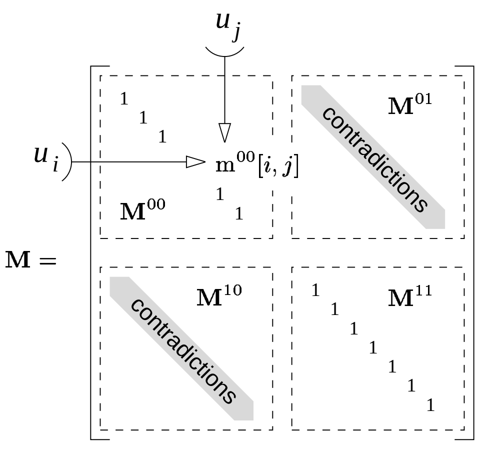


Figure 1.8: Structure of the inference matrix

units in the 'implied faulty' set is created. The 'implied faulty' graph set is obtained directly from the $\mathbf{M}^{01}$ minor matrix after removing the elements corresponding to surely classified units from the $\hat{\mathbf{M}}$ closed inference matrix. It is proven, that the 'implied faulty' graph contains a maximum matching which saturates each node corresponding to faulty units in a one-step t-diagnosable system. The minimum vertex set of this maximum matching is the set of faulty units.

Chapter 2

Centralized fault diagnosis

This chapter introduces a novel centralized probabilistic fault diagnostic approach called *local information diagnosis* (LID). The basic ideas of the new methodology are explained, then the approach is utilized to create three classes of LID algorithms. The classes differ in the extraction, representation and utilization of the diagnostic information in the fault classification process. Certain algorithms operate on the complete set of diagnostic inferences, for these we define three different original heuristic methods of fault classification. The created algorithms and heuristics are compared with each other and other algorithms taken from the literature by analytical and measurement methods. Based on the analytical results and measurement experiences the prospective application areas of the developed methods are outlined.

2.1 Local information diagnosis

The transitive closure is obtained using the implication rules derived from the generalized test invalidation model, and it is the complete source of topology and fault set independent diagnostic information. A diagnostic algorithm based on the transitive closure has the following structure:

1. One-step diagnostic implications are extracted using the parameterized implication rules and the actual syndrome.
2. Multiple-step implications are obtained by transitively combining one-step implications. Inference propagation may continue until all possible implication chains are expanded in full length, that is, the transitive closure is created.
3. All units involved in contradictions found in the transitive closure can be surely classified as fault-free or faulty.
4. Other units are diagnosed by a deterministic or probabilistic fault classification method.

For units whose fault state cannot be surely classified there are two possible diagnostic approaches. Deterministic algorithms usually assume that certain predefined conditions on the possible fault sets and the testing assignment hold. The restrictive conditions allow the diagnostic algorithm to eliminate diagnostic uncertainty by leaving certain fault sets out of consideration. This way a one-to-one correspondence is created between the fault sets and the resulting syndromes, and the generated diagnostic image is guaranteed to be correct and complete when the requirements are justified. However, there is no information on the behavior of the deterministic algorithms outside the valid range of their assumptions, i.e., they may produce *arbitrary* results. Probabilistic methods give a possibility to achieve a satisfactory diagnostic result even where the deterministic approach is useless. They apply fault classification heuristics to determine the fault state of system units. The aim of these heuristics to estimate the most likely fault state, while they must remain simple and computationally efficient at the same time.

There are two main performance bottlenecks of the above outlined procedure. First and foremost, generating the transitive closure of a large inference graph is a computation-intensive task. The underlying idea of *local information diagnosis* (LID) is that a probabilistic algorithm can achieve high probability of diagnostic correctness without expanding the implication chains in full length. The informal explanation of this claim requires us to examine the possible fault configurations.

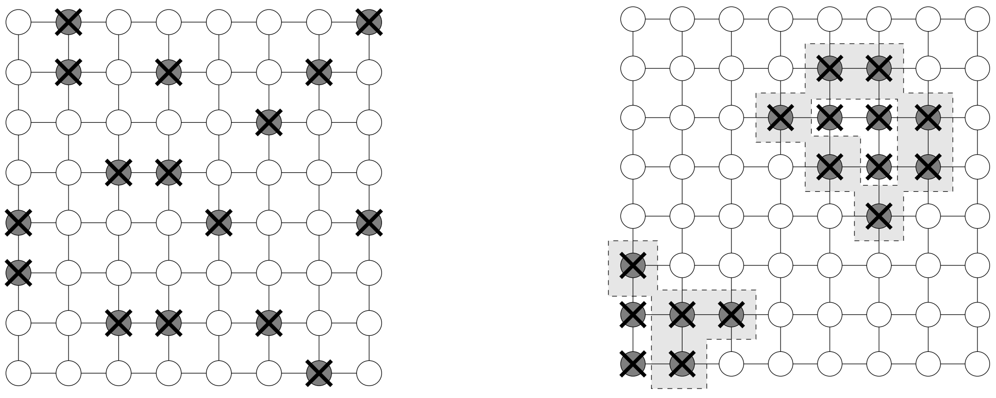
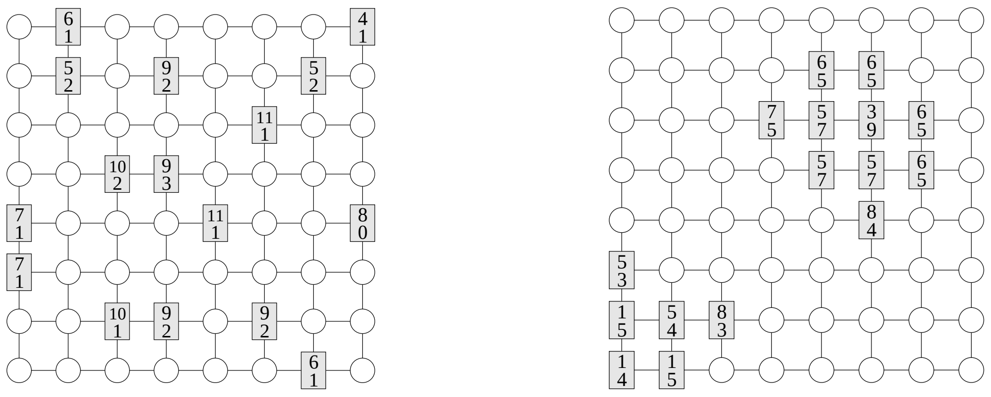

(a) Scattered fault set (b) Group fault set

Figure 2.1: Example of the two main fault pattern types

The possible fault patterns occurring in an MP system can be divided into two main types: (1) the faults are *scattered* throughout the system, separated from each other, and (2) the faults are located close to each other forming a *group*. Examples for both fault set types are shown in Figure 2.1. In most practical cases both situations can be handled using just a portion of the diagnostic information [79]. When faults are separated, the failed test results appear locally in the syndrome. The faulty units are surrounded by fault-free tester processors, thus enough diagnostic information is available to identify the faulty units. In the second case, however, diagnostic uncertainty caused by faulty units "blocks" the propagation of the inferences. In other words, the faults constituting the group border isolate the inside of the group from the rest of the system. The implication chains do not lead into the core of the group.

For this reason, any classification method can only attempt to identify the units on the fault group borders. These peripheral units are surrounded by fault-free testers similarly to separated faults, and can be reliably identified even if only a partial diagnostic information



(a) Ratio of testers (scattered faults)

(b) Ratio of testers (grouped faults)

Figure 2.2: Ratio of the fault-free/faulty tester processors

is extracted from the syndrome. The diagnosis of the fault group core will improve only little when the implication chains are calculated in full length. This phenomenon is illustrated in Figure 2.2. The figure revisits the two example fault patterns introduced in Figure 2.1. In the places corresponding to faulty units in the examples two values can be seen. The upper value gives the number of fault-free tester processors, while the lower value is the number of faulty tester processors located among the 1-neighbors of the given unit. It can be seen, that even in such a limited environment the fault-free tester processors are in majority in the case of individual separated faulty units and units on the borderline of group fault sets.

The other performance bottleneck originates in the classification of those units which are not involved in a contradiction and whose fault state cannot be surely identified. Deterministic algorithms require complex methods for this task, since they must guarantee a correct and complete diagnosis (if only in a restricted set of cases). A typical requirement employed by many traditional diagnostic algorithms is a static upper bound on the number of tolerated faulty units. This diagnostic t-limit is a very serious restriction in large systems, because a significant amount of fault sets consisting of more than t faulty units are unambiguously diagnosable [32]. Moreover, the amount of unambiguously diagnosable faults increases proportionally with the system size.

Along these guidelines the author developed a family of probabilistic diagnostic algorithms based on the local information diagnosis methodology [2, 7]. These simple and efficient algorithms use the *generalized test invalidation* principle making them able to handle a class of heterogeneous systems. They are significantly faster than deterministic algorithms as they analyze only a portion of diagnostic information contained in the transitive closure. Several one of them exploits the regular structure and low local complexity of MP systems by propagating implications only among the neighbor units. The fault classification step uses a simple heuristic rule, solely based on the collected one- and multiple-step implications independent on the number of faulty units. This further reduces the time complexity of the algorithms, and provides good diagnostic accuracy even for fault sets significantly larger than the t-limit.

The four local information diagnostic algorithms subject to this document can be divided into the following three categories [9, 8]:

Limited inference algorithms. Limited inference algorithms use the inference matrix representation of the one- and multiple-step implications derived from the syndrome. The repeated multiplication of the inference matrix transitively propagates the stored implication chains. The underlying idea of limited inference methods is to compute implication chains only in a limited, predetermined length. This way only a subset of the transitive closure is obtained. Units are classified on the basis of this incomplete diagnostic information.

Limited information algorithms. Another method of reducing the diagnostic complexity is to limit the amount of information taken into account during inference propagation. The concept of this approach is to "cut out" the local environment of the unit under diagnosis, and perform a full transitive closure in this restricted area. Thus, the computation-intensive transitive closure computation is performed n times, but only for a small, constant size $\mathbf{M}_k(u_i)$ reduced inference matrix.

Scalar algorithms. Scalar algorithms compute and utilize only the quantity of implications supporting a given fault hypothesis. They do not keep record of the implication chains connecting the fault states. The time and space complexity of this class of algorithms is quite low, while they provide considerably good diagnostic accuracy. On the other hand, the scalar representation obviously results in the loss of diagnostic information. The relationship of non-neighbor units cannot be determined and multiple-step contradictions remain undetected. A further consequence of this information loss is that the fault classification heuristics presented in this paper are not applicable to scalar algorithms. The heuristic classification rule in these methods is included in the form of a specific weight function used in the implication sum calculation.

2.1.1 Inference propagation based approaches

Inference propagation methods use a more verbose description of the diagnostic information than scalar methods: they represent each implication between two arbitrary units instead of only maintaining implication counters. The implications are extended by transitive propagation, and the resulting implication set is used for fault classification.

Limited inference algorithms

Limited inference algorithms process the complete set of diagnostic inferences, but propagate the implication chains only in a limited, predetermined length and classify the units on the basis of this partial transitive closure.

Limited Multiplication of Inference Matrix (LMIM) algorithm. The LMIM algorithm is a simplified variant of the Selényi algorithm [33]. One-step implications are collected and stored in the $2n \times 2n$ $\mathbf{M}$ inference hypermatrix. Recall, that the $\mathbf{M}$ matrix consists of four $n \times n$ binary minor matrices: $\mathbf{M}^{00}$, $\mathbf{M}^{01}$, $\mathbf{M}^{10}$, and $\mathbf{M}^{11}$. The $\mathrm{m}^{xy}[i,j]$ element of the

$\mathbf{M}^{xy}$ minor matrix equals to 1 if there exists a valid $f_i^x \to f_j^y$ one-step implication between units u_i and u_j, otherwise it is 0.

Transitive closure can be computed by the logical closure of the $\mathbf{M}$ matrix. This is achieved by the repeated application of the $\mathbf{M}^{(k+1)} \leftarrow \mathbf{M}^{(k)} \cdot \mathbf{M}^{(k)}$ (the contents of $\mathbf{M}$ in the $(k+1)$-th step is the square of $\mathbf{M}$ in the k-th step) iteration until no new implications appear in the subsequent steps. However, in the LMIM algorithm the $\mathbf{M}$ matrix is raised only to a small, constant power (hence the name of the method), i.e., the iteration is executed only a few, constant times. Thus, the resulting matrix contains only a subset of the diagnostic implications included in the $\hat{\mathbf{M}}$ transitive closure. The subset comprises both the original one-step and transitively propagated multiple-step implications. In the inference matrix representation these two implication types are not distinguished. That is, when in the inference graph there are more than one edge-disjunct paths between two respective fault hypotheses, they exist as a single implication in the inference matrix.

Algorithm 1 The LMIM algorithm

```
Ensure: Propagate $\mathrm{m}^{00}, \mathrm{m}^{01}, \mathrm{m}^{10}, \mathrm{m}^{11}$ implications
  for all $u_i \in U$ do {one-step implications}
    for all $u_j \in \mathrm{N}(u_i)$ do
      $\mathrm{m}^{00}[i] \Leftarrow u_j, \exists p_{ij} : f_i^0 \to f_j^0$
      $\mathrm{m}^{01}[i] \Leftarrow u_j, \exists p_{ij} : f_i^0 \to f_j^1$
      $\mathrm{m}^{10}[i] \Leftarrow u_j, \exists p_{ij} : f_i^1 \to f_j^0$
      $\mathrm{m}^{11}[i] \Leftarrow u_j, \exists p_{ij} : f_i^1 \to f_j^1$
    end for
  end for
  for all $k = 1$ to *number of iterations* do
    for all $u_i \in U$ do {transitive propagation of $p_{ij}$ implications}
      for all $u_j \in U$ do
        for all $u_x \in U$ do
          $\mathrm{m}^{00,(k+1)}[i,j] \Leftarrow \mathrm{m}^{00,(k)}[i,x] \wedge \mathrm{m}^{00,(k)}[x,j] \vee \mathrm{m}^{01,(k)}[i,x] \wedge \mathrm{m}^{10,(k)}[x,j]$
          $\mathrm{m}^{01,(k+1)}[i,j] \Leftarrow \mathrm{m}^{00,(k)}[i,x] \wedge \mathrm{m}^{01,(k)}[x,j] \vee \mathrm{m}^{01,(k)}[i,x] \wedge \mathrm{m}^{11,(k)}[x,j]$
          $\mathrm{m}^{10,(k+1)}[i,j] \Leftarrow \mathrm{m}^{10,(k)}[i,x] \wedge \mathrm{m}^{00,(k)}[x,j] \vee \mathrm{m}^{11,(k)}[i,x] \wedge \mathrm{m}^{10,(k)}[x,j]$
          $\mathrm{m}^{11,(k+1)}[i,j] \Leftarrow \mathrm{m}^{10,(k)}[i,x] \wedge \mathrm{m}^{01,(k)}[x,j] \vee \mathrm{m}^{11,(k)}[i,x] \wedge \mathrm{m}^{11,(k)}[x,j]$
        end for
      end for
    end for
  end for
```

Non-zero elements in the main diagonal of the $\mathbf{M}^{01}$ and $\mathbf{M}^{10}$ minor matrices signify contradictions. For example, if $\mathrm{m}^{01}[i,i]$ equals to 1, then the $f_i^0 \to f_i^1$ implication holds. This implies that the f_i^0 fault hypothesis is not compatible with the syndrome, thus unit u_i is surely faulty. Analogously, all u_j units corresponding to the non-zero $\mathrm{m}^{01}[j,j]$ and $\mathrm{m}^{10}[j,j]$ elements can be surely classified. For other, not surely classified units some heuristic fault classification rule, like those described in Section 2.1.2, must be used to determine their fault state.

Distribution of Inference Lists (DIL) algorithm. The DIL algorithm uses the same representation as the LMIM method. However, it has a better time complexity, because it exploits the properties of regular topologies. These topologies have a constant, relatively low connectivity of nodes, i.e., each processor has only a few neighbors. In such systems, the matrix multiplication used to compute the transitive closure performs a lot of redundant operations, especially in the first few iterations. For example, the $\mathrm{m}^{00}[i,j]$ element of the $\mathbf{M}^{00}$ minor matrix is computed as

$$\mathrm{m}^{00,(k+1)}[i,j] \leftarrow \sum_{x=1}^{n} \mathrm{m}^{00,(k)}[i,x] \cdot \mathrm{m}^{00,(k)}[x,j] + \mathrm{m}^{01,(k)}[i,x] \cdot \mathrm{m}^{10,(k)}[x,j]$$

In the first iteration only one-step implications exist and these can be combined into two-step implications only at neighbor nodes, therefore each multiplication where $u_k \notin \mathrm{N}(u_i) \cap \mathrm{N}(u_j)$ is surplus.

Algorithm 2 The DIL algorithm

```
Ensure: Propagate $\mathrm{m}^{00}, \mathrm{m}^{01}, \mathrm{m}^{10}, \mathrm{m}^{11}$ implications
  for all $u_i \in U$ do {one-step implications}
    for all $u_j \in \mathrm{N}(u_i)$ do
      $\mathrm{m}^{00}[i] \Leftarrow u_j, \exists p_{ij} : f_i^0 \rightarrow f_j^0$
      $\mathrm{m}^{01}[i] \Leftarrow u_j, \exists p_{ij} : f_i^0 \rightarrow f_j^1$
      $\mathrm{m}^{10}[i] \Leftarrow u_j, \exists p_{ij} : f_i^1 \rightarrow f_j^0$
      $\mathrm{m}^{11}[i] \Leftarrow u_j, \exists p_{ij} : f_i^1 \rightarrow f_j^1$
    end for
  end for
  for all $k = 1$ to *number of iterations* do
    for all $u_i \in U$ do
      for all $p_{ij} \in P[i]$ do {transitive propagation of $\mathrm{m}^{xy}$ vectors}
        if $p_{ij} = f_i^0 \rightarrow f_j^0$ then
          $\mathrm{m}^{00,(k+1)}[j] \Leftarrow \mathrm{m}^{00,(k)}[j] \cup \mathrm{m}^{00,(k)}[i]$
          $\mathrm{m}^{10,(k+1)}[j] \Leftarrow \mathrm{m}^{10,(k)}[j] \cup \mathrm{m}^{10,(k)}[i]$
        else if $p_{ij} = f_i^0 \rightarrow f_j^1$ then
          $\mathrm{m}^{01,(k+1)}[j] \Leftarrow \mathrm{m}^{01,(k)}[j] \cup \mathrm{m}^{00,(k)}[i]$
          $\mathrm{m}^{11,(k+1)}[j] \Leftarrow \mathrm{m}^{11,(k)}[j] \cup \mathrm{m}^{10,(k)}[i]$
        else if $p_{ij} = f_i^1 \rightarrow f_j^0$ then
          $\mathrm{m}^{00,(k+1)}[j] \Leftarrow \mathrm{m}^{00,(k)}[j] \cup \mathrm{m}^{01,(k)}[i]$
          $\mathrm{m}^{10,(k+1)}[j] \Leftarrow \mathrm{m}^{10,(k)}[j] \cup \mathrm{m}^{11,(k)}[i]$
        else if $p_{ij} = f_i^1 \rightarrow f_j^1$ then
          $\mathrm{m}^{01,(k+1)}[j] \Leftarrow \mathrm{m}^{01,(k)}[j] \cup \mathrm{m}^{01,(k)}[i]$
          $\mathrm{m}^{11,(k+1)}[j] \Leftarrow \mathrm{m}^{11,(k)}[j] \cup \mathrm{m}^{11,(k)}[i]$
        end if
      end for
    end for
  end for
```

The idea of DIL algorithm is to avoid these redundant operations by propagating inference chains only among neighboring nodes. For every u_i unit four binary vectors: $\mathbf{m}^{00}[i]$, $\mathbf{m}^{01}[i]$, $\mathbf{m}^{10}[i]$, and $\mathbf{m}^{11}[i]$ contain the set of one-step implications according to the local view of u_i (these are the row vectors of the respective minor matrices in $\mathbf{M}$). In every iteration the set of implications is propagated locally at each u_i unit by unifying the appropriate row vectors of u_i with the row vectors of its neighbors (see Algorithm 2). At the end of the process, sure classification is indicated by the i-th components of the $\mathbf{m}^{01}[i]$ and $\mathbf{m}^{10}[i]$ vectors. Unclassified processors are diagnosed with the help of heuristic fault classification rules, similarly to the LMIM algorithm.

Limited information algorithms

The limited information approach uses a different concept to utilize regularity. Instead of limiting the length of implication chains, this approach limits the *area* of inference propagation. That is, one-step implications are collected only among the k-neighbors of the unit to be classified. Units not included in the set of k-neighbors are left out of consideration, as if the implications would come from a smaller system. Then, the reduced implication set is propagated in full length.

Local Transitive Closure (LTC) algorithm. The LTC algorithm calculates a complete transitive closure, but only in a small local environment of the diagnosed units. The diagnosis of every u_i unit begins with the creation of an additional $2\nu_k \times 2\nu_k$ hypermatrix. This reduced $\mathbf{M}_k(u_i)$ matrix includes only the u_i processor and its k-neighbors, and the one-step implications among them. In the second step the transitive closure of the reduced matrix is computed, and the resulting implications are copied back to the $\mathbf{M}$ matrix. Since each $\mathbf{M}_k(u_i)$ matrix contains paths of at most $k+1$ length, the final $\mathbf{M}$ matrix includes implications of at most $2k+1$-step length. Space and time complexity is reduced due to the smaller size of the reduced matrix. The resulting $\mathbf{M}$ matrix is used for sure and heuristic fault classification identically to the LMIM and DIL algorithms.

Algorithm 3 The LTC algorithm

```
Ensure: Propagate $\mathrm{m}^{00}, \mathrm{m}^{01}, \mathrm{m}^{10}, \mathrm{m}^{11}$ implications
  for all $u_i \in U$ do
    for all $u_a \in \mathrm{N}_k(u_i) \cup u_i$ do {one-step implications}
      for all $u_b \in \mathrm{N}(u_a)$ do
        $\mathrm{m}_k^{00}(u_i)[a] \Leftarrow u_b, \exists p_{ab} : f_a^0 \to f_b^0$
        $\mathrm{m}_k^{01}(u_i)[a] \Leftarrow u_b, \exists p_{ab} : f_a^0 \to f_b^1$
        $\mathrm{m}_k^{10}(u_i)[a] \Leftarrow u_b, \exists p_{ab} : f_a^1 \to f_b^0$
        $\mathrm{m}_k^{11}(u_i)[a] \Leftarrow u_b, \exists p_{ab} : f_a^1 \to f_b^1$
      end for
    end for
    repeat {transitive closure of $\mathbf{M}_k(u_i)$}
      $\mathbf{M}_k^{(s+1)}(u_i) \Leftarrow \mathbf{M}_k^{(s)}(u_i) \cdot \mathbf{M}_k^{(s)}(u_i)$
    until $\mathbf{M}_k^{(s+1)}(u_i) = \mathbf{M}_k^{(s)}(u_i)$
    for all $u_j \in \mathrm{N}_k(u_i)$ do {copy back implications}
      $\mathrm{m}^{00}[j] \Leftarrow \mathrm{m}^{00}[j] \cup \mathrm{m}_k^{00}(u_i)[j]$
      $\mathrm{m}^{01}[j] \Leftarrow \mathrm{m}^{01}[j] \cup \mathrm{m}_k^{01}(u_i)[j]$
      $\mathrm{m}^{10}[j] \Leftarrow \mathrm{m}^{10}[j] \cup \mathrm{m}_k^{10}(u_i)[j]$
      $\mathrm{m}^{11}[j] \Leftarrow \mathrm{m}^{11}[j] \cup \mathrm{m}_k^{11}(u_i)[j]$
    end for
  end for
```

2.1.2 Fault classification heuristics

The inference propagation methods described in Section 2.1.1 generate a transitively propagated diagnostic implication set. To achieve fault diagnosis, these diagnostic implications must be processed by a special kind of decision making procedure, called *fault classification.* The fault classification procedure assigns either a 'fault-free' or a 'faulty' label to each diagnosed unit. For this purpose, it must assess the likelihood of the fault hypotheses based on the diagnostic implications.

When a $f_i^a \rightarrow f_j^b$ implication exist and the f_i^a fault hypothesis is fulfilled, then the f_j^b fault hypothesis is guaranteed to be justified as well. Consequently, the number of implications supporting a fault hypothesis can be used as the measure of its likelihood. The only problem is that only one of the two fault hypotheses of a unit can be true, and we do not know which one. Implications related to false fault hypotheses can (and usually do) support other false hypotheses. Therefore, we need to use some (heuristic) method of selecting the considered implications. Recall the example fault situations in Section 2.1. There we saw that the fault-free tester processors dominate the local environment of the diagnosed units. This observation can be used for fault classification as follows: only those implications must be taken into account which originate in a fault-free state, and then some form of majority decision must be applied. It is easy to see, that this heuristic approach performs well in the two situations mentioned in Section 2.1, namely for separated faulty units and fault group borders.

The quality of the employed fault classification heuristic significantly affects diagnostic accuracy. We developed three alternative heuristic methods of fault classification and compared their diagnostic performance. The three fault classification heuristics called *Majority*, *Election*, and *Clique* presented in this section are all based on the above outlined decision making procedure. However, each heuristic uses the assumption of local fault-free tester dominance differently. The heuristics can be applied on the implication sets computed by any of the previously presented inference propagation methods: LMIM, DIL, or LTC.

Majority heuristic

The idea of Majority heuristics is simple: the implications from the fault-free states (stored in the $\mathbf{M}^{00}$ and $\mathbf{M}^{01}$ minor matrices) are counted and compared. The $f_j^0 \rightarrow f_i^0$ and $f_j^0 \rightarrow f_i^1$ implications ($j = 1, 2, \ldots, n$) can be interpreted as votes for the fault-free and faulty state of the u_i unit, respectively. The fault classification is simply a majority decision between the votes for the fault-free/faulty state. The sum of votes, i.e., the sum of $f_j^0 \rightarrow f_i^0$ and $f_j^0 \rightarrow f_i^1$ implications can be calculated by counting the non-zero elements stored in the i-th column of the $\mathbf{M}^{00}$ and $\mathbf{M}^{01}$ matrices (see Figure 2.3).

Comparing the two sums $\Sigma^0[i] = \sum_j \mathrm{m}^{00}[j,i]$ and $\Sigma^1[i] = \sum_j \mathrm{m}^{01}[j,i]$, the unit is diagnosed as faulty if $\Sigma^0[i] < \Sigma^1[i]$, otherwise it is fault-free. In a system with more fault-free than faulty units and a completely connected testing graph the Majority heuristic will always correctly classify the fault state of all units (see also Theorem 2.2.10 in Section 2.2). The pseudo code of the Majority heuristic is described in Algorithm 4.

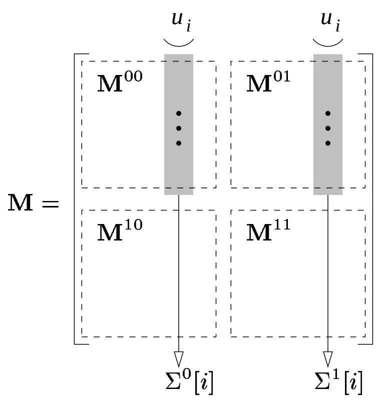


Figure 2.3: Calculation of the $\Sigma^0[i]$ and $\Sigma^1[i]$ values

Algorithm 4 The Majority heuristic

```
Require: filled $\mathbf{M}$ inference matrix
Ensure: fault classification of $\forall u_i \in U$
  for all $u_i \in U$ do {initialization}
    $\Sigma^0[i] \Leftarrow 0$
    $\Sigma^1[i] \Leftarrow 0$
  end for
  for all $u_i \in U$ do {compute votes}
    for all $u_j \in U$ do
      if $\mathrm{m}^{00}[j,i] = 1$ then
        $\Sigma^0[i] \Leftarrow \Sigma^0[i] + 1$
      else if $\mathrm{m}^{01}[j,i] = 1$ then
        $\Sigma^1[i] \Leftarrow \Sigma^1[i] + 1$
      end if
    end for
  end for
  for all $u_i \in U$ do {classification}
    if $\Sigma^0[i] < \Sigma^1[i]$ then
      $u_i$ is classified as faulty
    else
      $u_i$ is classified as fault-free
    end if
  end for
```

Election heuristic

The Election heuristic applies the mechanism of the CFT algorithm [2] to limited inference methods. The basic idea is to identify the faulty units sequentially one-by-one. Units are ranked according to the likelihood of them being faulty for the purpose of selection, and in each identification step the unit with the highest ranking is diagnosed as faulty. Then, the diagnostic uncertainty is decreased by removing the useless and confusing implications originating in the actually located faulty unit. Naturally, ranking must be recomputed each time the set of diagnostic implications is changed.

The CFT algorithm extends the idea presented by Dahbura et al. in [68]. At first the number of test failed on each u_i unit is counted and stored in the FT[i] (failed tests count) variable. Those units for which there exists at least one failed test are considered as *suspect* units (the set of suspect units is denoted by U^{S}). If there are no suspect units the algorithm terminates. Otherwise, it selects the $u_m \in U^{\mathrm{S}}$ unit with a maximum FT[m] value. Since the number of faulty units is relatively low, the selected unit is faulty with a high probability. When multiple units have maximum FT[m] values, then the one having a minimum NFT[m] (count of failed tests on neighbors) value will be selected. The NFT[m] value is the sum of failed tests on the testers of unit u_m. The choice of minimum NFT[m] assures that the testers of the selected unit are most likely to be fault-free. The u_m unit is then added to the U^1 set of faulty units. The unit and its test results are removed from the FT[i] and NFT[i] counters of each remaining $u_i \in U^{\mathrm{S}}$ suspect unit, and the next iteration of the selection procedure starts. When there are no more failed tests the remaining units are classified as fault-free.

Algorithm 5 The Election heuristic

```
Require: filled $\mathbf{M}$ inference matrix
Ensure: fault classification of $\forall u_i \in U$
  $U^{\mathrm{S}} \Leftarrow \emptyset$
  for all $u_i \in U$ do {initialization}
    $\mathrm{LF}[i] \Leftarrow \Sigma^1[i] - \Sigma^0[i]$
    if $\Sigma^1[i] > 0$ then
      $U^{\mathrm{S}} \Leftarrow U^{\mathrm{S}} \cup u_i$
    end if
    $\mathrm{NLF}[i] \Leftarrow \sum_j \mathrm{LF}[j] : u_j \in \Gamma^{-1}(u_i)$
  end for
  $U^{\mathrm{F}} \Leftarrow \emptyset$
  while $\exists i, j : \mathrm{m}^{01}[i,j] \neq 0$ do {election}
    for all $u_m \in U^{\mathrm{S}}$ do
      find $u_m$ with $\max_1^n \mathrm{LF}[m] \wedge \min_1^n \mathrm{NLF}[m]$
    end for
    $U^{\mathrm{F}} \Leftarrow U^{\mathrm{F}} \cup u_m$
    $U^{\mathrm{S}} \Leftarrow U^{\mathrm{S}} - u_m$
    for all $u_i \in U^{\mathrm{S}}$ do
      $\mathrm{m}^{00}[m,i] \Leftarrow 0$
      $\mathrm{m}^{01}[m,i] \Leftarrow 0$
      recalculate $\Sigma^0[i]$ and $\Sigma^1[i]$
      recalculate LF[$i$] and NLF[$i$]
    end for
  end while
  for all $u_i \in U$ do {classification}
    if $u_i \in U^{\mathrm{F}}$ then
      $u_i$ is classified as faulty
    else
      $u_i$ is classified as fault-free
    end if
  end for
```

The procedure of the Election heuristic is defined in Algorithm 5. The likelihood LF$[i]$ of the faulty state of unit u_i is estimated as LF$[i] = \Sigma^1[i] - \Sigma^0[i]$. For ranking units with identical LF values, the likelihood NLF$[i]$ of the faulty state of units testing u_i is also counted: NLF$[i] = \sum_j$ LF$[j]$ for each $u_j \in \Gamma^{-1}(u_i)$. The units are sorted to find the unit u_m most likely to be faulty with the most reliable testers, i.e., having the maximum LF$[m]$ and the minimum NLF$[m]$ values. The u_m unit is then added to the Φ set of faulty units. The unit and its $f_m^0 \rightarrow f_i^1$ implications are removed from the $\mathbf{M}$ inference matrix, and the entire selection procedure starts again. When there are no more implications in the $\mathbf{M}^{01}$ minor matrix the remaining units are classified as fault-free.

Clique heuristic

The Clique heuristic is based on the diagnostic algorithm by Maestrini et al. [80]. The concept is similar to the Majority heuristic: if some fault-free units could be located, then their test results could reliably identify the fault state of other units. However, instead of comparing the feasibility of the fault-free/faulty states individually, the algorithm tries to group the units into two separate cliques. Since in the worst case each clique can contain only one element, clique generation must be done on a per unit basis. The *friendly* clique C$^0[i]$ of unit u_i contains units with a fault state identical to u_i (they are either all fault-free or faulty), while the *foe* clique C$^1[i]$ groups units with a fault state opposite to u_i (if u_i is fault-free, then they can only be faulty, and vice versa). Obviously, the clique sets of neighbor fault-free units are identical.

Cliques are initialized using the implications in the $\mathbf{M}^{00}$ and $\mathbf{M}^{01}$ minor matrices. Clique membership is then extended using the following two rules: (1) "my friend's friend is my friend", and (2) "my friends foe is my foe". The other two possible rules: (3) "my foe's friend is my foe", and (4) "my foe's foe is my friend" are not used, since they could lead to inconsistent cliques due to faulty units. Then the algorithm searches for the u_m unit with a maximum cardinality C$^0[m]$ set and minimum cardinality C$^1[m]$ set. The units belonging to the C$^0[m]$ set are called the *Fault-Free Core*, they are classified as fault-free. Units in the C$^1[m]$ set are diagnosed as faulty. Since some parts of the system can be separated by faulty units, there can be units neither contained in the C$^1[m]$ set nor in the C$^1[m]$ set. These units get the *unknown* classification, i.e., the Clique heuristic may lead to an *incomplete* diagnostic image. The pseudo code of the method can be seen in Algorithm 6.

2.1.3 Scalar approaches

The decision factor in the case of scalar algorithms is the number of implications supporting a given fault hypothesis. These algorithm achieve a significant time and space complexity reduction by not representing the implication chains themselves. This is advantageous from the efficiency viewpoint, but obviously results in further loss of diagnostic information. As a consequence, the relationship of non-neighbor units cannot be determined and higher order contradictions remain undetected. To compensate for this effect additional information—like a weight function—must be included in these methods.

Count Inference Paths (CIP) algorithm. The CIP algorithm estimates the likelihood

Algorithm 6 The Clique heuristic

```
Require: filled $\mathbf{M}$ inference matrix
Ensure: fault classification of $\forall u_i \in U$
  for all $u_i \in U$ do {initialization}
    $\mathrm{C}^0[i] \Leftarrow \emptyset$
    $\mathrm{C}^1[i] \Leftarrow \emptyset$
    for all $u_j \in U$ do
      if $\mathrm{m}^{00}[i,j] = 1$ then
        $\mathrm{C}^0[i] \Leftarrow \mathrm{C}^0[i] \cup u_j$
      else if $\mathrm{m}^{01}[i,j] = 1$ then
        $\mathrm{C}^1[i] \Leftarrow \mathrm{C}^1[i] \cup u_j$
      end if
    end for
  end for
  repeat {clique closure}
    for all $u_i \in U$ do
      for all $u_j \in \mathrm{C}^0[i]$ do
        $\mathrm{C}^{0,(k+1)}[i] \Leftarrow \mathrm{C}^{0,(k)}[i] \cup \mathrm{C}^{0,(k)}[j]$
        $\mathrm{C}^{1,(k+1)}[i] \Leftarrow \mathrm{C}^{1,(k)}[i] \cup \mathrm{C}^{1,(k)}[j]$
      end for
    end for
  until $\mathrm{C}^{0,(k+1)}[i] = \mathrm{C}^{0,(k)}[i] \wedge \mathrm{C}^{1,(k+1)}[i] = \mathrm{C}^{1,(k)}[i]$
  for all $u_m \in U$ do {fault-free core selection}
    find $u_m$ with $\max_1^n |\mathrm{C}^0[m]| \wedge \min_1^n |\mathrm{C}^1[m]|$
  end for
  for all $u_i \in U$ do {classification}
    if $u_i \in \mathrm{C}^0[m]$ then
      $u_i$ is classified as fault-free
    else if $u_i \in \mathrm{C}^1[m]$ then
      $u_i$ is classified as fault-free
    else
      $u_i$ is classified as unknown
    end if
  end for
```

of a fault hypothesis as the number of implications supporting it [3]. For this purpose the algorithm maintains two counters $\Sigma^0[i]$ and $\Sigma^1[i]$ at each u_i unit, corresponding to the weighted number of edges in implication chains ending in the fault-free state f_i^0 and the faulty state f_i^1 of u_i in the inference graph. The $\Sigma^0[i]$ and $\Sigma^1[i]$ numbers are calculated by an iterative algorithm. The initial value of the counters is set using the one-step implications collected from the syndrome. One-step contradictions are detected during the initialization process, and they are used to surely classify the affected units. For processors without sure classification the number of the implications supporting both fault states of the unit are counted. The algorithm is outlined in Algorithm 7.

In the $(k+1)$-th iteration the counters are increased appropriately by the number of paths added to the counters of neighbor units in the k-th step. This *Update* mechanism is described in Algorithm 8. The value of the added paths is multiplied by the $W(p_{ij})$

Algorithm 7 The CIP algorithm

```
for all $u_i \in U$ do
  for all $p_{ij} : u_j \in \Gamma(u_i) \cup \Gamma^{-1}(u_i) \cup u_i$ do
    if $p_{ij} = f_i^0 \rightarrow f_i^1$ then {contradiction}
      surely classify $u_i$ as faulty
    else if $p_{ij} = f_i^1 \rightarrow f_i^0$ then {contradiction}
      surely classify $u_i$ as fault-free
    else {one-step implications}
      $P[i] \Leftarrow P[i] \cup p_{ij}$
    end if
  end for
end for
for all iteration do
  for all $u_i \in U$ do
    for all $p_{ij} \in P[i]$ do
      if $u_i$ is surely classified then
        surely classify $u_j$
      else {count edges in implication chains}
        compute $\Sigma^0[i], \Sigma^1[i]$
      end if
    end for
  end for
end for
```

weight of the $p_i j$ implication connecting the unit and its neighbor. The W weight function is set to compensate for the effect of incorrect implications corresponding to faulty units. In the subsequent iteration steps all implication chains of length 2, 3, ..., k are added to the calculation. The set of surely classified units is also extended using the implications drawn from the already surely classified fault states. After the given number of iterations the remaining unclassified units are diagnosed as faulty if $\Sigma^1[i] > \Sigma^0[i]$, otherwise they are assumed to be fault-free.

There are two variants of the CIP algorithm. The above outlined method is referred to as CIP-2, because it uses two counters to calculate the weighted number of edges in the $f_j^x \rightarrow f_i^0$ and $f_j^y \rightarrow f_i^1$ (where $x, y \in \{0, 1\}$ and $j = 1, 2, \ldots, n$) type of implication chains. The refined CIP-4 variant maintains four counters called $\Sigma^{00}[i]$, $\Sigma^{01}[i]$, $\Sigma^{10}[i]$, and $\Sigma^{11}[i]$. Now it is possible to separately calculate the $f_j^0 \rightarrow f_i^0$, $f_j^0 \rightarrow f_i^1$, $f_j^1 \rightarrow f_i^0$, and $f_j^1 \rightarrow f_i^1$ type of implication chains.

Furthermore, the refinement also reduces the number of circuits in the traversed inference paths, which results in a less distorted computation of the implication counts. Due to the mechanism of the CIP counting method, the local differences of the implication counters "travel along" the directed edges of the inference graph in each iteration step. While this mechanism works perfectly in a circuit-free graph, the "difference packets" get stuck in a circuit. Thus, they circulate endlessly, and get counted again in each l-th iteration step (assuming l is the length of the circuit). Note, that circuits in the inference graph represent tautology implications, and so they do not provide useful diagnostic information. This

Algorithm 8 Update phase of the CIP algorithm

```
Require: $P[i]$ set of one-step implications
Ensure: compute $\Sigma^0[i], \Sigma^1[i]$
  for all $u_i \in U$ do {initialization}
    $\Sigma^{0,(-1)}[i] \Leftarrow -1$
    $\Sigma^{1,(-1)}[i] \Leftarrow -1$
    $\Sigma^{0,(0)}[i] \Leftarrow 0$
    $\Sigma^{1,(0)}[i] \Leftarrow 0$
  end for
  for all $k = 1$ to *number of iterations* do
    for all $u_i \in U$ do
      for all $p_{ij} \in P[i]$ do {count edge increments}
        if $p_{ij} = f_i^0 \rightarrow f_j^0$ then
          $\Sigma^{0,(k)}[j] \Leftarrow \Sigma^{0,(k-1)}[j] + W(p_{ij}) \cdot (\Sigma^{0,(k-1)}[i] - \Sigma^{0,(k-2)}[i])$
        else if $p_{ij} = f_i^0 \rightarrow f_j^1$ then
          $\Sigma^{1,(k)}[j] \Leftarrow \Sigma^{1,(k-1)}[j] + W(p_{ij}) \cdot (\Sigma^{0,(k-1)}[i] - \Sigma^{0,(k-2)}[i])$
        else if $p_{ij} = f_i^1 \rightarrow f_j^0$ then
          $\Sigma^{0,(k)}[j] \Leftarrow \Sigma^{0,(k-1)}[j] + W(p_{ij}) \cdot (\Sigma^{1,(k-1)}[i] - \Sigma^{1,(k-2)}[i])$
        else if $p_{ij} = f_i^1 \rightarrow f_j^1$ then
          $\Sigma^{1,(k)}[j] \Leftarrow \Sigma^{1,(k-1)}[j] + W(p_{ij}) \cdot (\Sigma^{1,(k-1)}[i] - \Sigma^{1,(k-2)}[i])$
        end if
      end for
    end for
  end for
```

problem does not occur in inference propagation methods due to the different representation. Scalar methods maintain only one-step implications and therefore cannot detect and avoid circuits. Nevertheless, as the measurement experiments proved, the CIP algorithms achieve good diagnostic accuracy despite the distortion introduced by circuits.

The CIP-4 algorithm is otherwise completely identical to the CIP-2, only the Update routine must be adjusted to handle the four counters instead of two. The fault classification step is also different: the remaining (not surely classified) units are diagnosed as faulty if $\Sigma^{01}[i] > \Sigma^{00}[i]$, other units are fault-free. This way only the implications originating in fault-free unit states are considered, providing a better estimation of the likelihood of fault hypotheses. Note, that in the case of both the CIP-2 and CIP-4 scalar algorithms the fault classification heuristics introduced in Section 2.1.2 are not applicable. The “fine-tuning” of these methods can be achieved by modifying the elements of the W weight function.

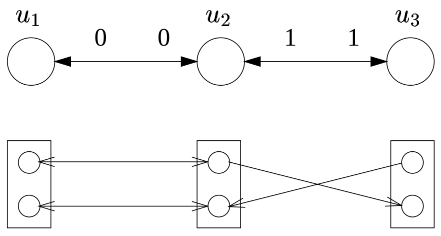


Figure 2.4: Example for determining the W weight function

Table 2.1: Properties of the LID algorithms

Algorithm	**CFT**	**CIP**	**LMIM**	**DIL**	**LTC**
Type	scalar	scalar	limited inference	limited inference	limited information
Time complexity	$O(\phi(n+c\phi)\nu)$	$O(n\nu)$	$O(n^3)$	$O(n^2\nu)$	$O(n\nu_k^3)$
Space complexity	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n\nu_k^2)$
Implication set	$O(1)$	$O(k)$	$O(2^k)$	$O(k)$	$O(k)$
Invalidation model	symmetric	general	general	general	general
Propagation	no	local	global	local	local
Heterogeneous	no	yes	yes	yes	yes

Determining the elements of the W weight function is explained in Figure 2.4. The figure shows a diagnostic situation with three units of which u_1, u_2 are fault-free, and u_3 is faulty. On the corresponding inference graph it can be seen that there is only one implication supporting the fault-free state of u_2, while two implications support its faulty state. This imbalance results from the duality implication which infers that if $f_2^0 \to f_1^0$, then also $f_1^1 \to f_2^1$. To compensate for this imbalance we set the elements of the W weight function as follows

$$W(p_{ij}) = \begin{cases} 2 & \text{if } p_{ij} \text{ is among the parameterized inference rules } \{\,[\text{a.2}], [\text{a.3}], [\text{c.2}], [\text{c.3}]\,\} \\ 1 & \text{otherwise.} \end{cases}$$

With this selection the $f_i^0 \to f_j^0$ implications get higher weight, and consequently fault-free testers of the diagnosed unit do not get outvoted by the faulty testers. Note, that the same considerations apply for the CIP-2 and CIP-4 algorithm, therefore the W weight function is identical in both cases. Although this simple setup of the weight function performed considerably well in the measurement experiments, the detailed analysis of diagnostic mistakes could probably result a more sophisticated setup which could provide better diagnostic accuracy.

2.2 Algorithm properties

The main properties of the presented local information diagnosis algorithms are summarized in Table 2.1. This section explains the derivation of these theoretical values and also presents correctness proofs for the operation of the transitive inference propagation and implication counting methods. The table also contains the CFT algorithm (developed by the author), which is a good example of the existing probabilistic methods and is included for comparison.

Time complexity. The fastest methods are the CIP algorithms which propagate the information among the adjacent neighbors locally, therefore their complexity depends only on the $\max_{u_i \in u} \nu(u_i)$ maximum connectivity of the testing graph, and it is linear respective to the system size. The complexity of the CFT algorithm is additionally determined by the ϕ number of faulty units, but it is also linear with regard to the system size. The LTC algorithm is the third linear method considering the number of units, but its complexity depends polynomially on the $\nu_k = \max_i \nu_k(u_i)$ number of k-neighbors examined, and so it cubically depends on the amount of diagnostic information considered during diagnosis. Note, that the ν and ν_k values are constant in the function of system size, so the CFT, CIP, and LTC algorithms have essentially $O(n)$ time complexity. The DIL method propagates binary vectors among adjacent units requiring $O(n)$ operations, but since the vectors are composed of n elements the total time complexity is $O(n^2)$. The LMIM algorithm has relatively low performance due to its cubical complexity which results from the redundant operations making it topology independent.

Space complexity. The values reflect the amount of data manipulated for diagnostic purposes, not including the syndrome. For large systems, and in case of small number of considered k-neighbors the LTC algorithm has the best space complexity among the inference propagation based approaches, but although the scalar CIP algorithms have also linear space complexity they require less memory.

Number of diagnostic implications. This row of the table shows the size of the transitively propagated implication set after k iteration steps. (CFT is not an iterative method.) In this respect LMIM is far more effective than the other algorithms, since it increases exponentially the length of the considered implication chains, and yet it has linear time complexity respective to the number of iterations. The LTC algorithm is the worst among LID methods in this respect: it can only linearly increase the amount of diagnostic information at the cost of cubical time complexity.

Other characteristics. The LID algorithms employ the generalized test invalidation model, handle arbitrary system topology and work in heterogeneous systems. On the other hand, the conventional CFT method takes only the primary diagnostic inferences of symmetric invalidation, therefore its performance cannot be improved in less restricted invalidation models.

In the following we present several lemmas and theorems which prove the correctness of the LMIM, LTC, and DIL inference propagation algorithms, confirm the validity of the values related to the propagated diagnostic information as included in Table 2.1, and provide insightful analysis of the operation of the developed diagnostic methods.

Lemma 2.2.1. *The transitively propagated implication set computed by the LMIM algorithm in the k-th iteration step contains every implication chain of at most 2^k-step length or less.*

Proof. Let a and b be two non-negative numbers. Define the *binary mapping* of a to b, denoted by $a \overset{1}{\longmapsto} b$, as

$$b = \begin{cases} 1 & \text{if } a \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

Define the binary mapping of two $n \times m$ matrices $\mathbf{A}$ to $\mathbf{B}$ as the binary mapping of the corresponding elements: $\forall i, j : a_{ij} \overset{1}{\longmapsto} b_{ij}$. Consider the adjacency matrix $\mathbf{A}$ of the inference graph belonging to the diagnosed system, arranged in the following way

$$a_{ij} = \begin{cases} 1 & \text{if} \quad f_i^0 \to f_j^0 & \text{and} \quad i, j = 1, 2, \ldots, n \\ 1 & \text{if} \quad f_i^0 \to f_{j-n}^1 & \text{and} \quad i = 1, 2, \ldots, n; \ j = n+1, n+2, \ldots, 2n \\ 1 & \text{if} \quad f_{i-n}^1 \to f_j^0 & \text{and} \quad i = n+1, n+2, \ldots, 2n; \ j = 1, 2, \ldots, n \\ 1 & \text{if} \quad f_{i-n}^1 \to f_{j-n}^1 & \text{and} \quad i, j = n+1, n+2, \ldots, 2n \\ 0 & \text{otherwise.} \end{cases}$$

It is easy to see, that the $\mathbf{M}$ inference matrix is the binary mapping of the $\mathbf{A}+\mathbf{I}$ matrix (where $\mathbf{I}$ is the identity matrix). Since the algebraic operations involved in matrix multiplication and inference propagation do not change the binary mapping relation, the relationship of the inference matrix and the modified adjacency matrix remains the same for any common powers of $\mathbf{M}$ and $\mathbf{A}+\mathbf{I}$. Therefore, $(\mathbf{A}+\mathbf{I})^k \overset{1}{\longmapsto} \mathbf{M}^k$. The k-th power of the adjacency matrix $\mathbf{A}^k = [a_{ij}^{(k)}]$ contains in its $a_{ij}^{(k)}$ element the number of the implication chains of exactly k length from f_i^x to f_j^y [81]. Consequently, the k-th power of the adjacency matrix $(\mathbf{A}+\mathbf{I})^k$ contains in its elements the number of the implication chains of *at most k* length [33]. Since the LMIM algorithm in the k-th iteration step computes the 2^k-th power of the $\mathbf{M}$ inference matrix, and $(\mathbf{A}+\mathbf{I})^{2^k} \overset{1}{\longmapsto} \mathbf{M}^{2^k}$, the $\mathbf{M}^{2^k}$ matrix contains 1 in its $m_{ij}^{(2^k)}$ element if there is at least one implication chain starting at f_i^x and ending at f_j^y, and 0 otherwise. □

Theorem 2.2.2. *The LMIM algorithm, if it is run in an adequate number of iterations, generates the transitive closure of the $\boldsymbol{M}$ inference matrix.*

Proof. The proof trivially follows from Lemma 2.2 and the fact that the diagnosed system and therefore its inference graph is finite. An alternative proof can be found in [82]. It is shown that the Boole AND and OR operators form a closed semiring over the $\{0,1\}$ set, and the authors prove using finite induction that the $\mathbf{M}^{(k+1)} \Leftarrow \mathbf{M}^{(k)} \cdot \mathbf{M}^{(k)} = (\mathbf{A}+\mathbf{I})^{(k)} \cdot (\mathbf{A}+\mathbf{I})^{(k)}$ iteration eventually terminates resulting the transitive closure of $\mathbf{M}$. □

Lemma 2.2.3. *The transitively propagated implication set computed by the LTC algorithm in the k-th iteration step contains every implication chain of at most $2k+1$-step length or less.*

Proof. The LTC algorithm constructs for each $u_i \in U$ unit the $\mathbf{M}_k(u_i)$ inference matrix of the $G_{I,k}(u_i) = (V_k(u_i), E_k(u_i))$ is a reduced inference graph composed of the $V_k(u_i)$ vertices representing the $u_x \in \mathrm{N}_k(u_i)$ k-neighbor units of u_i and the $E_k(u_i) = \{p_{ij} : p_{ij} \in \mathcal{P};\ u_x, u_y \in \mathrm{N}_k(u_i)\}$ one-step implications between them. Then, it computes the transitive closure of the $\mathbf{M}_k(u_i)$ matrix identically to the LMIM algorithm. As a consequence, the resulting $\hat{\mathbf{M}}_k(u_i)$ matrix contains every implication chain of at most $2k+1$-step length or less. It may contain even longer implication chains up to the 2^k+1-step length, however, the minimum distance of the two farthest nodes of $G_{I,k}(u_i)$ is $2k+1$. If there is an implication chain between these nodes, the transitive closure is guaranteed to contain it. For each $u_i \in U$ unit the resulting transitively closed implication sets are unified with the global $\mathbf{M}^{(k)}$ matrix.

Now suppose, that there exists a $[f_i^x, \ldots, f_j^y]$ implication chain of at most $2k+1$-step length which is not included in $\mathbf{M}^{(k)}$. As $|\,[f_i^x, \ldots, f_j^y]\,| \le 2k$ there is a u_m unit for which

$$|\,[f_i^x, \ldots, f_m^z]\,| \le k \wedge |\,[f_m^z, \ldots, f_j^y]\,| \le k$$

Consequently, $u_i, u_j \in \mathrm{N}_k(u_m)$ and therefore the $\hat{\mathbf{M}}_k(u_i)$ matrix contains the hypothetical $[f_i^x, \ldots, f_j^y]$ implication chain. Now, the global $\mathbf{M}^{(k)}$ matrix would not include this implication chain only if $u_m \notin U$ which is a contradiction. □

Theorem 2.2.4. *The LTC algorithm, if it is run in an adequate number of iterations, generates the transitive closure of the $\boldsymbol{M}$ inference matrix.*

Proof. Similarly to Theorem 2.2.2 the proof trivially follows from Lemma 2.2.3 and the fact that the diagnosed system and therefore its inference graph is finite. □

Lemma 2.2.5. *The transitively propagated implication set computed by the DIL algorithm in the k-th iteration step does not contain implication chains consisting of more than $k+1$-step implications.*

Proof. (By contradiction.) Suppose, that there exists an $[f_i^x, \ldots, f_j^y]$ implication chain of more than $k+1$-step length which is included in $\mathbf{M}^{(k)}$. This can only happen if in each step of the computation the length of this implication chains grow by at least one implication, and at a certain step (let it be the l-th) the length grows by at least two implications. Due to the mechanism of the DIL algorithm this implies, that at one of the $u_p \in \mathrm{N}(u_i)$ neighbor units of u_i the length of computed implication chains has grown by at least two implications in the $(l-1)$-th iteration step. This further implies that at one of the $u_r \in \mathrm{N}(u_p)$ units the length computed implication chains has grown by at least two implications in the $(l-2)$-th iteration step, and so on. Consequently, there exists a $u_s \in \mathrm{N}_l(u_i)$ unit among the l-th neighbors of u_i at which the length computed implication chains has grown by at least two implications in the first step. However, the initial $\mathbf{M}$ inference matrix initialized to include only the one-step implications derived from the syndrome, therefore in the first step only one-step implications are propagated, which contradicts our primary assumption. □

Lemma 2.2.6. *The transitively propagated implication set computed by the DIL algorithm in the k-th iteration step contains every implication chain of at most $k+1$-step length or less.*

Proof. (By finite induction.) The $k=0$ case trivially holds due to the initialization of the $\mathbf{M}$ inference matrix. It is also easy to see that the Lemma holds in the $k=1$ case, because the one-step implications at every u_i unit are extended by the one-step implications of each $u_m \in \mathrm{N}(u_i)$ neighbor unit, thus there is no $f_i^x \to f_m^z \to f_j^y$ valid implication chain which is not generated.

Assume that the Lemma holds for a certain k, but it does not hold for $k+1$. This means that there exists a $[f_i^x, \ldots, f_j^y]$ implication chain which was not computed by the algorithm in the $k+1$-th step. The length of this implication chain must be exactly $k+2$, otherwise if $|\,[f_i^x, \ldots, f_j^y]\,| \leq k+1$ then the implication chain would have been computed in the k-th step. Cut the examined implication chain into two parts at any u_m unit in a way that $[f_m^z, \ldots, f_j^y]\,| = k+1$ is a $k+1$-length implication chain and $f_i^x \to f_m^z$ is a one step implication. It follows, that the $[f_i^x, \ldots, f_j^y]$ implication chain is not generated in the $k+1$-th step iff either the $[f_m^z, \ldots, f_j^y]$ implication chain was not computed in the k-th step (which contradicts the induction condition) or $u_m \notin \mathrm{N}(u_i)$ (which contradicts the definition of the inference graph). □

Remark. The DIL algorithm selects the one-step implications for transitive propagation in the k-th iteration step identically to a breath-first search algorithm.

Theorem 2.2.7. *The DIL algorithm, if it is run in an adequate number of iterations, generates the transitive closure of the $\boldsymbol{M}$ inference matrix.*

Proof. Similarly to Theorem 2.2.2 the proof trivially follows from Lemma 2.2.6 and the fact that the diagnosed system and therefore its inference graph is finite. □

Lemma 2.2.8. *The implication counters $\Sigma^0[i]$ and $\Sigma^1[i]$ computed by the CIP-2 algorithm in the k-th iteration step contain the exact number of one-step implications constituting implication chains of at most k-step length ending in the respective f_i^0 and f_i^1 fault hypotheses, provided that the inference graph of the system is circuit-free and the W weight function specifies uniform 1 weight for each implication rule.*

Proof. (By finite induction, analogously to the proof of Lemma 2.2.6.) The $k = 1$ case trivially holds, as the $\Sigma^{x,(0)}[i] - \Sigma^{y,(-1)}[i]$ increment value is 1 for each $u_i \in \Gamma{-}1(u_j)$ pair of units, due to the initialization of the $\Sigma^{x,(-1)}[i]$ and $\Sigma^{x,(0)}[i]$ counter values (where $x, y \in \{0, 1\}$). Therefore, the $k = 1$ step adds up the number of $f_i^x \to f_j^y$ one-step implications in the $\Sigma^{y,(1)}[j]$ counters corresponding to the u_j unit. It is also easy to see that the Lemma holds in the $k = 2$ case, because the increment value in the second step is $\Sigma^{x,(1)}[i]$, thus the number of local one-step implications is supplemented by the sum of one-step implications at the neighbors of the u_j unit.

Assume that the Lemma holds for a certain k, but it does not hold for $k + 1$. The value of $\Sigma^{z,(k+1)}[j]$ and $\Sigma^{z,(k)}[j]$ is computed by the CIP algorithm as

$$\Sigma^{z,(k+1)}[j] = \Sigma^{z,(k)}[j] + \sum_{p_{ij} \in P(A)} W(p_{ij}) \cdot \left(\Sigma^{x,(k)}[i] - \Sigma^{y,(k-1)}[i]\right) \tag{2.1}$$

$$\Sigma^{z,(k)}[j] = \Sigma^{z,(k-1)}[j] + \sum_{p_{ij} \in P(A)} W(p_{ij}) \cdot \left(\Sigma^{x,(k-1)}[i] - \Sigma^{y,(k-2)}[i]\right) \tag{2.2}$$

Substituting the expression of $\Sigma^{z,(k)}[j]$ into the expression of $\Sigma^{z,(k+1)}[j]$ and taking into consideration that $\forall p_{ij} \in P(A) : W(p_{ij}) = 1$ we get

$$\Sigma^{z,(k+1)}[j] = \Sigma^{z,(k)}[j] + \sum_{p_{ij} \in P(A)} \left(\Sigma^{x,(k)}[i] - \Sigma^{y,(k-2)}[i]\right)$$

(The reduction of summation addends is possible since the set of p_{ij} one-step implications is the same in both expressions and the addends also correspond to each other.) Now, we can further substitute the expression of $\Sigma^{z,(k-1)}[j]$ into the above equation, then subsequently substitute the expression of $\Sigma^{z,(k-2)}[j]$, and so on. After k substitutions we get

$$\Sigma^{z,(k+1)}[j] = \Sigma^{z,(1)}[j] + \sum_{p_{ij} \in P(A)} \left(\Sigma^{x,(k)}[i] - \Sigma^{y,(0)}[i]\right) = Sigma^{z,(1)}[j] + \sum_{p_{ij} \in P(A)} \Sigma^{x,(k)}[i]$$

Which can be interpreted as: "The number of implications in the at most $k + 1$-step length implication chains ending in the f_j^z fault hypotheses of unit u_j equals to the number of implications in the at most k-step length implication chains ending in the f_i^x and fault hypotheses of unit u_i plus the number of $f_i^x \to f_j^z$ one-step implications." The value of $\Sigma^{z,(k+1)}[j]$ can only be incorrect if either $\Sigma^{z,(1)}[j]$ is incorrect (but that was proven to be correct in the first step) or $\Sigma^{x,(k)}[i]$ is incorrect (which contradicts the induction condition). □

Theorem 2.2.9. *Let $\hat{\mathrm{M}}$ be a transitively closed inference matrix of a certain system. Surely classable units (based on the contradictions to be found in the closed matrix) are eventually correctly identified by the CIP-2 algorithm, regardless of the W weight function provided that the inference graph of the system is circuit-free.*

Proof. (By contradiction.) Assume that there exists a u_i unit surely classified on the basis of the transitive closure. This means that there is an implication chain from f_i^x to $f_i^{\neg x}$ (i.e., the fault state of u_i is surely $f_i^{\neg x}$). Let this implication chain be composed of l steps $|\,[f_i^x, \dots, f_i^{\neg x}]\,| = l > 0$. Suppose that in the k-th iteration step $\Sigma^{x,(k)}[i] = \Sigma^{x,(k+1)}[i] = m$ is the total number of the implications ending in f_i^x computed by the CIP-2 algorithm, but $\Sigma^{x,(k-1)}[i] < m$ (i.e., the correct value of the total implication number was generated in the k-th step). Using the proof of Lemma 2.2.8 we can deduce that

$$\Sigma^{\neq x,(k+l)}[i] \geq \Sigma^{x,(k)}[i] + |\,[f_i^x, \dots, f_i^{\neg x}]\,| = \Sigma^{x,(k+l)}[i] + l$$

It follows that running the CIP-2 algorithm in at least $(k+l)$-th steps $\Sigma^{\neq x,(k+l)}[i] > \Sigma^{x,(k+l)}[i]$ thus the fault state of u_i will be correctly identified as $f_i^{\neg x}$. □

Theorem 2.2.10 gives sufficient conditions for correct diagnosis with the developed LID probabilistic diagnostic algorithms. Although the conditions only have theoretical significance, they highlight the methodology and the problems of the analytical study of diagnosability in probabilistic methods utilizing general test invalidation.

Theorem 2.2.10. *The following (sufficient but not necessary) conditions guarantee that the all of three developed fault classification heuristics correctly identify the fault state of the u_i unit under diagnosis (UUD), based on a partially closed inference matrix describing a transitively propagated implication set of a certain system as generated by the DIL algorithm in its k-th iteration step:*

1. *$|U_{i,k}^0| \geq |U_{i,k}^1|$, where the $U_{i,k}^0$ and $U_{i,k}^1$ are the set of fault-free and faulty k-neighbor units (reachable from u_i via undirected paths of k or less length), that is $U_{i,k}^0 = \{u_x : u_x \in \mathrm{N}_k(u_i) \wedge u_x \in (U - F)\}$ and $U_{i,k}^1 = \{u_y : u_y \in \mathrm{N}_k(u_i) \wedge u_y \in F\}$,*

2. *$G_{i,k}^0$ digraph is strongly connected, where $G_{i,k}^0 = (V_{i,k}^0, E_{i,k}^0)$ is a directed graph composed of the $V_{i,k}^0$ vertices corresponding to the $u_x \in U_{i,k}^0$ fault-free k-neighbor nodes and the $E_{i,k}^0 = \{p_{xy} : u_x, u_y \in U_{i,k}^0\}$ implications between these nodes.*

Proof. Assume that the u_i unit under diagnosis is fault-free. According to condition (1) the u_i unit will be tested by at least one fault-free tester unit. Fault-free units always test each other with a 0 test result, thus there can exist only two types of implication chains between two fault-free units $u_p, u_s \in U_{i,k}^0$ in every test invalidation model: either $[f_p^0, f_r^0, \dots, f_s^0]$ or $[f_p^1, f_r^1, \dots, f_s^1]$. Due to Lemma 2.2.6 the implication set generated by the DIL algorithm in its k-th iteration step contains every $[f_j^y, \dots, f_i^x]$ implication chains, where $u_j \in \mathrm{N}_k(u_i)$. According to condition (2) $\forall u_j \in U_{i,k}^0 : u_j \in \mathrm{N}_k(u_i)$. As a consequence, the value of $\Sigma^0[i] \geq \nu_k/2$, because the f_j^0 fault-free state of each $u_j \in U_{i,k}^0$ implies f_i^0.

Since there is no $[f_p^0, \dots, f_i^1]$ implication chain between a fault-free unit $u_p \in U_{i,k}^0$ and u_i unit under diagnosis, the value of $\Sigma^1[i]$ comes from $[f_r^0, \dots, f_i^1]$ implication chains between faulty $u_r \in U^1_{i,k}$ units and the UUD. The number of faulty units is $|U^1_{i,k}| \leq \nu_k/2$ due to condition (1), therefore the $\Sigma^0[i] < \Sigma^1[i]$ condition of incorrect diagnosis can not be satisfied. This directly proves the impossibility of misdiagnosis of the u_i unit using the *Majority* heuristic.

In the case of the *Election* heuristic we prove that the u_i unit under diagnosis will never be selected by the election process. In the first step there exist at least one $u_j \in U^1_{i,k}$ faulty unit tested by at least one $u_p \in U^0_{i,k}$ fault-free unit $u_p \in \Gamma{-}1(u_j)$. In each test invalidation model the $f^0_p \rightarrow f^1_j$ implication (and possibly the $f^0_p \rightarrow f^1_j$ implication) exists between u_p and u_j, but in none of the test invalidation models may the $f^0_p \rightarrow f^0_j$ or the $f^1_p \rightarrow f^0_j$ implication exist. Due to to condition (2) $\forall u_r \in U^0_{i,k} : f^0_r \rightarrow f^0_p$, and consequently $\Sigma^1[j] \geq \nu_k/2$. The value of $\Sigma^0[j]$ may come only from $[f^0_s, \ldots, f^1_j]$ implication chains, where unit u_s is faulty. As a result, $\Sigma^0[j] \leq \nu_k/2$ which means $\Sigma^0[j] < \Sigma^1[j]$ and so u_j will have a smaller LF$[j]$ than u_i. Therefore, u_i cannot have a minimum LF$[i]$ value and will not be selected.

The previous result also implies that in the first election step only a faulty unit can be selected. Suppose it was not u_j. The elected unit is classified as faulty and the implications starting at it are removed from the inference matrix. Let us examine how does this affect the implication counters! For unit u_i the value of $\Sigma^0[i]$ does not change, while $\Sigma^1[j]$ may decrease. For unit u_j the value of $\Sigma^0[i]$ may decrease, while $\Sigma^1[j]$ does not change. Consequently, LF$^{(2)}[i] \geq$ LF$^{(1)}[i]$ and LF$^{(2)}[j] \leq$ LF$^{(1)}[j]$, that is u_i cannot be selected in the second election step. Continuing this reasoning it follows that u_i will never have a minimum LF$[i]$ value during the election process.

For the *Clique* heuristic the proof procedure shows that the clique containing the diagnosed unit is always correctly selected as the Fault-Free Core. According to condition (1) the u_i unit is tested by at least one fault-free tester unit. Fault-free units are always connected by $[f^0_p, f^0_r, \ldots, f^0_s]$ or $[f^1_p, f^1_r, \ldots, f^1_s]$ implication chains. According to condition (2) $\forall u_j \in U^0_{i,k} : u_j \in \mathrm{N}_k(u_i)$ and so there is an $[f^0_p, f^0_s]$ implication chain between any two fault-free units $u_p, u_s \in U^0_{i,k}$ in every test invalidation model. Implications of type $f^0_i \rightarrow f^0_j$ or $f^0_j \rightarrow f^0_i$ cannot exist if $u_j \in U^1_{i,k}$, because then either the u_i or the u_j would be surely classified as faulty. Therefore, the cliques formed at the $u_p \in U^0_{i,k}$ units are identical, contain all fault-free units (and only them) including unit u_i. Due to condition (1) this clique is the largest, since the $u_j \in U^1_{i,k}$ faulty units can create $[f^0_j, f^0_m]$ implications chains only if $u_m \in U^1_{i,k}$ is also faulty. Since $|U^0_{i,k}| \geq |U^1_{i,k}|$ the clique generated by fault-free units will be correctly accepted as the Fault-Free Core.

The other part of the proof, showing that the u_i unit under diagnosis cannot be incorrectly identified as fault-free when it is faulty, can be deduced completely analogously. We omit this part of the proof for the sake of conciseness. □

It is easy to illustrate that Theorem 2.2.10 gives only sufficient conditions. Imagine a system composed of units with homogeneous asymmetric invalidation with all of the units but one being faulty. Assume that among the faulty tester units of the fault-free processor there is at least one which obtains a passed test result. Due to the properties of the asymmetric invalidation every unit which is tested successfully can be surely classified as fault-free, therefore the correct identification of the fault-free unit in our example was possible even if the conditions formulated by Theorem 2.2.10 obviously do not hold.

This example also highlights the problems of the mathematical analysis of the developed probabilistic diagnostic algorithms. Necessary and sufficient conditions can only be derived separately for each of the nine possible invalidation models. Then, the separate conditions must be unified to create some worst-case or typical diagnosability formulas for heterogeneous systems. Theorem 2.2.10 also indicates that beside developing these formulas one must also have a notion of the fault pattern structure. Such structural properties of the patterns resulting at certain fault percentages and fault injection mechanisms can be studied by a special branch of mathematical statistics called *percolation theory* [83].

2.3 Measurement conclusions

As suggested in Section 2.2, the analysis of probabilistic diagnostic algorithms is not straightforward. While the analytical study of the algorithm properties is feasible for homogeneous systems with uniform test invalidation model in each unit, the evaluation of heterogeneous testing situations is much more complex. Theoretical analysis of this kind exceeds the scope of this document. However, providing information about the performance and applicability of the developed LID algorithms and comparing them to existing methods known in the literature is imperative. The other option for obtaining the necessary information is *simulation* and *measurement* [4].

The experiments on the developed algorithms were performed in a dedicated simulation environment, developed by the author for the purpose of evaluating centralized fault diagnostic algorithms. Planning and execution of the measurement activities are described in detail in Appendix C. The appendix explains the selection of measured parameters and examined parameter interactions, and presents the collected measurement results. This section summarizes the conclusions of the experiences gained during the measurement activities listed in Table 2.2.

The measurements confirmed the analytical properties presented in Section 2.2, but also provided valuable observations the theoretical values do not reveal. For example, the LTC algorithm has a better time complexity respective to the number of units than the DIL algorithm, but in practical size systems the latter is much faster than the former. (Extrapolation shows that in a machine having 128×128 processors the execution time of the DIL algorithm is still one-third of the LTC algorithm.) In general, the developed LID algorithms proved to provide better diagnostic accuracy than the probabilistic methods taken from the literature. The difference in the diagnostic accuracy was even more obvious in the non-symmetric and heterogeneous test invalidation situations, where the improvement could be even in the one order of magnitude range. These results were achieved by running the LID algorithms in only one iterations, and using general fault classification heuristics without any optimization for particular test invalidation models.

The representation of the diagnostic information turned out to be determinative from the accuracy viewpoint. Scalar methods gave better result than the existing methods, since they are based on the generalized test invalidation model so they can detect and utilize

Table 2.2: Measurement activities

Measured characteristic	**Examined interaction**	**Varying parameters**		
		System	**Fault pattern**	**Algorithm**
Diagnostic accuracy	System size	System size		
	Fault set size		Fault set size	
	Fault arrangement		Group fault set	
			Custom fault set	
	Topology	Topology		
		Connectivity		
	Invalidation	Invalidation		
	Type of misdiagnosis			Heuristic
	Arrangement of misdiagnosed units		Custom fault set	
	Diagnostic information			Iteration
Algorithm complexity	System size	System size		
	Fault number		Fault set size	
	Topology	Topology		
		Connectivity		
	Diagnostic information			Iteration

one-step sure implications, and are capable of iteratively extend their diagnostic knowledge by considering the diagnostic inferences by non-adjacent units. Furthermore, inference propagation based methods achieved still better results than scalar methods, because they represent the complete set of implications so they can detect and utilize even multiple-step sure implications, and count multiple-step implications rather than edges in implication chains, therefore they can estimate more precisely the likelihood of a fault hypothesis. The more complex the representation is, the more time-consuming the given method will be, this rule was confirmed by the measurements. However, scalar algorithms have comparably low delay as the existing probabilistic methods, while most of the inference propagation based approaches are time efficient enough to be used in large computing systems.

The experiments also proved that the practical scattered and group fault situations can be correctly diagnosed with high probability even if only a partial diagnostic information is used. Diagnostic accuracy significantly improves in the first few inference propagation iterations, but quickly approximates to a constant value. The LID algorithms have a linear time complexity with respect to the number of iterations (except the LTC method), therefore the extension of the diagnostic knowledge does not significantly increase the duration of the diagnostic process.

During measurements we took extra care to ensure the correct implementation of the examined algorithms. Therefore, the inference extraction and propagation methods were implemented in multiple conceptually distinct instances, and the operation of the instances was compared to see if there are any deviations. Also, the implication sets generated by the LMIM, DIL, and LTC methods was executed in an appropriate number of iterations and then compared. The statistical functions of the simulation environment were tested by exporting the computed statistical data into a commercial statistical package and checking its correctness. And, of course traditional debugging techniques were also employed to verify the correct operation of the created program code.

2.4 Application of local information diagnosis

The traditional application environment of centralized probabilistic diagnostic algorithms are the *(massively) parallel* computing systems having a *regular interconnection topology.* The local information diagnosis approach and the corresponding probabilistic methods presented in this document are quite flexible regarding the testing assignment: they can employed in any arbitrary topology. The LMIM algorithm is advantageous in testing graphs having a large degree of connectivity, since it always tries to propagate implications in every possible pair-wise combinations of units. The effectiveness of the DIL and CIP methods increases in constant, low-degree regular interconnection topologies, yet they also are capable of operating in any system topology. The LTC algorithm is the most sensitive to the system topology as its time complexity is $O(\nu^3)$ with regard to the connectivity of the testing graph.

Although the probability of fault occurrence in massively parallel computers is significant, it is reasonable to assume that the percentage of faulty units even in a large parallel computer does not exceed 10 percent. The LID methodology provides very high diagnostic accuracy in the practically relevant 0–10 fault percent range: less than 0.1 percent of the system units (that is less than 1 percent of the faulty units) is misdiagnosed. Therefore, in this application range the LID algorithms can also be used as stand-alone diagnostic methods. This is especially true for the Clique fault classification heuristic, which does not make diagnostic mistakes at all in this range.

For larger fault probabilities the percentage of misdiagnosed units may become unacceptable. In this case two main application areas offer themselves for the LID diagnostic algorithms. The first option is the utilization of the developed methods in a *sequential diagnostic strategy.* In the sequential approach fault classification is followed by maintenance actions: repair or replacement. Although probabilistic algorithms may produce diagnostic mistakes, the majority of located faulty units will be really physically faulty. The thesis demonstrates, that local information diagnosis correctly identifies isolated faults and fault group borders. Due to the maintenance procedure the found faulty units are repaired/replaced, thus new tester processors become available again. Since some of the faults could have remained hidden, maintenance is followed by a new testing and diagnostic session. The diagnostic and maintenance cycle is repeated until there are no more faulty units revealed by the executed tests. During this iterative process the LID diagnostic technique reveals scattered individual faults in the first step, while group fault sets are uncovered gradually, advancing from the border towards the core of the group. It is known from the literature, that sequential diagnosis guarantees to bring the system into a fault-free state in finite number of cycles, provided the fault rate is low [55].

As a second option, the developed probabilistic algorithms can be employed to fulfill a "filtering" function. They can be used to generate a quick approximate diagnostic image, which designates those system areas that have been affected by the occurred faults. This primary image can be further refined by more sophisticated (and more time-consuming) deterministic fault classification methods. The secondary deterministic algorithms do not need to process the whole syndrome: they can only concentrate on the problematic areas revealed by the primary analysis. Such a "hybrid" approach can guarantee the highest achievable diagnostic accuracy while retaining its high execution speed.

An emerging new application area for system-level fault diagnosis is *wafer-scale testing* during the manufacturing of VLSI electronic circuits. Multiple such integrated circuits are built on a single silicon slice, which is later cut up into individual devices packaged separately. Unfortunately, even in the current advanced manufacturing technology the yield of the fabrication process is rather poor, many of the produced ICs are defective. The verification of the manufacturing process is realized by a complicated external testing equipment. The tester has special probes which connect to the I/O pins of the tested device. It is positioned over each tested circuit, then drives the inputs by dedicated test patterns and samples the signals on he outputs. When it finds a deviation in the test responses from the stored reference responses, the tested circuit is declared to be faulty and is thrown away.

There are several shortcomings of using an external testing equipment. The in-circuit tester device is very complex and expensive, therefore only one exists for each production line. Consequently, the verification process is *sequential.* Furthermore, moving the testing equipment into its exact position over the tested circuit requires high accuracy. This accuracy cannot be achieved at high speeds, thus the positioning process is *slow.* Moreover, the in-circuit test cannot be performed at full operational speed of the tested device, as the testing equipment is not able to drive and sample the I/O pins fast enough.

Most of the modern VLSI devices are programmable, and they are arranged on the silicon wafer in a regular pattern. This gives an opportunity to adapt the system-level diagnosis theory to wafer-scale testing. Rather than probing the ICs individually by means of a test machine, the ICs themselves can perform mutual tests, just as in parallel computers. To perform the tests, ICs can be assigned the same job (in principle this implies that they receive a common data and instruction input sequence), and the output sequences of adjacent circuits can be compared. The comparison outputs are collected by dedicated registers and are obtained by a central computer performing the analysis of the test results. The centralized fault diagnostic algorithm running in the central computer identifies the location of faulty ICs. The technique requires the integration of additional redundant electronic components (comparators, registers, interconnection lines, etc.) on the wafer, which can be done easily and at no significant added cost or time. The redundant components are later simply cut off and disposed of.

Compared to the traditional test strategy the new methodology offers several advantages. The testing of all ICs is carried out in parallel, thus greatly accelerating the verification process. Furthermore, the test programs can be executed at full operational speed of the checked circuits. This further reduces verification time, but more importantly, high-speed inspection may uncover design and manufacturing faults that do not expose themselves at lower clock frequencies. Also the costs are significantly lowered, since the expensive testing equipment can be replaced by a commonplace industrial computer.

Chapter 3

Distributed fault diagnosis

The previous chapter presented centralized diagnostic algorithms designed for large computing environments. However, a large part of massively parallel systems are fully distributed in order to maintain scalability, and does not fulfill the conditions of centralized diagnosis. For these machines an alternative approach: *distributed fault diagnosis* was created by Kuhl and Reddy [31]. This chapter presents the Parsytec GCel distributed parallel computer, and an event-driven distributed diagnostic algorithm developed specially for this machine. The implementation details (software structure, communication protocols) are also explained. Finally, we consider the problem of adapting the developed LID algorithms to the distributed application environment.

3.1 Illustrative system: the Parsytec GCel parallel computer

The Parsytec GCel is a massively parallel, distributed multiprocessor system. It was produced by the Parsytec Computer GmbH, located in Germany. The machine was a predecessor to the GC series of Parsytec multiprocessors. The GC parallel computers were designed to use the long awaited T9000 transputer from INMOS. When the T9000 project has been canceled, the company switched over to the PowerPC 601 processor of IBM, and created the Parsytec Power Xplorer series. GCel and Power Xplorer machines are used world-wide in many universities and scientific institutions. Their application includes many topics ranging from image processing to fluid dynamics.

The Parsytec GCel system is based on MIMD parallelism. The processing elements of the system are functionally identical, but they operate independently, have a dedicated local memory and execute separate tasks. Two main functional groups can be distinguished in the architecture. The first functional group is the network of *processing elements* running the user applications. This network is fully distributed to fulfill the requirements of massively parallel computing. It has a vertically organized three-level system hierarchy, in order to make the system complexity manageable. On the lowest level there are the processing elements, composed of a transputer, the local memory and some additional logic implemented in a special system ASIC. The next hierarchy level is formed by *clusters*, incorporating several application processors and routing chips. *GigaCubes* constitute the highest hierarchy level, composed of four clusters with additional communication facilities, power supply, and cooling devices.

The second functional group is the *front-end computer* providing both a user interface to the multiprocessor network, and access to external peripherals (like the display, storage medium, etc.) for the processing elements. The front-end or *host computer* is essentially an intelligent terminal facilitating management, control, and monitoring of the Parsytec GCel system. The two-way connection between the host and the multiprocessor network is handled by an interface program running on the host called the *server*. The server program is liable for booting the multiprocessor by downloading the user program to the PEs, collecting the results of monitoring tasks, and conduct the PE access to the global peripheral devices.

The processing elements employed in the Parsytec GCel massively parallel system are INMOS T805 transputers. Transputers are specifically designed for distributed multiprocessing applications. They include every functional unit necessary for independent operation, for this reason they are also called "single-chip microcomputers." Transputers integrate a 32-bit integer processor (CPU), a 64-bit floating point coprocessor, a *programmable memory interface* (PMI), 4 KBytes of on-chip cache memory, and a communication subsystem with four high-bandwidth serial communication links. Although the use of transputers in massively parallel systems has by now practically diminished, they are still quite popular components of *embedded systems*.

The integer CPU has a simplified instruction set, designed for efficient execution of compiled assembly code generated by high-level language compilers. The CPU is able to execute one instruction in each clock-cycle, supported by a three-stage pipeline architecture. It can be switched into a *protected mode*, enabling memory access protection and automatic address translation to prevent the execution of privileged instructions. A hardware kernel for scheduling processes and performing communication is also included, offering very fast interrupt response and context switch times. These operations are directly supported by the instruction set. A 4 KByte unified *on-chip cache* helps to achieve single cycle instruction execution and data access. The cache memory provides fast, multi-port access to program data, reducing the number of external memory operations. The built-in interface to the external memory is highly programmable, allowing the construction of flexible configurations of various memory devices using little or no additional logic. The PMI device handles DRAM memories including multiplexed row/column addresses, refresh and page mode addresses. Furthermore, other devices such as SRAM, ROM or memory mapped peripheral controllers are also supported.

The transputer architecture has always been distinguished by its communication features. A complete *communication subsystem* is integrated on the chip to perform high transfer rate inter-processor data exchange of more than 2 Mbytes/s. Due to the low-level hardware support both internal and external data transfers are handled identically, using the same set of machine instructions. The transputer includes four full-duplex serial communication links, each equipped with a pair of local direct memory access (DMA) channels. A communication coprocessor supervises all link activities. This coprocessor operates concurrently with the CPU so data transfers do not have any effect on the execution speed of the applications.

Much of the functionality of a Parsytec GCel processing node was implemented in a

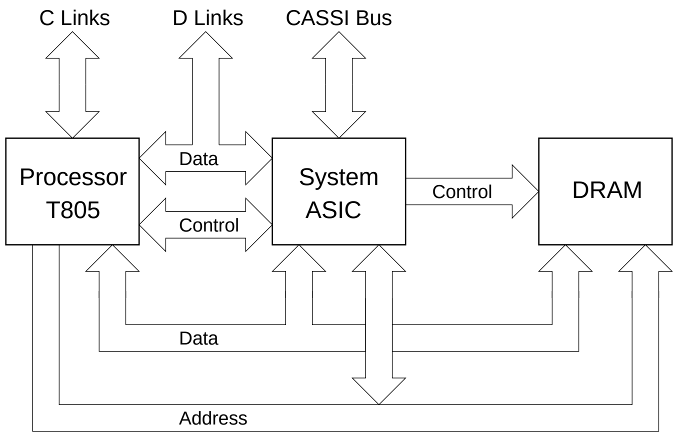


Figure 3.1: The structure of a Parsytec GC processing node

special system *Application-Specific Integrated Circuit* (ASIC). This device supplements the features of the transputer with error detection and correction for dynamic random-access memory (DRAM) accesses and includes many other features to make the processing elements efficient and reliable. The size of the local DRAM memory can be either 4 or 8 MBytes per node, the memory interface is 32 bits wide. Memory contents are protected from soft transient errors by an error detection and correction (EDAC) functional unit. For optimization of the inter-processor communication network and the cache usage of parallel applications, each processing node contains link activity and external memory usage monitoring hardware. The link monitor counts the number of data bytes transmitted over each link and the maximum possible byte transfer rate. These two values can be acquired by the user programs. The memory monitor has an equivalent function for DRAM accesses. Each Parsytec GCel processing node is equipped with various test functions to verify their correct operation. These test functions make possible the checking of system units with a special test software to detect hardware failures.

The next hierarchy level, and the smallest autonomously operating element of the Parsytec GCel system is a *cluster*. A GCel cluster incorporates 17 application processing nodes and 4 INMOS C004 routing chips. Only 16 of the PEs are active, the remaining one is a reserve processor. The active processors run the operating system kernel and the applications. The redundant processor is needed for fail-safe operation: in case of a hardware failure it is invoked to replace the defective node. Each cluster can be seen as a high-performance fault-tolerant computing node, the probability of a failure occurrence in a cluster is less than in a single transputer [84]. The interconnection topology within a cluster is a fully connected graph. The transputers are connected via crossbar routing devices (illustrated in Figure 3.2).

The physical construction of a cluster is distributed into three printed circuit boards. Eight processors are placed on a so called *processor board*, and two such boards are plugged into a common backplane, housing all connections between the two processor boards. The spare PE is placed on a redundant module on the control board, which contains the standby processors from all the four clusters of a GigaCube.

GigaCubes constitute the highest hierarchy level. They are the basic building blocks

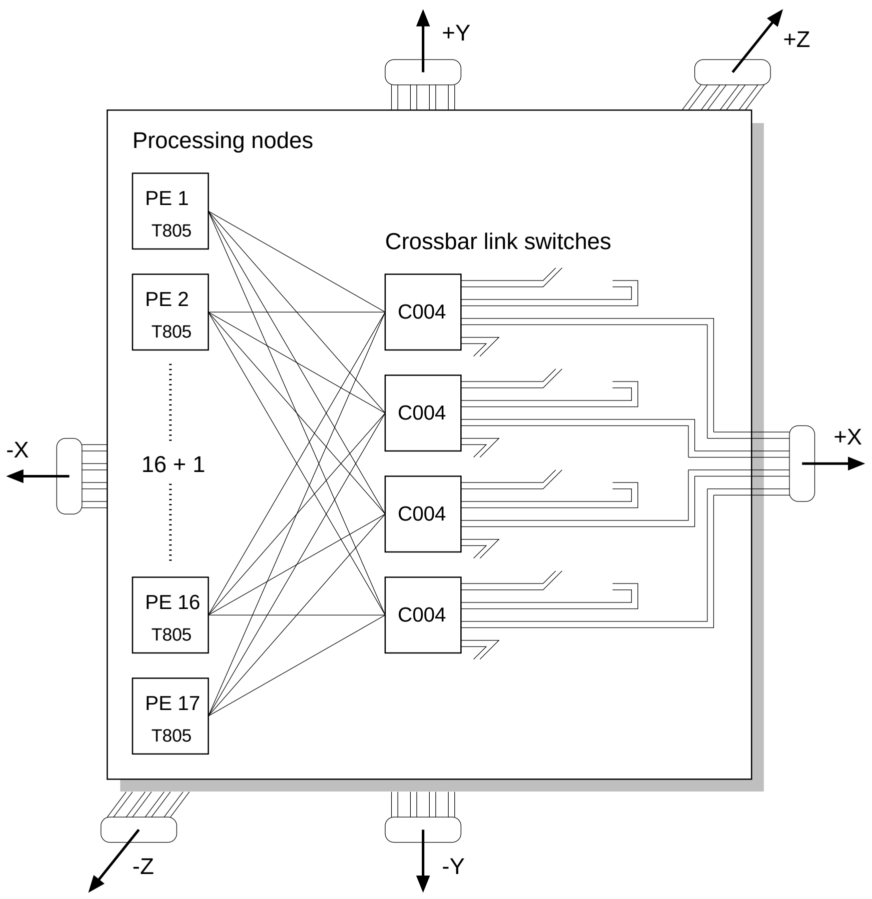


Figure 3.2: A Parsytec GC cluster

of the Parsytec GC series (even partly equipped GigaCubes, containing only one or two clusters, are available). GigaCubes are composed of four clusters with additional communication facilities, power supply, and cooling devices. They contain $4 \times 4 \times 4 = 64$ application processors logically organized in a three-dimensional mesh interconnection topology. The four clusters are monitored by a common *control processor*. The purpose of the control processor is to download the user programs to the transputers, to supervise the operation of the application processors and to interact in case of a processor failure, to monitor the power supply and the cooling system, and to perform administrative tasks. The I/O network is also driven by the control processor. Several GigaCubes can be connected in all the three spatial dimensions to produce a larger computing engine. The interconnection network among the GigaCubes forms a three-dimensional grid structure and consists of 8 data and one control channels in all the six spatial directions. This unified structure makes the connection of further GigaCubes into the system possible with minimal installation. A maximum of $8 \times 8 \times 4 = 256$ GigaCubes form the full configuration of the machine.
The Parsytec GCel system incorporates three different communication networks:

Data network (D-Network). The data network connects the processing elements to transfer information between the cooperating tasks of a task force. The overall structure of the data network is a three-dimensional grid, although its physical realization is different in the lower levels of hierarchy. Using this communication medium the

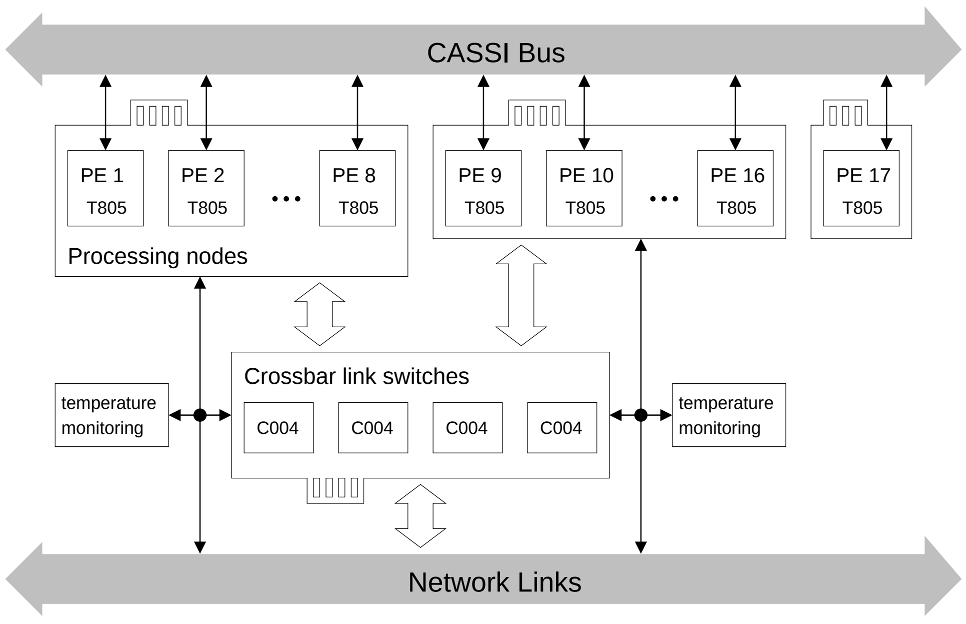


Figure 3.3: The hardware layout a cluster

GCel system is capable to create *virtual links* between any two running processes. Virtual links make the message transfer user-transparent: it possible to form any kind of logical communication topology among processes without knowing their physical disposition to processors. The data network has a fault-tolerant design. Failures in connectors, lines, and routing chips do not affect the ability of communication, only cause a reduced transfer bandwidth.

Input/Output network (I/O-Network): The input/output network provides the processing elements with access to global devices, like the host computer and mass storage.

Control network (C-Network): The control network connects the control processors, carrying diagnostic and administrative information among them. This network also serves as the downloading channel of the user applications to the processing elements. The control network has a central supervising instance: an operator terminal with access to this network. The software running on control processors ensures that every configuration and monitoring procedure can be initiated from the terminal. Since the control network is parallel to the I/O links, the whole I/O subsystem is accessible over the C-Network as well as from the processing nodes.

The interconnection structure of all three networks of the system is a three-dimensional mesh grid in the Parsytec GC family of computers. However, the Parsytec GCel is a special low-cost member of the GC family, and its communication networks have a two-dimensional mesh grid topology.

The software environment for the Parsytec GCel series is called *PARIX*. PARIX is a UNIX-based operating system with parallel extensions necessary for the multiprocessor hardware. These extensions can be found on both the front-end system and the processing elements of the multiprocessor network. As a distributed operating environment PARIX

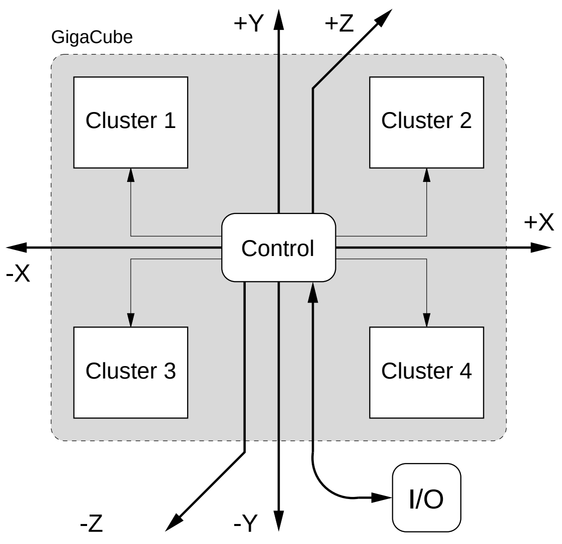


Figure 3.4: The logical organization of a GigaCube

combines software for system administration and configuration, application development, execution and debugging. The four main components of PARIX and their purpose are the following:

1. the User Development Environment incorporates all the services and tools of an advanced operating system, like program development tools, file server access, network integration, variety of peripherals, and code management system.

2. the User Runtime Environment is a part of the operating system running on the processing elements. The runtime environment includes a message passing kernel for inter-process communication, numerical libraries and parallel programming models.

3. the Multiuser Administration uses the front-end UNIX workstation to connect the parallel machine with the user. It provides interactive operation, creating user partitions in a multi-user environment, and operator access.

4. the Control Network Software runs on the control nodes of the clusters to ensure reliable operation. This software is also responsible for the creation of user partitions, downloading of applications, and control activities.

In the Parsytec GCel system an identical user program is downloaded to all the transputers. The transputers can identify their spatial position in the network based on a global data structure stored on each node (initiated during the boot process) and use conditional statements to execute different tasks depending on the position. The applications can obtain information about the physical and logical position of the processor they are running on. Three type of information is available: the physical location of the processor in spatial coordinates, the serial number of the processor (assigned to PEs in a sequential order), and the dimensions of the network. This information can be used to enable different parts of the identical `main()` program and thus creating different tasks on the various nodes.

Parsytec GC machines have a three-dimensional physical structure of the interconnection network. However, any kind of communication topology can be created using the so-called *virtual links*, in case of applications demand other arrangements for cooperation. Virtual links can connect any two arbitrary processes independent on the physical links on the software level. They are bidirectional, synchronized (non-buffered), dedicated point-to-point communication channels. Virtual links hide the difference of the local and remote communication providing a unified interface between processes running on the same and on remote transputers. Predefined virtual topology libraries are also part of the PARIX system offering calls for emulating grid, torus, pipe, tree and hypercube structures. In addition to the predefined virtual topology libraries the programmer can create his own virtual topologies in PARIX. Any number of virtual topologies can coexist on the same processor without interfering with each other.

The set of available communication methods has significant importance in multiprocessor systems, since message transfers are the basic forms of cooperation and mediation among processing elements. Three classic communication models exist in the PARIX environment:

Synchronous link-bound communication. This is the fastest connection method, directly supported by the transputer hardware. Prior to the data transfer a virtual link has to be created between the processes wanting to exchange information. The cooperating processes are synchronized upon the transmission. The process initiating the communication is blocked until the other process also becomes ready for the communication (receive/send). When the message transfer took place both processes continue the normal execution simultaneously. Another possibility is to send messages by *random-routing*, when the information is transmitted directly to the selected transputer without the need for a virtual link.

Asynchronous link-bound communication. This communication method allows to initiate data transfer in the background without having to wait for a handshake. Similarly to the synchronous link-bound communication a virtual link is necessary for the connection. After calling a data transfer function a separate communication thread is created, and the initiating process can continue normal program execution. The thread accomplishes the transmission and stores the result. Later the initiating process can check the stored result by a system call. Asynchronous link-bound communication is implemented using the synchronous message transfer functions, intermediate buffering is supported at both the sender and receiver side.

Asynchronous mailbox communication. This is the most general form of data exchange. Neither process handshake nor link connection is required. The sending process can immediately continue the execution after initiating a transmission, and the receiving process can asynchronously query its local mailbox for pending messages or wait for the arrival of a new message. The communication statements have a time-out feature (in the PARIX environment only the receive functions are equipped with time-out). When the time-out limit was reached without the recipient process accepting the message, the calling process is notified about the unsuccessful transfer.

Deadlock is avoided in the asynchronous communication model, since processes are not blocked upon the message transmission.

External services like access to mass storage devices or networking facilities are realized by Remote Procedure Calls. A process may execute an RPC by generating a message and transmitting it to the external server program running on the host computer. Upon receiving the request the server program performs the necessary actions, then sends back the result to the calling process.

3.2 An event-driven distributed diagnostic algorithm

The author developed a distributed system-level diagnostic algorithm [13] and applied the enhanced version of the algorithm to the Parsytec GCel massively parallel system [12, 11]. The application of the diagnostic algorithm requires the following conditions to be fulfilled:

Symmetric test invalidation. The algorithm treats the processing elements as "intelligent units" performing tests on other units directly accessible via a communication link. No assumption on the behavior of faulty processor is made, a test performed by a faulty tester may result in an arbitrary outcome, thus the symmetric test invalidation model is employed.

Link diagnosis. The communication subsystem including links but excluding the interconnection lines are part of the processor. The failure of the CPU makes the complete device faulty regardless of the fault state of the communication subsystem, since a faulty PE is unable to appropriately process or forward the incoming message even if it was correctly received. In other words, the failure of the CPU or all the four communication links are identical in diagnostic respect. The reverse condition does not hold, i.e., the CPU may remain operation regardless of individual link or interconnection line failures. The breakdown of less than four communication links is handled by the algorithm. There are no link-specific tests, thus the fault of an interconnection line cannot be distinguished from the failure of the two adjacent communication links of the connected processors.

Diagnosability. The majority of the distributed diagnostic algorithms also employs the diagnostic t-limit as its one-step diagnosability measure and fault classification mechanism. Recall, that the t-limit is the largest number of faults in any arbitrary fault set for which a proper diagnosis is always assured. The t-limit is a worst-case diagnostic measure (e.g., for the two-dimensional mesh it is as low as two faults). The reason for its use is to guarantee that the generated diagnostic image will be identical in every *fault-free* processor. This expectation coincides with the *completeness* property of centralized diagnosis. However, as we will presently see, its practical significance is arguable, and in most situations [73] it provides an over-pessimistic estimation [85].

Determining the message order. The arrival of diagnostic messages at a processor does not always correspond to the order of their sending due to communication delays. The change of the physical message order can lead to undesirable consequences, even to an incorrect diagnostic result. Such a situation can occur for example in an event-driven diagnostic algorithm when a PE becomes faulty during the testing process. To avoid this, the logical order of messages must be maintained even if their physical order becomes mixed. Therefore, time-stamps related to the time of test execution must be attached to the diagnostic messages, and the correct message order must be determined using a distributed event-ordering procedure like [86].

Tests are independently performed by each processor on its direct neighbors. Only normal interconnection links can be used for testing purposes. In addition to the error detection mechanisms of the link hardware, the authenticity and integrity of every message is assumed to be protected by algorithmic methods such as a checksum or digital signature. This authentication method serves as an additional test for both the sender processor and the communication link connecting the recipient PE with the sender. Each message must be acknowledged, and acknowledgement messages have the highest priority. Acknowledgements can be either *positive* or *negative*. The positive acknowledgement simply notifies the sender that the message was accepted. The negative acknowledgement requests the sender to repeat the last message. A sender process may transmit only a finite amount of messages before they would become acknowledged. If multiple messages were received from the same sender unit, then they are acknowledged in a reverse, LIFO order, starting with the most recent message. There is also a time-out for the acknowledgement. When the acknowledgement time-out expires, the corresponding message is automatically repeated. Messages are identified by their *sender sequence number* (SSN) to eliminate duplicate messages.

There is no explicit test request message, < I'm alive > or *heartbeat* messages are sent periodically by all processors to all neighbors within a predefined time-out limit. The < I'm alive > message based verification is a time-domain testing scheme, therefore it must be combined with a functional testing mechanism. For this purpose heartbeat messages carry two pieces of testing data: functional test inputs and test results. Each processor contains identical algorithmic testing procedures. They process the input data received from a given neighbor in an < I'm alive > message, process it with the built-in testing procedure and return the result in a subsequent < I'm alive > message sent to the respective neighbor processor. On receiving a test result each processor compares the received result with (saved or self-generated) local reference values. This combination of test result reporting and heartbeat messages prevents the algorithm from the occurrence of a deadlock due to lost messages and test requests [87], and reduces the number of required tests [88].

The developed diagnostic algorithm implements the *distributed fault-tolerance* concept of Kuhl and Reddy [31]. That is, each fault-free processor generates a diagnostic image of the whole system independently. The resulting diagnostic images are guaranteed to be identical for all cooperating fault-free processors. The units indicated as faulty in this diagnostic image are logically disconnected from the system. Fault-free processors achieve the isolation of the erroneous devices by refusing any communication or cooperation with them.

Data exchange takes place via a set of direct point-to-point links, therefore messages between non-neighboring processors must travel a path among processors and interconnection lines. Faulty processors or communication links cannot be included in this path, because they would possibly invalidate the flow of information. Therefore, a set of faulty processors and links may divide the cooperating processors into isolated groups and cut the interconnection graph into disjoint subgraphs. In this case the test results on the isolated units cannot be obtained and the diagnostic image generated by the fault-free processors cannot include every unit in the system, i.e., cannot be *complete.* Traditional distributed fault classification methods incorporate the diagnostic t-limit or other requirement on the set of possible fault pattern to still achieve a *complete* diagnosis. Due to the employed restrictions they may safely assume that the isolated are fault-free.

However, these restrictions are not necessary, because the knowledge of the fault state of units in the unreachable parts is not useful. Cooperation among processors participating in a distributed computation, and system-level testing takes place via the communication facilities. Therefore, only the *reachable* units can be considered as available resources. Consider the $G' \subseteq G_C$ graph composed of the $U-F$ fault-free units and the interconnection lines adjacent to fault-free units. Diagnosis of the whole system by every fault-free processor is possible until G' remains strongly connected. If a set of faulty processors and links cut the interconnection graph into disjoint subgraphs, diagnosis is restricted to each group of fault-free processors located in the same connected subgraph. Since subgraphs are isolated, they do not have a common knowledge with the other subgraphs. The host computer usually connects to the distributed multiprocessor system via a host link (as in the Parsytec GCel) to a dedicated PE. In the case the host computer (and therefore the user) can access only the PEs located in the connected subgraph containing the host link.

Following the above considerations, the developed distributed diagnostic algorithm does not require the limitation of the number of faults. Rather, it includes in the generated diagnostic image only the units located in the same connected subgraph of the system. Unreachable processors are identified by *detecting the isolating barriers* made of faulty processors. This barrier detection procedure must be executed each time a new unit failure is detected to verify whether additional processors became unreachable. The fault state of unreachable processors is classified as unknown. Another possible diagnostic event is a *unit repair.* Repairing units not only must be restored in a consistent state, but they also must be re-integrated in the distributed computation. Furthermore, when a faulty unit, formerly being part of a barrier dividing two strongly connected regions is repaired or replaced, it may open a "gate" through which the two isolated regions will join into one. In such case the processors of the two previously separate parts maintain two different local diagnostic images. These two images must be united in order to obtain a consistent diagnostic view in each fault-free processor of the joined parts.

3.2.1 Structure of the distributed algorithm

The diagnostic process described below is identical for the different processors. The algorithm consists of two stages: an *initial* and a *working* stage. Two observations motivated the splitting of the algorithm:

Different failure rates. It is known, that the majority of faults occurs (or already exists) during system switch-on/switch-off. This can be explained by the fact, that in the initial boot-up process (as well during the shut-down process) certain physical parameters (e.g., the current drain) may exceed the nominal range for short transients, other parameters (e.g., the device temperature) may require longer time to reach their nominal value. Therefore, computing systems typically use *power-on self-tests* (POST) for detection of initial permanent faults. At the end of the booting process the physical parameters of the system stabilize in their nominal range and the failure rate is expected to be lower during further operation.

Lack of diagnostic knowledge. Processors do not have any information on the fault state of other components in the initial stage of the diagnosis algorithm. All processors have to be tested once to generate the initial diagnosis image. Later the system fault state changes only gradually compared to the initial state due to lower fault rate. For this reason, a considerable overhead in communication and administration can be saved by calculating and distributing only the differences between the actual (changed) fault state and the stored (previous) diagnostic image. This means that diagnostic procedures are invoked only in the event of an error occurrence or repair/replacement, i.e., the diagnostic algorithm is *event-driven.*

Initial stage

The aim of the initial diagnostic stage is to create a *primary global* diagnostic image. For this purpose, on the startup every processor enters a *testing phase.* In the testing phase they conduct tests on all of their adjacent neighbor processors using the previously mentioned combination of dedicated functional and time-out tests. The so obtained local test results are collected in a *local test vector.* When all tests have been carried out an *inter-processor communication phase* starts. This communication phase implements a complete multicast message transfer: each (fault-free) processor put its local test vector in a message and forward this message to all of the processors in the system. Consequently, by the end of the communication phase every (fault-free) processor gathers the local test vectors of all the other (fault-free) processors, i.e., it obtains the complete syndrome. The details of the communication protocol used in the initial stage are explained in Appendix D.

The syndrome is analyzed by a simple algorithm, shown in Algorithm 9. The fault classification starts with the u_i unit performing the distributed diagnosis. The unit initializes its own fault state to be fault-free. Subsequent steps of the algorithm implement a breath-first search of the testing graph starting from the u_i unit. In the first step test results on links adjacent to u_i are examined. Passed and failed t_{ij} test result between unit u_i and its neighbor u_j indicate the fault-free or faulty state of the corresponding communication link connecting u_i and u_j. The search continues by following the fault-free links adjacent to u_i, faulty links are ignored. The search terminates when every path made of only fault-free links starting from the u_i unit have been traversed. When the search algorithm terminates, in a subsequent step those processors are classified as faulty which have only faulty adjacent interconnection links, since the failure of a CPU or all of its communication links are identical in diagnostic respect.

Algorithm 9 The fault classification algorithm of the initial stage

```
Require: $A_i$ set of test results collected by unit $u_i$
Ensure: $U^0[i], U^1[i]$ set of units classified as fault-free and faulty, $C^1[i]$ set of faulty links
  $U^0[i] \Leftarrow \emptyset$
  $C^1[i] \Leftarrow \emptyset$
  $\mathrm{NODES}_i \Leftarrow u_i$
  while $\mathrm{NODES}_i \neq \emptyset$ do
    Remove the next element $u_j$ from $\mathrm{NODES}_i$
    $U^0[i] \Leftarrow U^0[i] \cup u_j$
    for all $u_k \in \mathrm{N}(u_j)$ do
      if $u_k = \perp \vee \exists u_k \in \mathrm{NODES}_i \vee \exists u_k \in U^0[i]$ then
        Skip $u_k$ {non-existing or already processed neighbor}
      end if
      if $u_j \in \Gamma^{-1}(u_k) \wedge a_{jk} = 0$ then
        $\mathrm{NODES}_i \Leftarrow \mathrm{NODES}_i \cup u_k$
      else if $u_j \in \Gamma^{-1}(u_k) \wedge a_{jk} = 1$ then
        $C^1[i] \Leftarrow C^1[i] \cup (u_j, u_k) \cup (u_k, u_j)$
      end if
    end for
  end while
  for all $u_j \in U$ do
    if $\forall u_k \in N(u_j) : \exists (u_j, u_k) \in C^1[i]$ then {classification}
      $U^0[i] \Leftarrow U^0[i] - u_j$
      $u_j$ is classified as faulty, $U^1[i] \Leftarrow U^1[i] \cup u_j$
    end if
  end for
```

Working stage

After completing the initial stage a primary diagnostic view of the multiprocessor system is created. This view helps to enumerate the available resources and determine the starting system configuration. Based on the starting configuration the boot loader process can assign the user tasks to PEs and download the executable code to the processors. When the boot-up procedure is finished the application execution can begin. At the same time the distributed diagnostic algorithm enters the *working* stage.

In the working stage the syndrome is acquired and processed differently from the initial stage. During this stage all processors have at any point of time a consistent, up-to-date system-level diagnostic image. Therefore they do not need to refresh the complete image when they detect or get informed of a new diagnostic event (a unit failure or repair). Test results can also be communicated to other processors in an event-oriented way. Test are conducted periodically via < I'm alive > messages, as described previously. However, a given processor broadcasts a certain test result only if it differs from the previous ones, i.e., its neighbor was previously fault-free and now suddenly the test on this neighbor fails or vice versa. Since individual PEs are assumed to have high MTBF values, this mechanism assures that test result reports are communicated rarely in the system.

When an u_i processor detect a difference in the test results of its u_j neighbor unit, it

Table 3.1: Fault classification in the distributed algorithm

Symptom	**Processor**	**Links**
All tests on the unit pass	fault-free	all fault-free
Some tests on the unit fail, some tests pass	fault-free	some faulty
All tests on the unit fail	faulty	all faulty
The unit is unreachable, no test result reports are received about the unit	unknown	all unknown

immediately reports the new test result to all of the adjacent fault-free processors, then starts the fault classification procedure described below to identify the state of u_j. The recipient processors of the test result report indicating a new diagnostic event are of two kinds: either they are testers of u_j or they are not. Non-tester units of u_j simply forward the test result to other units and they start the same fault classification procedure as well. Tester units of u_j *re-test* the u_j neighbor unit to obtain a new test result, then they forward both the received test result report and their new test result. This way each accessible tester unit broadcasts a *test result report* (TRR) about the u_i *unit under diagnosis* (UUD).

Test result reports are received by all fault-free units in the respective connected subgraph of the system. The so obtained syndrome is analyzed by the fault classification algorithm defined by the state diagram in Figure 3.5. The figure describes the different states of the classification of a certain UUD (in our case unit u_j). An instance of this state diagram is maintained in each u_i processor for each other unit in the system, the actual status of the diagram indicates the fault classification of the respective unit at any point of time. Four possible fault classifications are possible as shown in Table 3.1. The layout of the state diagram in Figure 3.5 reflects the actual classification in its structure: the states are organized in columns, and each column corresponds to a certain diagnostic result.

Transitions between states are indicated by directed arcs in the diagram. A transition is activated by receiving a test result report about the UUD. There are six possible starting states, depending on the diagnosis of the given UUD in the initial stage. State 1 represents a fault-free unit without faulty interconnection links, while State 6 is a faulty unit with only faulty interconnection links. State 0 denotes an unreachable and thus unknown processor. State 3.4, State 4.3, and State 5.2 symbolize a fault-free UUD with 1, 2 or 3 link failures (the diagram is adapted to the case of Parsytec GCel, different state diagrams belong to other communication topologies).

We begin the description of the algorithm at State 1, assuming that the UUD is unit u_j, the diagnostic algorithm is executed by unit u_i, and that unit u_j was classified in the initial stage as fault-free. Also note that the diagram is tailored to the transputer architecture, i.e., a PE is equipped State 1 is located in the second column, implying that the UUD is diagnosed as fault-free as well as all of its communication links. The fault classification remains in State 1 until a test result is obtained by the $<$ I'm alive $>$ testing mechanism, or a message containing a test result report about the UUD is received from one of the tester processors. Local test results and test result report messages are processed differently:

Local test result. In this case the u_i unit is a tester of the UUD, and it verified the state

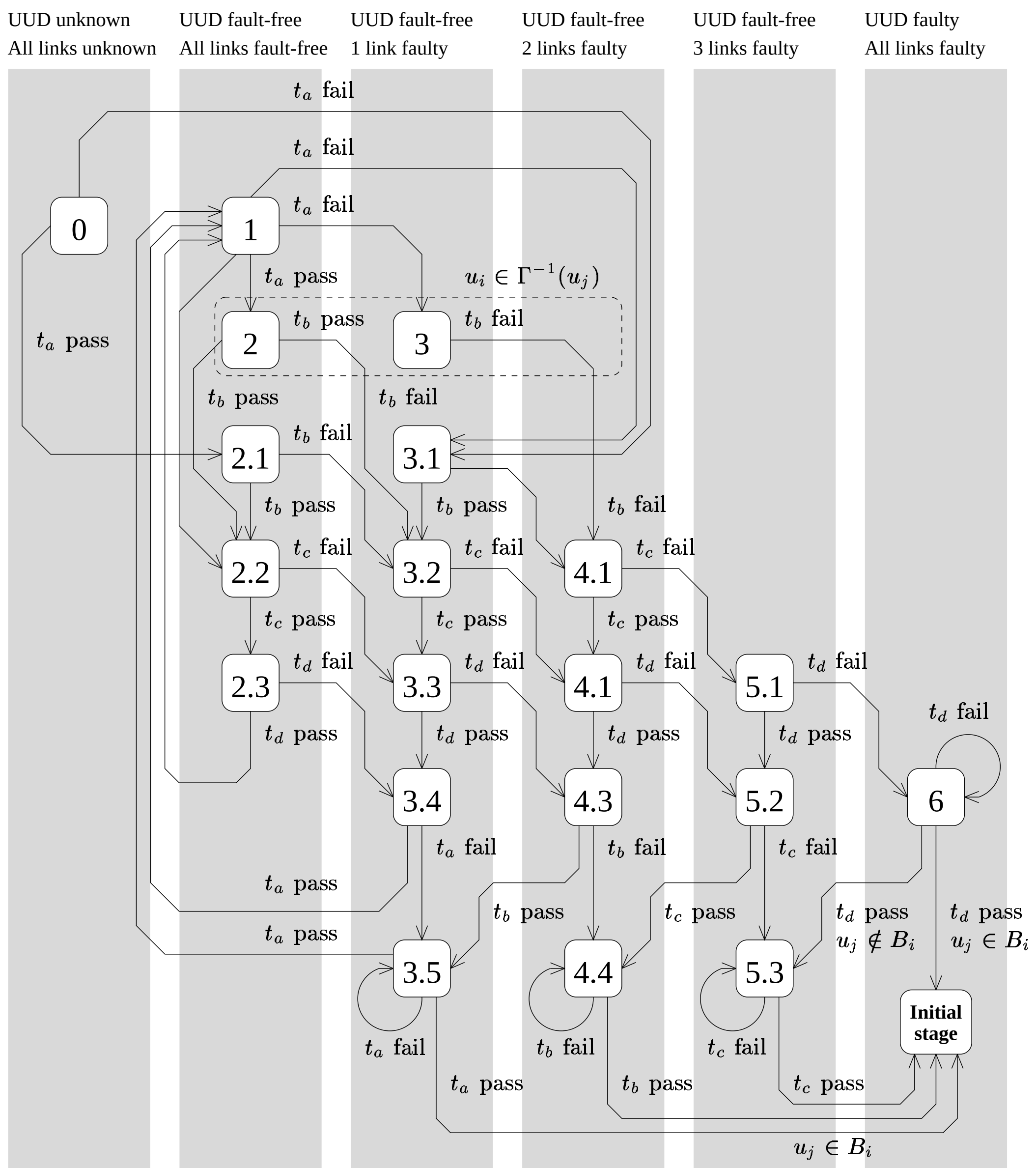


Figure 3.5: The fault classification algorithm of the working stage

of unit u_j by a local < I'm alive > test. If the test passed, then the classification does not change and the algorithm enters State 2.1. A failed test, however, indicates a new diagnostic event. The u_i tester unit prepares and broadcasts a test result report message to let the other reachable fault-free processor know of the failed test, then it moves to State 3.1. Note, that the failure of the UUD is not confirmed at this point, therefore it is still assumed to be fault-free. However, the interconnection link between u_i and u_j is classified as faulty.

Test result report message. When the diagnostic information about the UUD was acquired in a test result report message, two different activities must be performed depending on the topological relationship of the diagnosing PE and the UUD. If unit u_i is a tester of unit u_j, then it first must check the relevance of the received message. The message relevance rules are similar to the packet dominance rules of [78]. The message is *relevant*, if it is either more recent than the last TRR started by u_i, or it is concurrent and the sending tester processor has a higher priority than u_i (there is a static priority among tester units). The message can also be relevant if it was generated as a response to last TRR started by u_i. Otherwise, the included TRR is less recent or has a lower priority and so the message is declared to be *irrelevant* and is simply discarded.

When a new, relevant test result report is received, it dominates over the old TRR of the diagnosing PE, which is dropped. The tester u_i unit prepares a response message to the TRR by executing a new test on the UUD. Then the unit forwards both the received TRR and the new response TRR message. This mechanism assures that each processor eventually receives an up-to-date test result *from all the testers* of the UUD. These activities are performed in State 2 and State 3.

If the PE is not one of the neighbors or the incoming message is a response to the last TRR started by unit u_i, then the classification is either in State 2.1 or in State 3.1. In this case the unit only forwards the message and enters the next state: State 2.2, State 3.2, or State 4.1, depending on the received test result.

Further test result reports from the other testers of the UUD are obtained and analyzed from State 2.2 to State 6. The number of the subsequent states during analysis equals to the number of neighboring (testing) processors (in the case of the Parsytec GCel it is 4). Transitions are dependent on the received test results. The purpose of these states is to obtain all information from the tester processors. As more and more TRRs are received the classification of the UUD and its communication links is gradually refined.

If all of the tests on the diagnosed u_j unit have passed, the state diagram traverses the route: State 2 or State 2.1, State 2.2, State 2.2, and finally again State 1. During the whole process the classification of the unit and the links remains fault-free. When at least one of the tests on unit u_j fails, then the state diagram enters the third (States 3.x), fourth (States 4.x), and fifth (States 5.x) column, indicating one, two, or three communication link failures. If every tester found the UUD to be faulty, then the unit and all of its is communication links are classified as faulty (State 6). Processor failures and simultaneous breakdown of all adjacent communication links are diagnostically equivalent as they produce identical syndromes. Thus, at the end of the syndrome analysis process the state diagram will be in one of the following five states: State 1, State 3.4, State 4.3, State 5.2, or State 6; corresponding to a fault-free processor, one or more link failures, or a processor failure.

A time-out mechanism is used in receiving the test result report messages. Test result reports not received within the time-out period (e.g., due to an extremely large communication delay or due to the inaccessibility of the sender unit) are assumed to be missing. Missing TRR messages are taken into consideration during the diagnostic process as failed test results. If the message was not lost but only delayed, this mechanism may cause a diagnostic inconsistency (e.g., a fault-free communication link is temporarily classified as faulty), however, the inconsistency is eliminated when the message eventually reaches its destination. A distributed message ordering and message dominance scheme [86, 78] ensures that the valid TRR order is maintained. If none of the test result report messages are received within the time-out limit, or when the UUD is inaccessible, the classification remains unknown (State 0). This classification remains valid until a test result report concerning the UUD is received from one of its testers.

State 3.5, State 4.4, and State 5.3 provide the possibility of taking on-line repairs into consideration. Here an extra diagnostic process is required to assure consistency between the diagnostic images stored in processors belonging to different connected subgraphs. This situation may arise if faulty links or processors previously isolating two or more connected subgraphs have repaired, and so the separate subgraphs are joined. To avoid the potential differences of the diagnostic knowledge in the processors a special broadcast procedure could be performed. During this broadcast the various diagnostic knowledge of the nodes located in previously isolated subgraphs could be merged into one. Such a knowledge unification procedure would be both complex and message-intensive, therefore in our solution all of the units simply return to the *initial phase* when they detect that the dividing barrier(s) became traversable.

In the following an example is given to illustrate the diagnostic concept of the working stage. The example, shown in Figure 3.6, presents three diagnostic events in a system having 4×4 processors arranged in a two-dimensional (bordered) mesh topology. In the upper row the testing graph of the system is drawn, including the actual fault set and the syndrome. Not all of the tests are displayed, trivial test results and connections were omitted for simplicity. In the lower row the diagnostic image, corresponding to the actual fault situation and syndrome, can be seen. Units are numbered sequentially from left to right, unit u_1 is located in the bottom-left corner. The diagnostic image was generated by the last processor in the lowest row (unit u_4), marked by a grey ring. Assume, that two processors (units u_2 and u_6) were diagnosed as faulty during the initial stage. The example lists three subsequent diagnostic events: (a) the failure of the $c_{11,7}$ communication link in unit u_{11}, (b) the failure of unit u_{12}, and (c) the repair of the $c_{11,7}$ communication link.

The error in the $c_{11,7}$ communication link is detected independently by units u_{11} and u_7. Therefore, both of them start test result report messages stating that tests $t_{11,7}$ and $t_{7,11}$ failed. Recall, that a test compares the values (results of self-test programs) received in $<$ I'm alive $>$ messages from the neighbors with stored or processed local reference values. Hence, an error is detected in three different ways: the heartbeat message does not arrive within the predefined time interval, the TRR message contains incorrect checksum, or the received functional test result value carried by the message does not match the local reference value. Since both units u_{11} and u_7 are fault-free, the TRRs are valid. Tester units of u_{11} (u_{10}, u_{12}, and u_{15}) as well as testers of u_7 (u_3 and u_8, but not the faulty u_6) perform tests on these units and all the tests pass. Since the communication graph of the fault-free units is strongly connected, all of the fault-free processors will receive every test result report and identify the the $c_{11,7}$ and $c_{7,11}$ communication link as faulty. Although only the $c_{11,7}$ has a physical fault, the failure of the two adjacent links cannot be distinguished, as they produce identical syndrome.

Next, unit u_{12} fails. This way a barrier is created, cutting up the system into two separate parts. For this reason, the diagnosing u_4 unit receives only the test result report of unit u_8. TRRs from units u_{10}, u_{15}, and u_{16} cannot reach unit u_4, and the result of

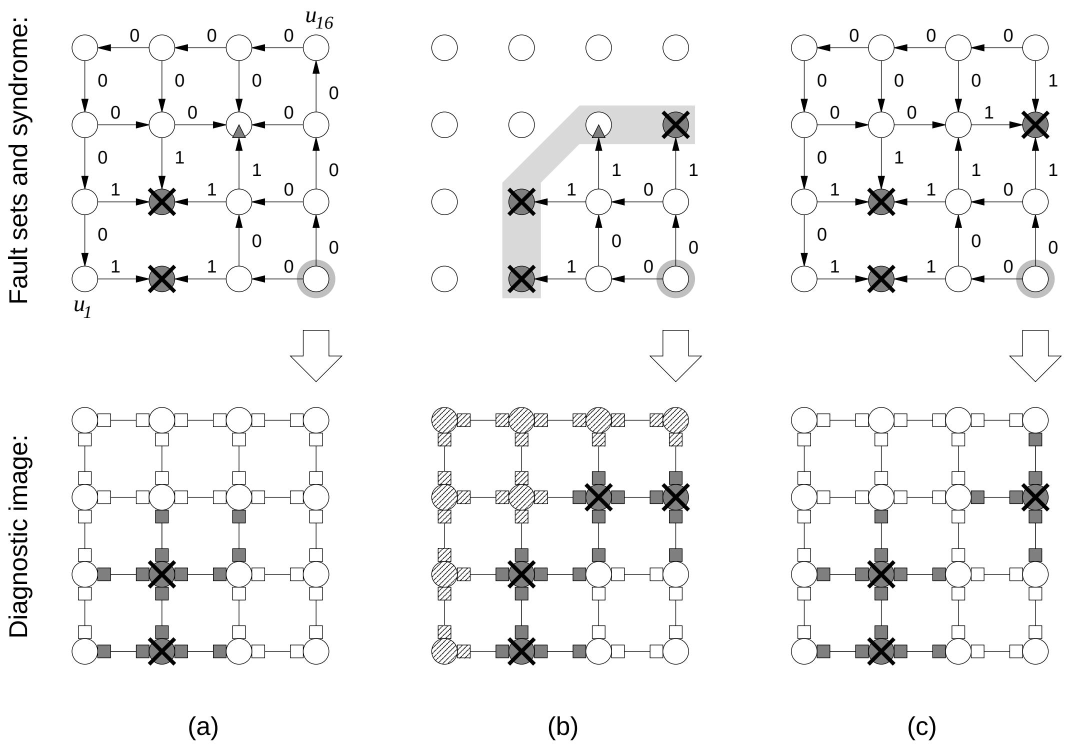


Figure 3.6: Diagnostic example: (a) link failure, (b) additional unit failure, (c) link repair

these tests is considered as *failed*. Consequently, unit u_{12} is diagnosed as faulty, and the classification of unit u_{11} is also changed from '1 link fault' to 'faulty unit'.

In th third step, the $c_{11,7}$ communication link is repaired. Units u_{11} and u_7 (which belong to different subgraphs) notice that again they receive correct $<$ I'm alive $>$ messages from their neighbor. They both broadcast test result messages stating that tests $t_{11,7}$ and $t_{7,11}$ pass. Other fault-free processors receiving this TRR notice, that unit u_{11} (or from the viewpoint of the other subgraph: unit u_7) was part of a barrier and now is repaired. The strong connectivity of the communication graph is restored. Therefore, they all immediately return to the *initial stage* and start a complete diagnostic session. At the end of the initial stage they correctly identify the u_2, u_6, and u_{12} units as faulty.

As it was obvious from the description of the algorithm, much time is spent by determining the node accessibility graph related to a given unit. The whole algorithm can be made more efficient by improving this procedure. The objective of the *barrier detection* mechanism is to speed up the execution of the diagnosis algorithm. The node accessibility graph contains all the fault-free processing units connected with the considered node via a path of fault-free processors and communication links. To generate this graph, the fault-free neighboring nodes starting form a particular node have to be discovered recursively. For large networks this is a time-consuming task.

The main idea behind *barrier detection* is to examine the faulty units in the system, instead of the fault-free ones. Faulty processing units block some of the communication paths and may form barriers dividing the system into separate parts. Without barriers every fault-free processor can exchange information, so the search for unreachable regions (i.e., the generation of the node accessibility graph) is unnecessary. That gives a possibility

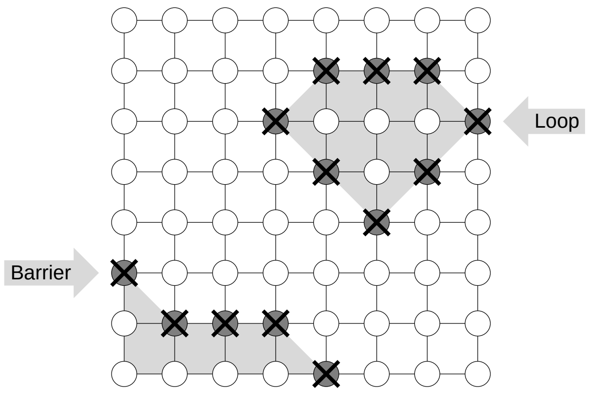


Figure 3.7: Barrier and loop

to accelerate the process by using a detection method to recognize whether or not the fault pattern present in the system forms such obstacles. If no barrier was found, then the communicating unit can expect test result report messages from all of the fault-free processors, else the node accessibility graph has to be obtained to locate the unreachable units. The applicability of the barrier detection procedure is based on the assumption that the number of fault-free nodes in a real system is probably (much) higher than the number of faults. Consequently, a depth-first graph search method operating on only faulty nodes will be significantly faster than a depth-first graph search algorithm for the set of reachable fault-free and faulty nodes.

Two kinds of obstacles may be formed by faulty processors: barriers and loops. They can be distinguished by their different character (this difference is built in the barrier detection process, so it can even classify the found obstacles). A continuous path of neighboring faulty nodes extending from one border of the grid structure to another is considered to be a *barrier*. The barrier cuts up the system into two separate parts. A circle formed by neighboring faulty units is a loop. The loop isolates its inner region from the rest of the system. Note, that barriers exist only in bordered topologies (such as the 2D mesh topology of the Parsytec GCel or the 3D mesh topology of the Parsytec GC series), in completely regular topologies (like the 3D toroidal mesh topology of APEmille) only loops can be formed.

The detection procedure employs the following mechanism: the initial (faulty) processor (which potentially belongs to a barrier) is placed in the NODES$_i$ buffer. Node data is recorded in the following form: $[u_x, u_y]$, where u_x is the actually placed node, while u_y is its ancestor (i.e., u_x was reached by extending u_y). The node data elements stored in the NODES$_i$ buffer are taken out sequentially one-by-one. The most recently removed node is remembered in the B_i set, then it is extended, that is all of the faulty neighbors of this node are examined. If a certain neighbor does not exist (i.e., because in the given direction the grid topology has a border), the ancestor node is marked in BORDER$_i$. When the detection procedure encounters a second border, then the set of faulty units in B_i forms a *barrier*. If the algorithm runs into the same faulty node two times from different directions (this is why the ancestor node is necessary to remember), then the set of faulty units in B_i is classified as a loop.

The search process continues until no more faulty neighbors are found. In some cases, when faults form a loop touching two borders of the processor network, the barrier detection process started from certain nodes finds the fault pattern to be a loop, while from other nodes a barrier is detected. However, despite of the difference in classification, the result indicates that an separation exists, and that the node accessibility graph must be generated. The pseudo-code of the barrier detection procedure is presented on Algorithm 10.

Algorithm 10 The barrier detection algorithm

```
Require: $U^1[i]$ set of units classified as faulty, $u_j \in U^1[i]$ starting node
Ensure: $B_i$ set of units forming a barrier or loop, or no barrier exist
  $B_i \Leftarrow \emptyset$
  $\mathrm{NODES}_i \Leftarrow [u_i, \perp]$
  $\mathrm{BORDER}_i \Leftarrow \emptyset$
  while $\mathrm{NODES}_i \neq \emptyset$ do
    Remove the next element $[u_x, u_y]$ from $\mathrm{NODES}_i$
    $B_i \Leftarrow B_i \cup [u_x, u_y]$
    for all $u_k \in \mathrm{N}(u_x)$ do
      if $u_k = \perp$ then
        if $\mathrm{BORDER}_i \neq \emptyset$ then {second border}
          exit Barrier was detected!
        else
          $\mathrm{BORDER}_i \Leftarrow u_k$ {first border}
        end if
      else if $u_k \in U^1[i]$ then
        if $(\exists[u_k, u_z] \in \mathrm{NODES}_i \vee \exists[u_k, u_z] \in B_i) \wedge u_z \neq u_x$ then {already processed node}
          exit Loop was detected!
        end if
        $\mathrm{NODES}_i \Leftarrow \mathrm{NODES}_i \cup [u_k, u_x]$
      end if
    end for
  end while
  exit No barrier containing unit $u_j$ was detected.
```

3.2.2 Implementation details

The realization of the working stage can be done in different ways, depending on how the processing power is divided between the diagnostic process and the running application. In the following the implementation of the algorithm will be described. Alternative approaches to the testing mechanism of neighboring processors, termination rules, distribution of local test results, and processing of diagnostic information are presented to show that the algorithm can be adapted to several systems and requirements.

The main implementation structure of the working stage of the algorithm is shown in Figure 3.8. If no fault event is detected, the algorithm periodically tests the neighboring processors. Testing is accomplished by assigning independent process modules to each tested neighbor unit. If the tests detect an error in the respective neighbor processor, exception handling is invoked by issuing an error indication from the corresponding *testing module*

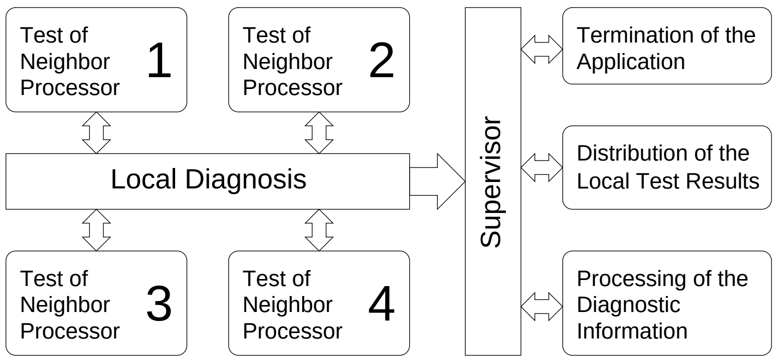

Test of Neighbor Processor 1
Test of Neighbor Processor 2
Local Diagnosis
Test of Neighbor Processor 3
Test of Neighbor Processor 4
Supervisor
Termination of the Application
Distribution of the Local Test Results
Processing of the Diagnostic Information

Figure 3.8: Software modules of the distributed algorithm

to the *local diagnosis module*. The local diagnosis module gives control to the *supervisor module*, which handles the exceptions caused by the detected error. The supervisor module activates the modules responsible for *terminating the current application*, for *distribution of the local test results*, and for *processing of the diagnostic information*. The purpose and realization of the different software modules and their interaction is presented below:

Test of neighbor processors. Each testing module is comprised of three threads: one for receiving local test results from the neighboring processors, one for sending such messages (< I'm alive > message), and one for evaluating the result (i.e., verifying whether the responses are delayed or incorrect), respectively. If the evaluation indicates a faulty behavior of the neighboring processor, the thread sends an error message to the local diagnosis module. This < I'm alive > testing mechanism offers a possibility to control the testing related run-time overhead. On the one hand very precise and thoroughgoing self-tests can be used resulting in a decreased application performance due to the more intensive diagnosis process. Such self-tests can take two forms: either realized in software or in hardware.

On the other hand, the use of < I'm alive > messages indicating the alive or dead state of a processor reduces the requirements of processing capacity to a fraction, thus yielding more computational power to the application. Although this kind of tests is easy to process, it can only detect permanent processor and link faults, and more post-processing is required later. However, since the application is quickly stopped after error detection, there is a sufficient time remaining for more finely granulated tests and post-processing in a subsequent separate testing stage.

Termination of the application. The function of this module is to interrupt the execution of the application on all of the PEs as soon as possible. This is necessary for the prevention of error dissemination and to decrease the fault latency in the multiprocessor system. If the application is quickly suspended, the probability of error propagation is reduced, because no further communication with the exception of diagnostic information transfer will take place.

The module initiates a fast broadcast for quick termination of the application. The broadcast messages are received at every node by the local diagnosis module, which initiates immediately exception handling. No specific routing mechanism is required for the implementation of the fast broadcast, as the existing routing mechanism of the Parsytec GCel system extended with a high-level, fault-tolerant communication protocol is fully sufficient. The broadcast is based on flooding the multiprocessor with a so-called stop message indicating the occurrence of an error. Each processor sends this message to all of its fault-free neighbors. The neighbors forward the message to their neighbors (excluding the sender), continuing this process until the message arrives at every accessible fault-free node. The advantage of using flooding is its easy algorithm and its inherent fault-tolerant behavior; all fault-free processors within a connected subgraph are reached.

Distribution of the local test results. The module for distribution of the local test results is activated after terminating the application. At first, only the neighboring processors of the faulty processor start a separate testing stage, executing fault localization tests. The assumed causes are faulty links and faulty processor components. These additional tests even assure the classification of faults as temporary or permanent. The outcome of these tests as well as the tests results obtained by the error detection mechanism constitute the local test results.

The distribution module transfers the local test results to the supervisor module of each processor using the fault-tolerant broadcast. Different criteria can be used for terminating the distribution stage. Our first implemented criterion was to wait until all the local test results from each tester of the faulty node have been received, but we found that this method is not robust against errors during the diagnosis (e.g., lost messages due to node failure). A time-out criterion is used for elimination of this lack of robustness. The distribution process waits for a certain time interval, in which all of the local test results must be received. The main advantages of this method are its safety and simplicity, but the optimal time-out limit must be estimated in the design stage.

Processing of the diagnostic information. Syndrome decoding is invoked by the supervisor module on receiving a new local test result. This way the test result report distribution and the fault localization modules are executed alternatively. Therefore, the processor will not be idle even if the time-out limit used in the distribution process is not optimal.

3.3 Adapting the LID approach to distributed systems

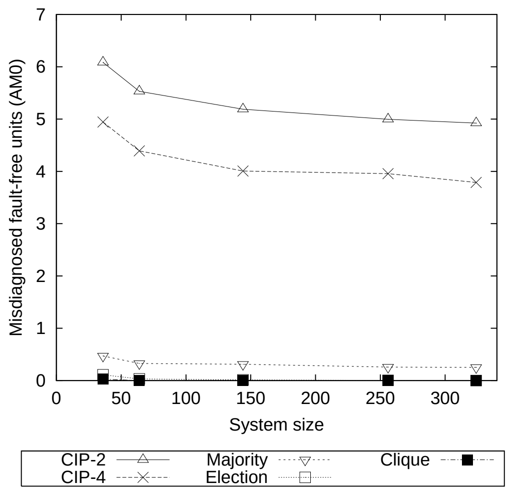
The previous sections described an event-driven distributed diagnostic algorithm developed by the author for application in the Parsytec GCel system. It could be seen that the task of distributed diagnosis is not limited only to syndrome analysis, but includes additional problems like the reliable dissemination of test result reports, determination of the real message order, deadlock-free communication in spite of faults, etc. Even the distributed syndrome analysis process itself differs in many aspects from its centralized counterpart. Each unit performs diagnosis using both first-hand knowledge (locally performed tests) which it assumes to be reliable, and second hand knowledge (collected test result reports). The fault classification task is executed by all processors in parallel, independently from others.

Communication with the detected faulty units is forbidden. Therefore, in the presence of faulty units the local syndrome is only partial. The results of tests performed by failed processors is not included in the diagnostic information analyzed in the fault classification procedure. This can be interpreted as the methodology of distributed fault tolerance employs a safe but minimalist PMC-like model (yet disregarding even some implications present in symmetric test invalidation). The diagnostic process employs a bootstrap-like approach. The set a fault-free units is gradually extended starting from the diagnosing unit and accepting neighbors with a passed test result by an already reached fault-free tester processor.

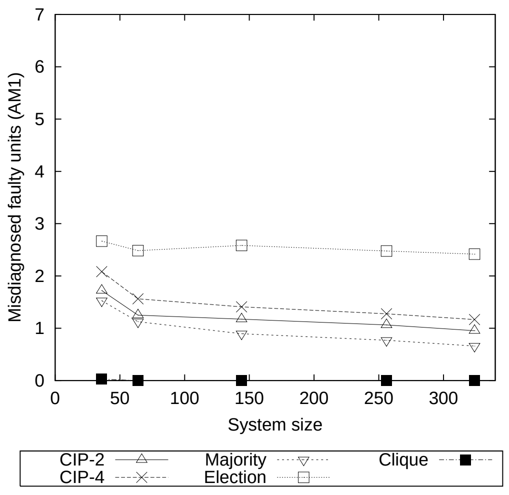
The question arises if the generalized test invalidation model and all its consequences can be applied in a distributed system. With relevance to the topic of this document, we are interested in how the concept of local information diagnosis could be employed in distributed environments, and particularly, the Parsytec GCel machine. The first problem to be considered is the dissemination of the testing information. In the framework of centralized diagnosis it is assumed that the central syndrome analyzer device receives all of the system-level test results reliably from each processor, and the ways of achieving this are outside the scope of centralized methods. However, this problem is not to be pushed aside in the case of distributed diagnosis.

To remain on the safe side, we assume that faulty units are unreliable from the communication viewpoint in all failure modes. Thus, messages sent by faulty units are potentially invalidated and cannot be accepted. To extend the set of available testing information the test job must be made independent from the application processors and delegated to external tester coprocessor devices. Of course, the tester devices need also to be tested by the adjacent processors, and these test results must also be considered during the diagnosis. The processors must be able to access the tester coprocessors of the adjacent units independently and reliably. Assuming, that the tester coprocessors are not capable of communication, the test results of units isolated by faulty processors are still not obtainable. Nevertheless, the test results of processors adjacent to fault-free units became available, which means in general a significant improvement. In any case the diagnostic information available for fault classification will be constrained.

However, the amount of diagnostic information used in diagnostic can also be constrained on purpose. The basic idea of local information diagnosis is to use a reduced inference amount in the syndrome analysis process. In many applications the interdependence of the distributed tasks is low, i.e., the processing elements are cooperating only with other PEs located near to them, and are completely ignorant of far PEs. In this case only the small environment of the given unit must be reliably identified. This gives a possibility for reducing the volume of disseminated testing messages by assigning *scope* (that is, a maximum travel distance) to the messages. Units receiving the message farther than the specified maximum distance refuse to forward it, thus the message will not reach processors outside of its scope. This creates a communication model where units gather testing data



(a) Average misdiagnosed fault-free units



(b) Average misdiagnosed faulty units

Figure 3.9: Average misdiagnosed units in a bordered two-dimensional mesh

only from their k-neighbors (where k is the maximum distance defined by the scope).

Suppose that we employ one of the developed probabilistic LID diagnostic algorithms and use implication chains of l-step length. In this case units which are reachable in less than m interconnection links and $l+m \leq k$ will be diagnosed using the whole considered implication set. These units will be classified as if they were located in an infinitely large system and a centralized diagnostic algorithm were used. But what happens to the diagnosis of those units for which $l+m > k$? A good analogy can be found in bordered topologies. The peripheral processors of a bordered two-dimensional mesh topology have similarly incomplete testing information as the distant units in the constrained communication model. To illustrate the influence of the topology border, Figure 3.3 presents measurement results performed in a 12×12 two-dimensional bordered mesh topology. As it can be seen, opposite to the toroidal topologies the lower system sizes induce less accurate diagnosis, and the values approximate the lower limit from above. This tendency is caused by the incomplete diagnostic information at the borders information. In a larger system relatively less units belong to the border are, therefore their effect becomes weaker.

To integrate the developed probabilistic LID diagnostic algorithms in the Parsytec GCel system the problem of link diagnosis must be solved. For this purpose we need to derive a test invalidation model that takes into account also the interconnection links. Consider two adjacent processors called CPU$_1$ and CPU$_2$ connected by the L_1^1 and L_2^3 communication links, as illustrated in Figure 3.3. Suppose that the processors test each other complying to the symmetric invalidation model, and that the failure of the communication links is always perceptible. These assumptions result the comprehensive test invalidation model described by Table 3.3(a). Unfortunately, this table is not appropriate for diagnostic use, as the methodology of generalized test invalidation can handle only two interacting units. Studying the component interdependences pair-wise, several two unit "projection models" can be created. As an example, Table 3.3(a) shows the relationship of the CPU$_1$ processor and the L_1^1 interconnection link, obtained by omitting the rows and columns of Table 3.3(a) related to CPU_2 andL_2^3.

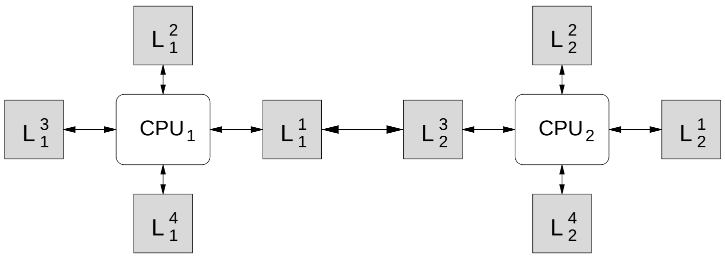


Figure 3.10: Parsytec GCel communication scheme

Table 3.2: Test invalidation including interconnection links

CPU_1	CPU_2	L_1^1	L_2^3	a_{12}
0	0	0	0	0
0	1	0	0	1
1	0	0	0	X
1	1	0	0	X
X	X	1	X	1
X	X	X	1	1

(a) Complete test invalidation scheme

CPU_1	L_1^1	**Result**
0	0	0
0	1	1
1	0	X
1	1	1

(b) The relationship of CPU_1 and L_1^1

Note, that the relationship of the CPU_1 and L_1^1 is essentially the asymmetric test invalidation model. The same procedure can be carried out to examine the other pair-wise combinations of the CPU_1, CPU_2, L_1^1, and L_2^3 devices. Then, the partial testing models can be combined to form a comprehensive test invalidation model, as seen on Figure 3.3. The ellipse around the three logical test connections between CPU_1 and the adjacent devices indicate that they all correspond to the same t_{12} physical test connection. Therefore, when extracting one-step implications from the syndrome for all of the three logical test connections the same a_{12} test result must be substituted in the parameterized inference rules.

Before the created comprehensive test invalidation model could be used for diagnostic purposes, the meaning of the one-step implications must be studied in the new model. From an $a_{12} = 0$ test result we can infer that "if CPU_1 is fault-free, then CPU_2 is also fault-free" (due to the symmetric invalidation between the processors), and that "the L_1^1 and L_2^3 interconnection links are both surely fault-free" (due to the asymmetric invalidation between processors and links). However, the interpretation of the $a_{12} = 1$ test result is more intricate. It implies, that "if CPU_1 is fault-free, then *either* the CPU_2 or any of the L_1^1 and L_2^3 interconnection links are faulty."

It is important to notice, that the above OR connection of the fault hypotheses is not handled by the presented local information diagnostic algorithms. The inference mechanism of these algorithms is *proposition calculus* or *zeroth-order predicate calculus*, where the consequences of the implications are in AND connection and are fulfilled simultane-

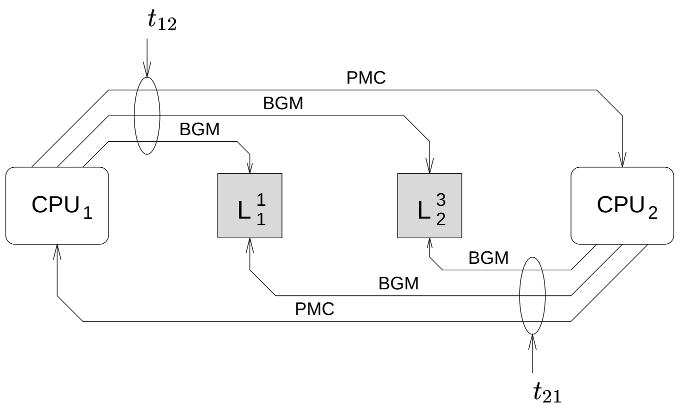


Figure 3.11: Comprehensive test invalidation model of the Parsytec GCel

ously. The matrix multiplication and DIL techniques used for transitive propagation of implications conform to the *proposition calculus* and cannot be adapted to handle alternative consequences. Therefore, other inference propagation and fault classification techniques must be used. One such opportunity is to use the AND-OR graph representation known in the literature of artificial intelligence research are [89], instead of the inference graph. The inference propagation and fault classification in the AND-OR graph representation can be for example solved by the A0* algorithm, which is a heuristic graph-searching method capable of dealing with alternatives. Yet, the whole diagnostic problem must completely be re-formulated within the new framework, therefore this approach exceeds the scope of our document. Another possible solution is provided by the theory of constraint-based diagnosis, which is outlined in Section 4.6.

Chapter 4

Fault tolerant architecture of large computing systems

Even in high-performance computing environments the amount of computation involved in solving the targeted scientific problems may require several days or weeks of continuous, uninterrupted operation. The complex structure and large number of devices constituting these systems makes them error-prone, despite the excellent quality of the components used. Therefore, without built-in fault tolerance one may expect multiple fault occurrences during a single job execution. System engineers realized the consequences of this fact and the recent parallel supercomputers are all equipped with built-in support for fault-tolerance in order to increase the availability of the system.

This chapter introduces a typical example of number-crunching multiprocessor machines, the APEmille parallel computer, and describes the design of a backward error recovery scheme developed for APEmille. In the introductory part we present the hardware architecture, then the built-in hardware support for diagnosis and the fault diagnostic procedure are described. The selection of the appropriate error recovery mechanisms required the derivation of a failure model of the APEmille components. The failure model and the assumptions on the failure semantics of different system components are explained. Based on the component failure semantics a recovery strategy for each component is outlined, giving alternatives for tolerating single or multiple component faults. The problem of providing a reliable storage to record the recovery information is also considered. The closing part of the chapter examines the possible methods of integrating the developed LID fault diagnostic algorithms in the created fault tolerance scheme of APEmille.

4.1 Illustrative system: the APEmille parallel computer

APEmille is the latest generation of the APE family of supercomputers. The APEmille project is a joint effort of several European research institutes to develop a number crunching machine optimized for applications in particle physics, particularly *Lattice Gauge Theory* simulations. The machine is an evolution of APE100 computer, which was commercially available from the Quadrics Supercomputers World Ltd., situated in Italy. APE100 is a versatile parallel processor capable of delivering 100 GFlops peak performance in a 2048-

node configuration. The new machine has roughly one order of magnitude higher peak computational performance than the preceding generation (scalable from 4 GFlops to 1 TFlops). It also incorporates many architectural enhancements: *local addressing* capability at each processing element, *broadcast* and *soft* routing methods providing data exchange between two arbitrary nodes, and *independent partitioning* of the system to concurrently run different user programs [90]. The APEmille computer is now next to the assembly and test stage before the final release; and its controlling software, the APEOS operating system, is also under extensive development.

From the user's point of view APEmille is a parallel computer consisting of processing elements optimized for floating-point arithmetic. The PEs are arranged in a three-dimensional mesh topology with toroidal wrapped-around interconnections in all the six spatial directions. They have private memory banks, thus different PEs can independently work on their own specific data. The machine has a *Single Instruction Multiple Data* (SIMD) architecture, i.e., every processor executes exactly the same instruction in each step of the computation.

The APEmille hardware architecture is built up of three hierarchically structured levels. On the lowest level there is the so-called *Processing Board* (PB) or *cluster*, the smallest independent functional unit of the machine. It is built up of three basic components:

1. application processors called *Jmille*, executing the user programs,
2. control processors called *Tmille*, performing mainly the program-flow control, global address generation for each Jmille, signal handling, and common integer operations,
3. communication processors or *commuters* called *Cmille*, managing the data exchange and providing hardware diagnostic support.

These devices are realized as custom designed integrated circuits. A cluster contains eight Jmille processors, one Tmille control processor, and one Cmille commuter. The eight Jmille processors are logically placed in the vertices of a cube. Boards can be *wrapped* by setting an appropriate Cmille register. A wrapped board is logically disconnected from other boards of the system, and behaves as if it were a small 3-degree binary hypercube. This feature is utilized during the system-level diagnosis of the Jmille units (see Section 4.2). The components and arrangement of a Processing Board are shown in Figure 4.1.

Jmille supports arithmetic operations on both integer and floating point data words. Floating point data can be of scalar and complex type, and of single and double precision. Two floating point operations (a so-called *normal* operation: $a \times b + c$), or an integer instruction can be carried out at each clock cycle. The pipeline structure of the processor is relatively long. The operands and/or the result are stored in a large *register file* (RF) composed of 512 registers. This register file provides a fast temporary storage, achieving better performance in classes of scientific computations that require frequent access to *all* elements of a large data set than the traditional cache-based organization [91]. Each Jmille node has a local memory for its own exclusive use. Memory addresses can be global, generated by the Tmille processor and sent to every Jmille on the same PB. Global addresses can furthermore be modified with a value local to the nodes, giving additional flexibility to support a wider range of applications.

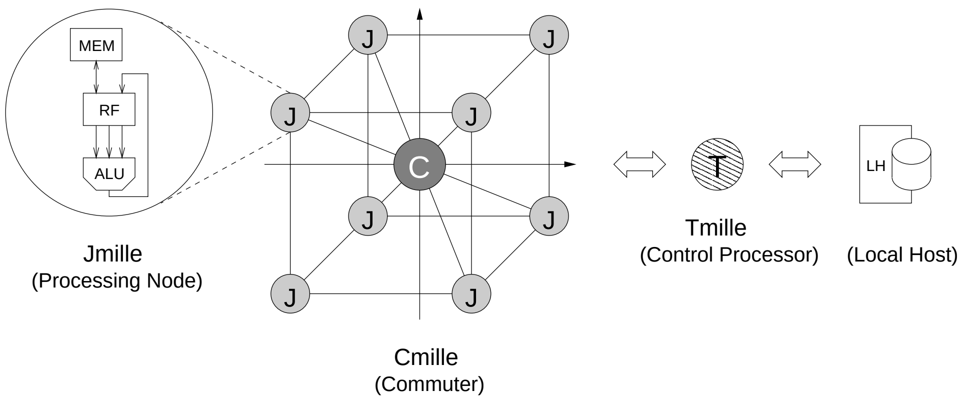


Figure 4.1: The logical structure of a Processing Board

Tmille also has its own data memory, an integer *Arithmetic and Logic Unit* (ALU), and an *Address Generation Unit* (AGU). The processor has a short pipeline and a very long instruction word, which can specify the operation of several independent devices. Tmille is also equipped with a register file having 128 registers, that is able to feed data to at most four different devices at each clock cycle. In each clock cycle Tmille fetches the next instruction from its own memory and broadcasts it to the eight Jmille nodes. If necessary, it also generates a global address for accessing the operands of the instruction, and broadcasts it similarly. Finally, Jmille fetches the operands from the register file or the memory using the address received from Tmille, and performs the instruction. All Tmille processors belonging to the same partition execute the same instruction stream and are initialized with a common set of values.

The Cmille commuter has both a simple topological structure and a restricted set of capabilities. Each Cmille is connected to the eight Jmille units and the Tmille of its cluster. Additionally, it is connected to the six nearest neighbor commuters of the adjacent Processing Boards. The Cmille circuitry is actually partitioned into four slices. The slices are identical and perform the same operations of different data sections. Every message packet sent by a Jmille is divided up into four distinct parts, and each Cmille slice implements the routing for one part of the packet.

The next hierarchy level over the Processing Boards is called the *APE Unit.* APE Units are built of four clusters connected to an external, stand-alone computer called the *Local Host* (LH). The Local Host computers act as supervisors over the attached boards and provide disk storage services. The four boards and the Local Host are hosted by a custom backplane, which is implemented according to the Compact PCI standard. A Processing Board is integrated in a single plug-in card, while the Local Host is situated on another card. Figure 4.2 illustrates the hardware layout of an APEmille PB with its supervising Host computer. Each PB is interfaced to the corresponding LH by a dedicated PCI bus. Using this bus the hosts can read and modify the memory and registers of the attached Jmille, Tmille, and Cmille processors. On the other hand, Tmille units can trigger interrupts to signify exceptions, service requests, or local conditions. These signals are handled by the *Root Board* built in the LH. The host computers have a PC-compatible architecture, and incorporate a local disk unit on which they store the local portion of the operating system

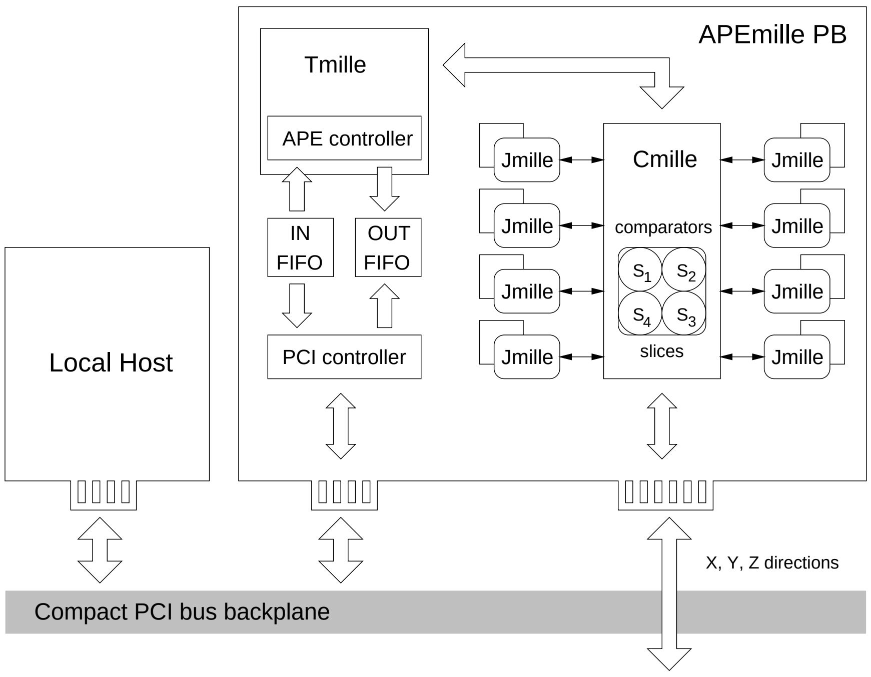


Figure 4.2: Hardware layout of a Processing Board

and application-specific data.

A configuration of four APE Units housed in a standard rack is called a *crate* (see Figure 4.3). A rack contains the custom Compact PCI backplane with 16 APEmille Processing Boards on one side, and 4 Local Host Boards with supplemental Service Boards (Control Network controller cards, SCSI controllers, Fast Ethernet cards) on the other side. The hosts of different APE Units are cooperating over a general-purpose, high-performance Control Network. Also attached to this network is a special computer acting as the *Global Host* of the entire APEmille machine. This central host computer manages the global disk storage and collects the global signals like halt requests, exceptions, `if` conditions, etc. Global signals are received from all parts of the machine by the Root Board plugged in the Global Host. Several crates can be stacked on top each other until the system reaches its maximum configuration of 4096 ($64 \times 8 \times 8$) processing elements. A large APEmille machine can be split up into logically independent partitions, making it possible for multiple users to simultaneously run separate programs.

The interconnection between Processing Boards is routed via a synchronous network managed by the commuters on the boards. The interconnection network is optimized for low latency communication between the neighboring units [92]. There are three main groups of routing capabilities:

1. The basic routing method is called *rigid* routing. In this case a single destination address is specified in the form of a 3-tuple: $(\Delta x, \Delta y, \Delta z)$, indicating the offsets of the destination nodes relative to the source nodes. That is, the network of PEs is translated rigidly by a given vector.

2. An enhanced routing method is the *broadcast*. It can be either one-, two-, or three-

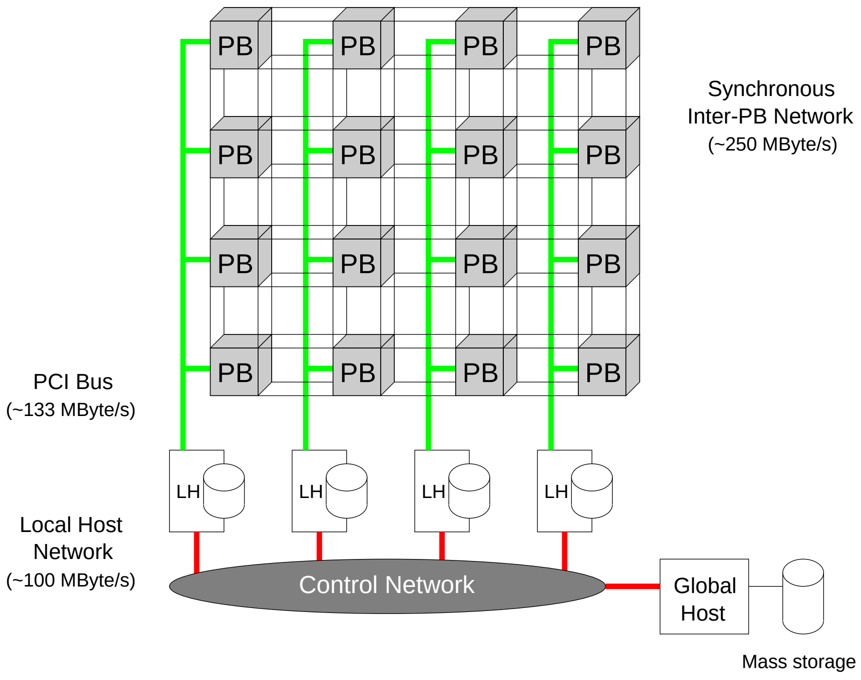

PB
Synchronous Inter-PB Network (~250 MByte/s)
PCI Bus (~133 MByte/s)
LH
Local Host Network (~100 MByte/s)
Control Network
Global Host
Mass storage

Figure 4.3: The APEmille crate

dimensional. In case of a 3D broadcast, one PE acts as the source, denoted as the *broadcaster* node. Data from the broadcaster is delivered to each other PE in the machine. In two-dimensional mode, the inter-processor network is divided geometrically into planes, with each plane having its own broadcaster. The one-dimensional mode is analogous to the previous ones.

3. Under development is the most advanced communication form realized by *soft* routing, where each sender node provides a destination for its own data block. This communication mode is useful to handle data arranged in less regular structures, but due to the more complicated routing mechanism involved, it has a lower bandwidth and higher latency.

The communication between two Jmille processors takes place in three phases. At first, each sender Jmille transmits its data block to the associated Cmille commuter. Then, all Cmille units route data to the specified destination. In the third step, data arriving to the Cmille connected with the destination Jmille nodes are passed to the receiving processors. There is an apparent contradiction between the fact that data blocks may take different time to distant recipients and the strict synchronism of the instruction execution of PEs. This problem is solved by delivering data packets only when all of them have arrived. In practice, the network of Cmille units produces a kill signal that stops machine operation until data are ready to be loaded.

4.2 Built-in support for fault-tolerance in APEmille

A novel feature of the APEmille computer is that low-level diagnostics support is integrated in the system. The testing of the Jmille, Tmille, and Cmille units is done transparently on the hardware level. The main testing mechanism employed is *comparison.* For this purpose self-checking comparators are integrated in the circuitry of the Cmille communication processor. The comparators have self-checking design which detects single stuck-at faults. All comparators are composed of four identically structured slices, each of them comparing $1/4^{\text{th}}$ of the packet data provided by the same pair of adjacent Jmille nodes. The executed comparisons are OR-ed in sequence in Cmille registers. The slices can interchange data parts by executing shift operations. This technique ensures that a a single faulty commuter slice cannot mask Jmille faults.

The three basic components (Jmille, Tmille, Cmille) of the Processing Boards are tested separately. Pairs of Tmille processors or pairs of Cmille commuters can be tested by comparing the output sequences they write on the bus. These units are controlled in a uniform way and perform the same sequence of instructions. They operate on the global variables and the instruction flow process identical data, which are identical due to the SIMD execution model. For this reason, the testing of the Tmille and Cmille units can proceed concurrently with the computation assigned to the machine.

The test of Tmille processors consists of the comparison of checksums. The checksums are generated independently at each Tmille during the fetching of the next instruction. They are computed by the double-rail encoding of the instruction code parity. The checksums of a pair of Tmille processors T_x and T_y are compared by a pair comparators K_{xy} and K_{yx}, residing respectively on processing boards PB_x and PB_y. The output of the comparisons is accumulated in a special Cmille register. The number of mismatches is counted, and if the value of this counter exceeds a predefined threshold an exception is generated. Since the comparisons are performed concurrently with normal computation and their results are collected over a long time period, it is reasonable to assume that there will be at least one mismatch between a faulty and fault-free Tmille processor, and the error does not remain undetected.

The Cmille commuters are tested quite similarly, however, the checksums are generated in a different way. The sender Cmille unit C_x computes a double error detecting, single error correcting code, and adds it to every data block to be transmitted. The receiving Cmille unit C_y contains a decoding device H_{yx} that verifies the incoming data block using the attached checksum. Similarly, there is a decoder H_{xy} in C_x to detect errors in transmissions from C_y. The decoder outputs are encoded in double-rail code and are accumulated in special registers. Although single bit errors are corrected on-the-fly, their number is counted and exceeding a predefined threshold raises an exception. Double bit errors are detected but cannot be corrected, so double bit error detection causes an immediate exception.

The Tmille and Cmille units are diagnosed periodically, based on the comparison syndrome obtained during job execution in `run` mode. If the diagnosis identifies at least one faulty component, the machine is switched into `equal` mode and a special diagnostic session is started to check the Jmille units as well. The diagnostic session is also activated following

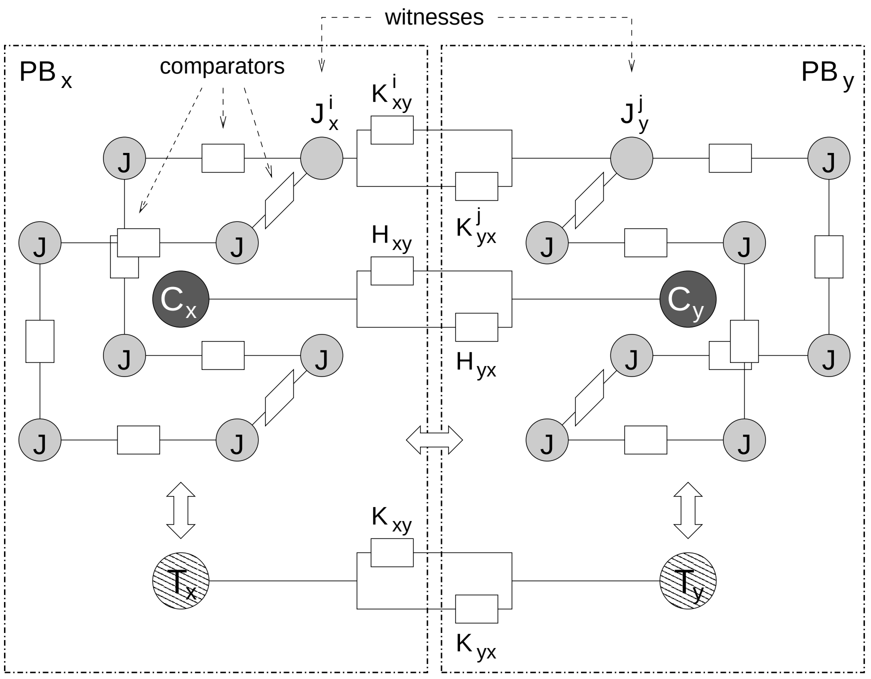


Figure 4.4: Testing the components of Processing Boards

the successful completion of a job.

Although the Jmille units also execute identical instructions, they normally work with local data that varies for each node. To make the test-by-comparison approach also applicable to the Jmille units, each Jmille is assigned with the same job and initialized with the same values in a special *diagnostic session.* The testing of Jmille processors is carried out in the so-called `equal` mode. Equal mode is composed of two phases, called the *Local Equal* and the *Remote Equal* phase, correspondingly. The first, Local Equal phase creates a preliminary, board-level diagnostic image. The second, Remote Equal phase refines and extends this image to the system-level by unifying the diagnostic information collected at each PB. The following steps are performed after entering in `equal` mode:

Local Equal. In this phase an *intra-cluster* diagnosis is performed: each board is wrapped and a logical ring is formed among the Jmille units of the PB (Figure 4.4). Then, Jmilles are loaded with the same set of instruction and memory values. When the test program is executed, Jmille units read/write synchronously the same data, to/from their respective memories. Every memory access and instruction fetch operation is routed through Cmille. Therefore, on a Jmille memory operation each Cmille slice receives a data part and performs eight comparisons in parallel by means of self-checking comparators. In this way, Jmille processors are pair-wise compared along the logical ring. The comparison outcomes are OR-ed in sequence in a Cmille register, providing a single-bit comparison result at the end of the diagnostic session.

The comparison results are processed by a system-level diagnosis algorithm [93]. The local test results give a preliminary classification of the fault state of each PE. Jmille processors are ranked into three classes: *zero* (Z) units, *dual* (D) units, and *faulty* (F) units. Z units are pairs of PEs that have identical fault states: both of them are either fault-free or faulty. D units are pairs of PEs with at least one of them being faulty, but at this stage of diagnosis it is impossible to differentiate between them. F units are surely faulty. To uncover the fault states also of Z and D units it is necessary to relate processors located in different Processing Boards to each other in the Remote Equal phase.

Remote Equal. This is the *inter-cluster* diagnosis phase. For each pair of adjacent boards PB_x and PB_y two Jmille processors J_x^i and J_y^j are elected to be *witnesses* of PB_x and PB_y, respectively. The two witness Jmille nodes are compared by a pair of comparators K_{xy}^i and K_{yx}^j located in the Cmille commuters of the corresponding boards. The so obtained global comparison syndrome is further analyzed by the level diagnosis algorithm and every Processing Board is labeled as fault-free, faulty (or partly faulty with respect to certain directions), or suspected (when the diagnosis is incomplete and the state of some Processing Boards cannot be decided).

4.3 Failure model of APEmille

APEmille has a peculiar two-fold nature [91]. On one hand, it contains a computational engine with its uncommon structure of Tmille, Cmille and Jmille processors. Further on we will refer to this part of the machine as the *SIMD part.* On the other hand, it employs a network of host computers for delivering services and controlling and monitoring the SIMD part. These networked machines can be viewed as a special case of the *Multiple Instruction Multiple Data* (MIMD) class of parallel systems, hence we will refer to them as the *MIMD part* of APEmille. The two-fold nature is also reflected in the operating modes: during `run` mode the SIMD part is active and executes the user applications, while in `system` mode it is frozen and the MIMD part delivers the requested services and supervises the machine.

Reasoning about recovery techniques is possible only if a well-defined model of the failure behavior of system components is available. Given this model, the most suitable methods of error recovery can be selected, their applicability and characteristics can be verified. This section presents an component-level failure model of APEmille, which grabs specifically those aspects that need to be taken account in planning the recovery actions. The model only serves as a basis for selecting the fault tolerance mechanisms. It was developed by the author from the system specification by purely theoretical methods, therefore it needs to be refined when practical experience and measurement data will be available at a future stage of the APEmille project. The modeling methodology is inherited from Lampson [94]. The main classes of components considered are processors, memory devices, disk storage, and the communication network. The derived failure model is summarized in Table 4.1.

Based upon the Lampson model, first we account for the failure characteristics of the components in the SIMD part. Jmille and Tmille processors conform to the general processor model with the restriction that they execute only a single task. Therefore, these processors can be modeled as a deterministic *finite state automaton* (FSA). We do not consider arbitrary or Byzantine failure semantics for these processors; such failure behavior is quite improbable and hard to handle due to the special architecture. Halting, omission

Table 4.1: Failure model of APEmille components

`run` mode				
Component	**Testing method**	**Failure semantics**	**Undetected**	**Recovery**
Jmille processor	comparator	value	common-mode run-time transient	checkpoint
Tmille processor	comparator	value	common-mode	checkpoint
Cmille commuter	comparator	omission/value	common-mode	checkpoint
memory	EDAC	omission/value	n-bit ($n > 2$)	overwrite
(inter-PB) network	EDAC	performance	n-bit ($n > 2$)	retransmission
`system` mode				
Component	**Test method**	**Failure semantics**	**Undetected**	**Recovery**
(Host) processor	watchdog timer	crash (fail-stop)	value	message logging
(Host) memory	parity	omission/value	n-bit ($n > 1$)	overwrite
(Host) disk	EDAC	omission	multiple bit	duplication
(Control) network	checksum	performance	value	retransmission

and performance failure behavior implies that the processor does not drive the bus in time, leaving the bus signals in an arbitrary state. Due to the asynchronous interface the external observer sees it as a *value* failure. We assume that all of the permanent value failures are detected by the mutual comparison testing of the processors. *Common-mode* failures remain undetected after testing, but they are uncovered at a later stage by the diagnostic algorithm. Transient value failures of Jmille units, however, cannot be detected during `run` mode.

On investigating processor interactions, we assume that a Jmille processor cannot make another Jmille to fail (in the physical sense), since there is no direct connection between them. The communication is memory mapped: a special address signifies a remote access to the memory of another Jmille processor. A value failure during a memory write operation can corrupt the local or the remote Jmille memory contents, causing cumulative errors. There are three possible erroneous memory write operations respective to the failure affecting the address and/or the value of the written data. Of these three, we model the 'invalid data to right address' action as a *bad write* error, the 'valid/invalid data to wrong address' actions as the combination of a bad write (to the good address) and a spontaneous *decay* (at the wrong address) errors.

Tmille processors are also indirectly connected, so they cannot make each other to fail similarly to Jmilles. There is also a very low probability of them causing cumulative errors in Jmille processors by sending wrong instruction codes/invalid global addresses, because apart from common-mode faults these are detected by comparison testing. The memory of a Tmille processor contains only programs. By ruling out self-modifying programs we can assure that this memory will never be written. Thus, 'bad write' errors become impossible, leaving 'decays' to be the sole source of Tmille memory errors.

The Cmille commuter is the interface among the Jmille and Tmille processors, and the adjacent Processing Boards. Its value failures may produce cumulative errors similarly to Tmille processors by delivering incorrect data/instructions to Jmilles. However, transmission value failures caused by transient or permanent faults in the interconnection network are either covered by the employed *error detecting and correcting* (EDAC) code, or manifest themselves as 'bad write' or 'decay' errors of the destination memory component. Furthermore, crash failures during remote communication are supposed to be detected by the other involved Cmille commuter, which in this case raises an exception and—being unable to deliver the data—creates an omission failure. We do not account for the fault in the comparators used for testing (although they are also a part of the Cmille circuitry), this duty is left to the fault diagnosis algorithm.

The processors of the Global and Local Hosts fully comply with the general processor model of Lampson. Additional considerations can be drawn from the fact that these computers employ a multitasking, multi-user operating system. The OS (with hardware support) isolates different processes from each other, and prevents accessing the resources assigned to other processes in an unauthorized way. It provides primitives for synchronization and mutual exclusion to support the interaction of processes and sharing of resources. On this basis, we assume that a failed process will eventually raise an exception (either due to an undesired condition such as division-by-zero or a protection violation) and gets stopped, i.e., the hosts processes have a fail-stop failure semantics. During the crash, processes cannot interfere with the local state of another process, but can (and probably will) make their own resources inconsistent or corrupted. Processors are composed of a set of processes and hence behave similarly: crash on error, and after a period of time become fail-silent. The halted state can be detected using a hardware or software watchdog timer, and in the case of communication by time-out on periodically sent diagnostic < I'm alive > messages.

The memory components in both the SIMD and MIMD part employ an EDAC code to detect (and possibly correct) read errors. In the case of Jmille and Tmille memories a modified Hamming code is used, capable of detecting all double-bit (and some triple-bit) errors and correcting all single-bit errors. Corrected errors are transparent to the user and so they cannot be considered as failures. Yet, the number of corrected errors is counted in a special register, because frequent recurrence of single-bit errors might indicate an underlying permanent or intermittent fault as the cause. The host computers add a single parity bit to each word of memory, detecting all single-bit errors (and some multiple bit errors as well). Detected but uncorrectable errors raise an exception and are noted by the operating system, however, the memory contents are lost and cannot be recovered. Thus, the memory exhibits *omission* failure semantics. Undetected read errors become *value* failures.

The communication in the synchronous inter-PB network is also guarded by EDAC. The asynchronous Control Network uses a commodity network protocol. The validity of messages is controlled by various checksum schemes in several protocol layers. Lost or incorrectly delivered messages of the SIMD part are transformed into 'bad write' or 'memory decay' errors due to the memory mapped nature of the communication. In the Control Network lost messages are assumed to be detected by the delivery control mechanisms and retransmitted until it arrives successfully. Undetected value failures in the messages are disasters. Consequently, both communication networks are modeled as having a *performance* failure semantics.

Disk storage can be found only in the Global and Local Host computers. Its model is quite similar to the model of the memory components. Additionally, on the basis of the advanced manufacturing technology and the combination of EDAC and checksums transparently managed by modern hard disks, we assume that all value failures will be detected. Moreover, a large part of them will be corrected transparently by the disk controller hardware, notifying the OS to avoid the unreliable disk area by adapting its file allocation strategy. Should this assumption fail, the user can substitute a redundant disk array (RAID) in place of a single hard disk to make the assumption true. Detected but uncorrectable read errors manifest themselves as *omission* failures.

4.4 Backward error recovery in APEmille

The typical duration of a computation running on APE machines ranges from several days to several weeks. This surely remains true for the future even if APEmille is far more powerful than the previous generations, since the targeted scientific problems are also getting more and more complex. Considering the large number of components built into a typical APEmille configuration, without fault tolerance the expected *Mean Time To Failure* (MTTF) duration falls into the same time range. Thus, the user could suspect that his valuable, long-running computation has been invalidated by a system failure, and he could never be completely sure that the results of a seemingly successful computation have not been affected by an undetected hardware fault. Finally, the necessity would drive him into implementing intermediate state saving and acceptance testing routines, just as it was the practice in the case of APE100.

It is easy to see that the characteristics of the error compensation based fault tolerance approach do not coincide with the aims of the APEmille project. Therefore, the developers of APEmille decided to base their fault tolerance concept on error recovery. The requirements of system-level fault diagnosis have been taken into consideration from the initial design phases, since the formulation of the APEmille self-diagnosis theory and the planning of its hardware support started in parallel. As a result the APEmille fault diagnostic scheme is already mature, well formulated, and tested both by simulation and in practice [80, 95, 96]. On the other hand, the error recovery part of the fault tolerance concept has not yet been elaborately studied. The author was given the opportunity to contribute to the APEmille project by forming a comprehensive recovery concept for the machine [15, 17]. The problem was complicated by the fact that unlike diagnosis, the hardware support for error recovery was not present, and at the actual state of the project the extension of the completed hardware plans was unworkable. Consequently, the error recovery mechanism of APEmille had to be formed using only the existing system components [14, 16].

An erroneous system state can be transformed into a valid state in two directions. *Backward error recovery* chooses a known valid state from the system's past. This is accomplished by periodically saving the complete or partial system state to a reliable medium (called the *stable storage*) during the failure-free execution. Then, after an error occurrence the tasks are *rolled back* to the previous valid state using the information stored on the stable storage. *Forward error recovery* obtains a new valid state that is not included in the history of the system. This approach is more efficient than backward error recovery, because a history of the valid system states is not needed, and the recovery process performs valuable work while creating the new consistent system state. Unfortunately, forward error recovery is also quite application-specific, and since APEmille is a general purpose computer, rollback recovery remains as the only useful alternative.

Backward error recovery attempts to maintain the *consistency* of a computation by returning the system to a previous state upon an error occurrence. Two basic services are used in this process [97]: *state recording*, which stores global system states to a reliable medium, and *state restoration*, which assembles the stored information and recreates a consistent global restored state. These services can be either *semi-automatic* if they are executed upon the invocation of the application programmer, or *user-transparent* if the necessary actions are carried out automatically, without any user interaction. Recovery techniques can be classified in three major classes according to the characteristics of the above two services:

Checkpointing. Checkpointing saves a copy of the application state (and optionally the state of the communication channels). Upon a failure, the checkpoint can be used to restore a previous, failure-free state of a failed process. *Independent* checkpointing methods take the checkpoints process-wise. In this case, there is no guarantee that a consistent global state can be restored from the stored set of process states. The state restoration service must be able to determine which the most recent recovery line (also called as *maximum recoverable global state*) included among the set of stored local checkpoints. This makes the recovery process more complicated, and there is the possibility of *domino effect*. *Consistent* checkpointing methods avoid these disadvantages by coordinating the checkpointing actions among the processes to guarantee that the stored set of local checkpoints is always a recovery line.

Message logging. Message logging records the changes in the local process states rather than saving the complete global state of the system. State changes are called *events*. They are of two main types: *external* (such as the interactions between processes), and *internal* (like the branches of a computation). Events drive the transition of a process from one state to another, thus determining the course of computation. In a *piece-wise deterministic* system the range of external events is limited to the sending and receiving of messages. Recording of the relevant information about messages is called *logging*. The recovery protocol must recreate the exact order and content of each message transmitted before the failure occurred. Storing the contents of messages in the log speeds up the recovery process. Yet, it is not strictly necessary for a successful rollback, since the messages can be regenerated together with the other constituents of the distributed computation.

Hybrid techniques. Although it is possible to implement backward error recovery solely based on message logging, the stored log data could grow unacceptably large over a long time span. Therefore, it is customary to combine logging with checkpointing: when a global state becomes stable among the recorded checkpoints, log entries related to messages preceding the global state are no longer meaningful, thus can be discarded.

The maintenance policy of the computer is also significant in the process of selecting the

Table 4.2: Component-level fault tolerance techniques for APEmille

	Number of faulty components		
Component	One ($k = 1$)	Some ($1 < k \ll n$)	(Nearly) all ($k \approx n$)
Jmille processor	parity checkpointing	EDAC checkpointing	complete state saving
Tmille processor	checkpointing		
Cmille commuter	checkpointing		
(host) processor	sender-based logging	family-based logging	
(host) disk	available copy replication		—

most suitable approach for fault tolerance. Two different strategies can be followed: *system repair* and *reconfiguration*. Backward error recovery fits into either framework. The first strategy involves bringing the system in a consistent state and (partially) shutting it down, substituting the faulty replaceable units with spare components, restoring the valid system state, and continuing the computation. This is an obvious maintenance policy, and putting it into practice is made easier by the vast amount of experience accumulated over the years. Note, however, that system repair cannot be fully automated, its use requires continuous support by trained personnel.

System repair was more or less the policy used in the case of the APE100 series of supercomputers, in a large extent because the APE100 architecture (like the majority of SIMD machines) is not suitable for reconfiguration. APEmille includes several additions to the traditional SIMD design (like local addressing, flexible message routing, etc.) that make reconfiguration *theoretically* feasible. The following main steps constitute reconfiguration: (1) isolation (and possibly shutdown) of the failed replaceable units, (2) identification of the available resources, (3) redistribution of the tasks to fault-free processors (preferably complemented with load balancing), (4) updating the configuration databases (like routing tables, location-specific data, etc.), (5) restoring the system state preceding the error, and (6) resuming the computation. The replaceable units of the APEmille system are the Processing Boards, the Local Host Boards, and the Service Boards. The advantages of reconfiguration over system repair are its automated nature, significantly less down-time, ability to provide uninterrupted (albeit degraded) service without continuous human support. Still, reconfiguration in APEmille is so intricate and requires low-level operation system support in such an extent, that it is unlikely to be implemented.

The fault tolerance techniques proposed for the implementation of backward error recovery in the APEmille computer are summarized in Table 4.2. For each main component class of APEmille the table lists the cases when a single, multiple, or all components in the class fail, and suggests the most appropriate method accordingly. The chosen methods are explained and briefly described in the following sections (for more information on recovery techniques see [98]).

4.4.1 Recovery of APEmille processors

For the processors in the SIMD part the most suitable recovery mechanism is *checkpointing*. This choice is supported by the following arguments:

- It is difficult to implement message logging for the processing elements. The communication among the PEs is memory mapped. The PE initiating the transfer knows when a remote access takes place, but the other participant of the communication is the remote memory, and the owner processor is not notified explicitly of the read or write operation that affected his memory. For this reason, it would be complicated to realize the acknowledgement and bookkeeping schemes most message logging protocols require from the recipient processor.

- The state of the PEs changes quickly, and in a large extent. The applications are assumed to frequently access all elements of large data sets. There are many operations that affect the local and remote states; most of them are not (and cannot be) implemented as atomic transactions. Therefore, message logs would be updated often and would grow quite huge. The applications would be suspended frequently during the writing the message logs to stable storage (even if optimistic or causal message logging is used). The local and remote states are expected to have a complex interdependence, making it even more cumbersome to maintain the message logs.

- Since there is only a single process running on the PEs, logging the messages user-transparently is only possible by compiler-assisted inserting of the bookkeeping code in the user programs. This would make the executable parts of the applications grow larger.

The implementation of checkpointing is straightforward. The checkpointing or recovery procedures are invoked after the diagnostic session, when machine operation has been switched into `system` mode. Due to the transition to `system` mode, there is no ongoing computation or communication in the SIMD part, and the host computers have complete control over the memory and register-file areas of the APEmille processors. There is no need for checkpoint coordination and it is sufficient to store the state of the application without the state of the communication channels, since all data transfers have been carried out before the checkpointing procedure was invoked.

The checkpoint must contain the value of all variables and data structures that describe the actual state of the application process. Hence, it is up to the system or application programmer to decide what must be included in the set of stored status information. The most simple implementation of checkpointing saves the whole address space that belongs to the given process. Note, that this implies a huge amount of data in the case of APEmille: all of the local memories belonging the the 8×4 Jmille nodes of an APE Unit must be stored on the disk of the supervising Local Host computer. Starting from an full initial checkpoint there are methods that reduce the extent of later checkpoints, provided the state of the process changes "moderately." The techniques we propose are based on coding techniques. In the following we describe mention methods which store a reduced checkpoint generated by error detection and correction (EDAC) encoding of the data, and exploit the additional failure information provided by system-level diagnosis of the application processors.

Parity checkpointing uses $n + 1$ parity instead of a full replica to maintain the stored global state [99, 100]. The mechanism of parity checkpointing is as follows: the respective memory locations (the memory bits in the same address and position) of the n parallel processors are treated as an n-bit group. Every group of n memory bits is supplemented by a single parity bit, and the resulting parity bit array is stored on a central reliable storage medium. If a single-bit memory fault occurs, a new array parity is computed using the actual state of the local processor memory contents, and the new array parity is compared with the stored reference array parity. The difference pinpoints the location of the erroneous bit, which can be corrected by a simple inversion.

A significant drawback of parity checkpointing is that it handles only a single node failure. The problem originates in protecting a group of n memory bits with only a single parity bit. In the *EDAC checkpointing* scheme [15] the parity checkpointing scheme is combined with the multiple-bit fault-tolerant *$P + Q$ Redundancy* employed in RAID Level 6 [101]. The underlying idea is to assign an l-bit long, p-bit error detecting, q-bit error correcting error detecting and correcting code to every group of n memory bits. The generated EDAC code can be used to repair the corrupted memory contents of the failed processors, provided that the number of processors to fail always remains below p. Based on the bit-wise EDAC code the recovery algorithm can determine the *quantity* of the incorrect bits in a certain memory area, and exploit the system-level diagnostic image to find the *location* of errors. The memory contents can be recovered by simply inverting the corrupted bits. To consider the space saving represented by this approach, suppose that the system has n processors with local memories of w words, each having the size of b-bits. Using EDAC checkpointing, a code of $w \times b \times l$ bits is generated and stored in the stable storage, instead of the total state saving which requires $n \times w \times b$ bits. Clearly, the EDAC code saves a significant amount of storage space until $p \ll n/2$. However, large amount of incorrect memory bits (caused for example by an undetected transient value failure propagated to several Jmille units) can only be tolerated by storing the complete global state.

4.4.2 Recovery of Global and Local Host computers

Unlike the processors in the SIMD part, the suggested recovery mechanism for the host computers is *message logging*. The following arguments justify the preference of message logging over checkpointing in this case:

- There are two kinds of actions that a host computer must perform: serving requests and handling exceptions coming from the Processing Board, and executing *remote procedure calls* (RPC) of other hosts. These are a few, well-defined, complex operations. Provided they are implemented to be atomic and/or restartable, and their execution order/dependencies are recorded reliably, then it is possible to reset the computer (if needed) and repeat or replay the failed operation(s).

- Since the host computers are devoted to mainly supervising and monitoring tasks, the frequency of service and RPC requests is expected to be relatively low compared to the execution speed of the PEs. Therefore, the size of the message logs will not grow over a reasonable limit during a computation.

- The host computers offer a layered multitasking environment. The operating system consists of many small processes. In the case of checkpointing the state of each process should be recorded separately, while paying special attention not to violate the consistency of the global state these individual process states are supposed to form. Message logging may concentrate only on the interaction of processors via the communication interface leaving the interaction of local processes out of consideration.

- The recovery procedure can be realized as an operating system layer, consisting of a separate process (or a collection of processes). By placing the Recovery Layer low enough in the layer hierarchy, it can capture every incoming service requests and exceptions signals. Then, it can perform the necessary logging and bookkeeping actions, suspend or deliver the received messages; in other words, it has a complete control over the communication interface. In this way, message logging can be implemented user-transparently. Host computers may even utilize the idle time while the machine is in `run` mode to write the volatile message logs to stable storage.

- The number of host computers is small relative to the number of PEs. Each host is independent from the other, has its own power source and disk storage. Certain run-time transient faults causing detected errors (parity error in memory, access protection violation, etc.) may be handled at the OS level, preventing them to become failures. For these reasons, the MIMD part is assumed to have a higher availability than the SIMD part. Fault tolerance may take advantage of this, and we can employ techniques that tolerate only a limited number of failed host computers, but run more efficiently than worst-case solutions.

Message logging requires the operations working with system resources to be *atomic*, in order to keep them always in a consistent state. An action is atomic if it has both of two basic properties: it is *unitary* and *serializable*. Analogously, the notion of *atomic transactions* can be introduced. The system guarantees that after a recovery from a crash either all of the commands constituting an atomic transaction will have been successfully carried out, or none of them will have been. Additionally, atomic transactions are indivisible with respect to other transactions that may be executing concurrently; that is their execution is always equivalent to some serial order. Furthermore, there is a third condition (independent on atomicity) which denotes a transaction to be *restartable* if the transaction can be successfully repeated during crash recovery even if it has been in progress when the crash occurred.

Like checkpoints, the message log must also be retrieved reliably, thus it is also stored on stable storage. Log entries are smaller but more often updated than checkpoints, therefore the frequent access to the slow stable storage can create a performance bottleneck. There is a speed/reliability tradeoff between the two main classes of logging algorithms: the *pessimistic* and *optimistic* approach. Pessimistic methods always synchronize message delivery and logging by writing each log entry immediately to stable storage. They guarantee that the logged information is always consistent. On the other hand, optimistic methods favor performance to consistency. They assume that failures are rare, hence they try minimize the logging overhead during normal computation. They record the log entries temporarily in volatile memory, and handle the inconsistencies that may arise due to a failure during recovery. Another design decision is to choose the units that perform the logging action.

With *receiver-based* logging, the processes participating in a distributed computation log the messages upon receipt. When recovering from a failure, the failed process restarts from scratch or a previous checkpoint and replays the messages in the log. In *sender-based* logging, messages are stored at the sender process. If a process fails, the messages needed for execution replay are resent by their originator.

Sender-based logging protocols have the favorable property of tolerating a single host failure even in the optimistic setting, when the messages are stored in the volatile memory of the sender process [102]. Due to its unbeatable performance and simple implementation it is the method of choice when the failure rate is low, i.e., only a single Host computer is expected to fail at a time. If multiple simultaneous host failures may occur, an extended version of sender-based logging must be used. This approach known as the *family-based logging* (FBL) protocol [103, 104].

The underlying idea of FBL logging is that a message is partially logged by the time is is sent and it must be fully logged by the time it is *relevant*. FBL is an optimal message logging protocol in the sense that it does not send any additional messages over those needed to mask transient link failures. FBL protocols for $f > 1$ failures distribute the implementation of the recovery procedure associated with a process to a larger degree than in the single failure case. For this purpose they need to maintain and disseminate some *dependency data* about partially logged messages. The dependency data contains a mixture of sender- and receiver-based information. This means that a process must not only log the content of all the messages it sends, but it may also be required to log the messages it receives and are received by its ancestors (successors in the communication chain) up to a degree of f.

In the above discussion of rollback recovery techniques the problem of saving the checkpoints and message logs to the stable storage was mentioned many times. In Appendix E we explain the concept of a stable storage, and outline a practical approach to realizing such a device based on the disk storage units of the APEmille Global and Local Host computers.

4.5 Integration of local information diagnosis

Integration of the developed LID probabilistic fault diagnostic algorithms into the fault tolerance framework of the APEmille parallel computer is a straightforward process. The architecture of APEmille is particularly suitable for the application of centralized algorithms. The testing equipment is built in the Cmille processors in the form of self-checking comparators, and the test results (the cumulative result of the comparisons) is stored in a special register of Cmille. While the machine is in `system` mode, the Processing Boards are frozen and the Local Host computers can access the whole memory and register are of the Cmille processors. This way the host computers can reliably collect the partial syndrome data from the supervised APE Unit, and they can communicate the collected test results to the Global Host computer. Thus, the Global Host obtains the global syndrome and can execute a centralized fault diagnostic algorithm to analyze it.

Therefore, the only remaining task is to develop a comprehensive test invalidation model, which beside processors takes also the comparators used for verification into consideration. This task can be performed completely analogously to the train of thought followed in

Section 3.3. The test assignment in APEmille is illustrated by Figure 4.4. Let us examine two adjacent Jmille processors: Jmille_1^1 in the Processing Board PB_1 and Jmille_8^2 in PB_2. According to our notation these units are compared by the K_{18}^1 and K_{81}^2 comparators. Assume that these comparators comply to the basic comparison testing model introduced in Section Table 1.2. Table 4.3 describes both two unit "projection test invalidation models" derived from the comparison testing model. As it can be seen, the relationship of two processors follows the irreflexive (HK2) invalidation model, while the logical connection of processor and comparator units can be expressed by the symmetric (PMC) invalidation model.

Table 4.3: Test invalidation including interconnection links

Jmille_1^1	Jmille_8^2	**Result**
0	0	0
0	1	1
1	0	1
1	1	1

(a) The relationship of two Jmille processors

K_{18}^1	Jmille_1^1	**Result**
0	0	0
0	1	1
1	0	X
1	1	X

(b) The relationship of comparator and Jmille

The partial testing models can be unified in the comprehensive test invalidation model displayed in Figure 4.5. The ellipse around the three logical test connections between Jmille_1^1 and Jmille_8^2 processors and the K_{18}^1 comparator device indicates that these test invalidation edges correspond to the single t_{18} physical test connection. The same problem arises with this combined test invalidation model that was already studied in Section 3.3. The interpretation of an $a_{18} = 0$ test result is that "either the Jmille_1^1 and Jmille_8^2 processors are both surely fault-free, or the K_{18}^1 comparator is faulty", the $a_{12} = 1$ test implies, that "there is at least one faulty device among the Jmille_1^1, Jmille_8^2, and K_{18}^1 units." The inferences that originate from the comprehensive model can again only be described by an AND-OR graph representation, so the inference propagation and fault classification tasks need to be fulfilled by the A0* algorithm or an equivalent method. A better option is to use constraint-based diagnosis, which has even an stronger description power than the AND-OR graph representation.

Yet, this situation is much simpler when self-checking comparators are used. If the self-checking logic built in the comparator circuits provides sufficiently high error coverage, the comparators can be designed to always give a failed test result in case of an internal error. Consequently, the relationship of processors and comparators alike will obey the HK2 invalidation model, that is the comparators can be modeled as if they were part of the adjacent Jmille unit. This modification results a conventional testing graph of homogeneous irreflexive test invalidation without explicit comparators in the model. Therefore, the methodology of local information diagnosis becomes applicable to the system without any adaptation. One may argue that this simplification may cause that a given Jmille processor is incorrectly accused of being faulty instead of an adjacent comparator. Note however, that the aim of the diagnostic process in APEmille is to identify *faulty Processing Boards*

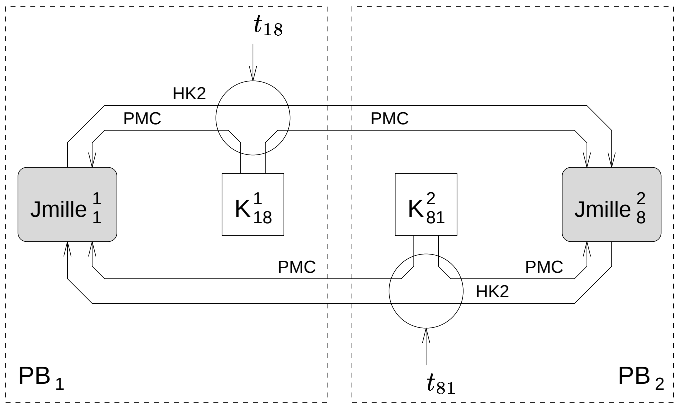


Figure 4.5: Comprehensive test invalidation model of the Parsytec GCel

and not an accurate fault localization inside the PBs. The proposed modification of the comprehensive testing model does not affect fault classification on the Processing Board level.

4.6 Constraint-based system-level fault diagnosis

So far this document argued the advantages of the system-level diagnostic approach and the methodology of local information diagnosis. However, Section 3.3 and Section 4.5 also pointed out that for certain situations the generalized test invalidation model is not flexible enough. The applicability of the presented methods can be restrained by the binary fault model and the binary test outcomes. The inability of handling multiple units per test or dealing with alternative consequences of implications are also limiting the possible application area. The author participated in the development of a new diagnostic approach: the application of the constraint satisfaction methods to solve diagnostic problems [50].

The novel modeling technique for system-level fault diagnosis in massive parallel multiprocessors was presented by Pataricza et al. [49], further enhancing the theoretical basis of the Selényi algorithm. The authors re-formulated the problem of syndrome decoding to a constraint satisfaction problem (CSP). The CSP-based approach is able to handle detailed and heterogeneous functional fault models on a level similar to the Russel-Kime model [36, 37]. Multiple-valued logic is used to describe system components having multiple fault modes. The granularity of the model can be adjusted to the diagnostic resolution of the target without altering the methodology.

Constraint satisfaction problems (CSP) deal with the derivation of single or all consistent solutions of large-scale relation systems. Formally a CSP is a (X, D, C) tuple, where $X = X_1, \ldots, X_n$ is a set of variables defined over the set $D = D_1, \ldots, D_n$ domains, and $C = C_1, \ldots, C_k$ is a set of constraints. Constraints are relations between the variables. A solution of a CSP is a vector $[x_1, \ldots, x_n]$ of values that satisfies all the constraints.

The ultimate goal of syndrome decoding and CSP is very similar: the algorithm must classify the fault state of system components in a consistent way that conforms to the given diagnostic model, test invalidation rules, and the actual test outcomes (syndrome elements).

These restrictions can be represented as binary relations between the fault state of the tester and the tested units. The use of relations instead of logical functions supports the handling of diagnostic uncertainty appearing in some test invalidation models. A diagnostic problem can be very easily transformed to a constraint satisfaction problem. The variables of the CSP represent the fault states of the system components. The constraints correspond to the restrictions derived from the test invalidation model and the current syndrome elements, thus the test invalidation rules determine a set of candidate relations, from which the actually used one has to be selected, when knowing the actual test outcome.

Syndrome decoding is driven by implication rules, represented by constraints. They originate from the system structure, the test invalidation model and the actual test outcomes. All of the constraints are binary over the fault state of the tester and the tested component, to achieve a greater simplicity: the test results (syndrome bits) are eliminated from them as variables after receiving the test outcome. The generated constraint network is then preprocessed. The applied simple consistency algorithms greatly reduce the amount of computation required to solve the CSP problem. After the preprocessing phase the CSP solver is executed. When all of the constraints are satisfied, the domains of the variables corresponding to the system units contain the possible classifications for the units, in other words, the transitive closure. If not all the components have a sure classification in the transitive closure, the remaining ones are classified using further restrictions on the fault model (limitation on maximal number of faults, exclusion of certain faults).

The diagnostic algorithm has the ability to create diagnosis "on-the-fly", using only partial syndrome information. In this case the diagnostic part is invoked several times at the arrival of partial test results. Implications from new syndrome elements are added to this dynamic constraint network during each call and a path-consistency based preprocessing of the network is performed, successively restricting the solution domain. As a result, the algorithm will extract all of the useful diagnostic information available in the partial syndrome, and the diagnosis is improved at the arrival of each new test result.

If one-pass diagnosis is required, simply a static binary CSP is produced after performing all tests. However, in the case of on-the-fly diagnosis, newly generated test results must be processed immediately. Only a few syndrome elements are present at the beginning, so the complete set of relations cannot be created at once. Every incoming test result adds new relations to this set, while the previously constructed ones remain still valid. Thus the solution space is gradually reduced, in other words a monotone dynamic CSP is produced.

We did not mention so far the main disadvantage of constraint-based fault diagnosis: its time complexity. Decreasing the complexity of the CSP solution be achieved by preprocessing the constraint network. The so-called *consistency algorithms* exclude locally inconsistent value combinations from the domains of variables, since these values surely cannot appear in a globally consistent solution. Unfortunately these methods work generally only on binary CSPs, which can evaluate independently every variable. Despite the employed preprocessing mechanisms the time complexity of constraint-based diagnosis remains significantly higher than that of local information based probabilistic diagnosis, therefore in time critical applications where the modeling power of traditional generalized test invalidation is sufficient, LID algorithms remain as the only option.

Chapter 5

Conclusions and future work

The work presented in this document encompasses different, but quite interdependent areas of multiprocessor fault tolerance. Contributions are related to three main topics: *centralized fault diagnosis*, *distributed fault diagnosis*, and *backward error recovery* of parallel computing systems. The achieved results are mainly of theoretical nature, as most of the contributions is related to general-purpose fault diagnostic algorithms. The developed methods and procedures are not applicable only to the illustrative real-world multiprocessor systems included in the document as examples, but they can be employed in any system conforming to the requirements of system-level diagnosis. Despite the theoretical and general features, much effort was put into creating practically relevant techniques which have appealing properties to users and system engineers. For this purpose, *efficiency* and *low resource usage* were the two main guidelines of the research. The presented concepts represent a kind of compromise: in the literature many complex deterministic approaches can be found providing reliable diagnosis (although only in strictly restricted situations) or simple probabilistic approaches exist offering high execution speed (but producing only mediocre diagnostic results). Our work attempts to fill the gap between these approaches by combining their advantages.

The first part of the thesis deals with centralized probabilistic fault diagnosis of multiprocessor systems. We introduce a novel concept called *local information diagnosis.* The LID concept bases the fault classification of each system unit on diagnostic information gathered from the limited local environment of the unit. By exploiting the reduction of processed data it was possible to decrease the typical $O(n^3)$ time complexity of deterministic algorithms to $O(n^2)$ and furthermore even to linear complexity $O(n)$ in the function of system size. Similar reduction was possible in the space complexity of the working memory used during data processing from $O(n^2)$ to a constant additionally required space $O(1)$.

These complexity properties are competitive with the existing probabilistic methods, yet, the developed algorithms use significantly more of the diagnostic information contained in the syndrome. Due to their iterative mechanism the algorithms provide the possibility for a performance/diagnostic accuracy trade-off: the more iterations they are run, the more accurate the results will be, but of course the execution time increases accordingly as well. We also demonstrated that in general fault situations the amount of information extracted from the syndrome relatively soon reaches a saturation point (complying to the idea of the LID concept). Further iterations do not significantly improve the diagnostic solution.

The developed diagnostic methods utilize the *generalized model* of test invalidation. This makes the methods easily adaptable to the non-symmetrical invalidation situation and even to heterogeneous systems. An important advantage of the underlying generalized test invalidation model is the existence of *sure* fault classifications derived from detected contradictions in the implication set. Surely classified units are reliably identified despite the probabilistic approach and this capability of the developed algorithms is unique among the probabilistic diagnostic techniques existing in the literature. The employed implication representation and processing mechanism make it possible to divide the diagnostic process into two separate phases at this point.

In the first phase one-step implications are extracted from the syndrome and propagated in a limited length to create a a partially transitively closed implication set. This phase terminates with the diagnosis of surely classable units. In the second phase those units that cannot be surely identified are classified by a heuristic method. The thesis gives several alternative solutions for both phases. These different solutions can be appropriately combined according to the user requirements. Separating the inference extraction and fault classification phases has the additional advantage that new heuristics tailored to the particular application environment can be easily introduced in the diagnostic system without having to re-implement the implication-handling procedures.

The second part of the thesis introduces a new distributed diagnostic algorithm which was developed with the reduction of the diagnostic message volume in mind. As a novel feature, the algorithm divides the diagnostic activities into two stages, in order to schedule the higher communication load during the initialization of the system when the performance requirements are not strict. During the *initial* stage a system-wide local diagnostic image is generated in each processing element. In the later *working* stage this basic diagnosis is updated incrementally by taking the consequences of newly occurred diagnostic events (failure, repair) into account. This diagnostic strategy also supports the *event-oriented* dissemination of the local test result information, therefore new reporting messages are only transmitted when they really represent relevant information.

The faulty units prevent the reliable message transport and may separate different parts of the system. The scope of the produced diagnostic image extends to all fault-free processors located in the same strongly connected component and the faulty units forming the isolating barrier. The algorithm uses a *barrier detection* procedure to speed up the determination of diagnosability boundaries. The faults in the interconnection links are also included in the diagnostic model. Links are classified using a functional failure model without the use of dedicated link tests.

The diagnostic procedures are supplemented by two communication protocols, tailored to the characteristics of the two stages of diagnosis. For the initial stage a synchronous protocol is given, which is designed to perform high-intensity, low-latency message transfer until every test result report sent at each processor reaches its destination. The protocol employs a simple routing mechanism which does not attempt to avoid redundant messages, since the transmission of these messages does not create additional delay due to the rigid scheduling of communication actions. On the contrary, additional "dummy" messages and a separate termination phase must be employed in order to keep up the smooth operation of the synchronous protocol. For the working phase an asynchronous protocol was developed, which provides nearly optimal message volume even in the presence of faulty units in the system. The routing mechanism of the protocol is a combination of an optimal message distribution scheme for faultless networks, and supplemental forwarding rules to create additional messages which compensate for the deficiencies of the distribution scheme in a faulty environment. The correctness of the presented protocols was confirmed by practical implementation.

In the third part of the thesis a backward error recovery scheme is developed for the APEmille parallel computer. The created recovery scheme unifies the various error recovery mechanisms selected for the different components. The selection of recovery mechanisms was oriented by a component-level failure semantics model, which was (in the absence of experimental reliability data) obtained from the system specification by theoretical methods. As part of the recovery scheme we also present a prospective implementation of a stable storage, which fits into the architecture of the APEmille computer and does not require the inclusion of external hardware components.

The thesis also studies the possibility of extending the local information diagnosis formalism to practical centralized and distributed systems. Dedicated test invalidation modes are derived adjusted to the particulars of the examined machines. The shortcomings of the traditional system-level modeling technique are pointed out, and the alternative solutions are proposed.

Naturally, the work described in this document cannot be considered to be closed. The presented achievements still leave opportunities for refinement and further research. In the case of the developed local information based probabilistic fault diagnostic algorithms the promising directions for improvement are related to increasing the diagnostic accuracy. Several alternatives are open for investigation. First of all, the diagnostic performance of the presented fault classification heuristics and the scalar weight function can be studied by detailed analysis of the diagnostic mistakes. The results of this analysis can be used to refine the heuristics and to more carefully set up the weight function in order to increase the diagnostic accuracy in non-symmetric and heterogeneous testing situations. The analysis may also yield new heuristic methods of approximating the likelihood of fault hypotheses. Another option to achieve a more favorable diagnostic performance is to invent a way of "converting" some percentage of the diagnostic inaccuracy from malign misdiagnosis to benign misdiagnosis, or even eliminate malign diagnostic mistakes completely.

The complementation of the distributed diagnostic algorithm can be done mainly with respect of the used communication protocols, as the extension of the diagnostic model to the constraint-based methodology was already accomplished [50] (with the participation of the author). The use of the presented protocols is restricted to the two-dimensional mesh topologies due to the mechanisms of the built-in routing and barrier detection procedures. These procedures need to be extended to other system topologies as well, in order to extend the applicability of the developed distributed algorithm for other systems than the Parsytec GCel machine.

In the case of the proposed error recovery scheme for APEmille, the remaining work is mainly of practical nature. The introduced scheme is only a high-level framework, the detailed description of the component-based recovery mechanisms can be developed in compliance with the presented guidelines. It is also necessary to experimentally verify the assumptions behind this scheme, since these assumptions are based on theoretical considerations and their validity is not yet confirmed. The formulation of the detailed specification of recovery procedures can be followed by the practical implementation of the developed error recovery scheme.

Appendix A

Definitions and notation

This section describes the main concepts of system-level diagnosis and their notation, adopted for use throughout this document.

A potential fault in the system is denoted by f_i, and the set of all potential faults under consideration by $\mathcal{F} = \{f_1, \ldots, f_n\}$. Since multiple faults may occur, the subset of $\Phi^j \subseteq \mathcal{F}$ faults present in the system is referred to as a *fault pattern.* The set of considered fault patterns is $\Phi = \{\Phi^0, \ldots, \Phi^{m-1}\}$. Here, Φ^0 symbolizes the fault-free case: $\Phi^0 = \emptyset$. The number of faults in a fault pattern is $\phi^j = |\Phi^j|$. The set of faults actually present in the system is denoted by F, and the number of actually faulty units is $\phi = |F|$.

Tests have a terminology and notation similar to faults. A test is denoted by t_i, and the set of all tests devised for the system by $\mathcal{T} = t_1, \ldots, t_p$. A *test pattern* Θ^j contains the tests that failed on the application of the test set $\mathcal{T}$. The set of all test patterns that are expected to occur is $\Theta = \Theta^0, \ldots, \Theta^{q-1}$. Analogously, the $\Theta^0 = \emptyset$ test pattern represents the situation when all of the tests passed.

A parallel system is composed of a set $U = \{u_1, \ldots, u_n\}$ of $n = |U|$ units (processing elements, PEs). In system-level fault diagnosis each unit u_i is capable to execute tests on a particular subset of other units in U. A given test performed by u_i on u_j is denoted by t_{ij} and is called a *test connection.* The predefined set of test connections is the T *testing assignment.* Note, that there is a one-to-one correspondence between the sets T and $\mathcal{T}$. The testing assignment can be drawn as a directed graph $G_T = (V, E_T)$, called the *testing graph*, where a vertex $v_i \in V$ corresponds to the $u_i \in U$ unit, and the directed edge $(v_i, v_j) \in E_T$ represents the test connection $t_{ij} \in T$ between units u_i and u_j. Multiple test connections between the same two units are not allowed, therefore any two vertices are connected by at most one edge in a certain direction in E_T. A directed edge sequence $[v_i, \ldots, v_j]$ in G_T is called a *path*, its length $k = |\,[v_i, \ldots, v_j]\,|$ is the number of its edges.

With each u_i unit the following sets can be associated:

$$\begin{aligned}
\Gamma(u_i) &= \{u_j : t_{ij} \in T\} \\
\Gamma^{-1}(u_i) &= \{u_j : t_{ji} \in T\} \\
\Gamma_k(u_i) &= \{u_j : |\,[v_i, \ldots, v_j]\,| \leq k \mathrm{in} G_T\} \\
\Gamma_k^{-1}(u_i) &= \{u_j : |\,[v_j, \ldots, v_i]\,| \leq k \mathrm{in} G_T\}
\end{aligned}$$

The $\Gamma(u_i)$ is the set units *tested by* u_i (devices under test, DUT). The $\Gamma^{-1}(u_i)$ is the set units *testing* u_i (testers). These notion can be extended to a broader local environment of diameter k. The set of units, that reach u_i via directed paths of k or less length are called the $\Gamma_k^{-1}(u_i)$ set of k-testers. Units reachable from u_i via directed paths of k or less length are the $\Gamma_k(u_i)$ set of k-DUT units.

Similarly, the following sets can be defined for any set of units $U_m \subseteq U$:

$$\begin{aligned}
\Gamma(U_m) &= \bigcup_{u_i \in U_m} \Gamma(u_i) - U_m \\
\Gamma^{-1}(U_m) &= \bigcup_{u_i \in U_m} \Gamma^{-1}(u_i) - U_m
\end{aligned}$$

A unit is an *internal unit* of a set U_m if it belongs to that set, otherwise it is an *external unit* to the set U_m. Then, $\Gamma(U_m)$ and $\Gamma^{-1}(U_m)$ are the set of external units to the set U_m, which are tested by and testing the units internal to the set U_m.

In most multiprocessor systems cooperation among the parallel tasks is realized using messages transferred over the built-in inter-process(or) communication facilities. A given message transfer capability from unit u_i to u_j is denoted by c_{ij} and is called a *communication link.* The set of implemented communication links is the C *communication network.* The communication network can also be represented in graph form. Since usually communication links are bidirectional, the *communication graph* is in most cases an undirected graph $G_C = (V_C, E_C)$, where a vertex $v_i \in V_T$ corresponds to the $u_i \in U$ unit, and the edge $(v_i, v_j) \in E_C$ represents the communication link $c_{ij} \in C$ between units u_i and u_j. With each u_i unit the following sets can be associated:

$$\begin{aligned}
\mathrm{N}(u_i) &= \{u_j : c_{ij} \in C\} \\
\mathrm{N}_k(u_i) &= \{u_j : |\,[v_i, \ldots, v_j]\,| \leq k \in G_C\}
\end{aligned}$$

The units adjacent to u_i in the communication graph constitute the $\mathrm{N}(u_i)$ set of *neighbors.* The units reachable from u_i via undirected paths of $k+1$ or less length are the $\mathrm{N}_k(u_i)$ set of k-neighbors. In those parallel systems, where tests are conducted only via the built-in communication network, the testing graph is a subgraph of the communication graph: $G_T \subseteq G_C$. In such systems the neighbor units are either testers of, or are tested by u_i, therefore $\mathrm{N}(u_i) = \Gamma(u_i) \cup \Gamma^{-1}(u_i)$. This is also valid in the case of k-neighbors: $\mathrm{N}_k(u_i) = \Gamma_k(u_i) \cup \Gamma_k^{-1}(u_i)$. The cardinality of neighbor sets are denoted by $\nu(u_i) = |\mathrm{N}(u_i)|$ and $\nu_k(u_i) = |\mathrm{N}_k(u_i)|$.

A successfully executed t_{ij} test produces an a_{ij} test result. Test results are binary: they either *pass* or *fail.* Passed test results have a value of 0, while failed test results have a value of 1. The A^i set of test results obtained by one execution of all test connections is a *syndrome.* The set of all possible syndromes is $A = A^0, \ldots, A^{q-1}$. Note, that there is a one-to-one correspondence between the sets A and Θ. Edges of the G_T testing graph can be labeled by the corresponding elements of the syndrome to create an *annotated testing graph.*

The diagnostic algorithm D is a projection between syndromes and fault patterns: $A \mapsto \Phi$. In other words, the input of the algorithm is the actual syndrome A^i, and as the result of its execution it generates the set of diagnosed fault-free U^0 and diagnosed faulty units $U^1 \in \Phi$. In the case of *diagnostic inaccuracy*, the U^1 diagnosed fault set and the physically present F actual fault set are not the same. The differences can fall into two categories. When the fault state of some processors could not be identified, that is $U^0 \cup U^1 \subset U$, where $U^1 \subset F$, then the diagnostic image is said to be *incomplete*. If the U^0 diagnosed fault-free units and F fault sets or the U^1 diagnosed faulty units and $U - F$ reliable units have common members: $U^0 \cap F \neq \emptyset \vee U^1 \cap (U - F) \neq \emptyset$, then the diagnostic image contains *misdiagnosed* units and is *incorrect*. Misdiagnosis can be of two types, either a fault-free unit is diagnosed as faulty $u_i \in U^1 \wedge f_i \notin F$, or a faulty unit is diagnosed as fault-free $u_j \in U^0 \wedge f_i \in F$. From the technical viewpoint treating a fault-free unit mistakenly as a faulty unit reduces the amount of resources but does not threaten system availability, therefore it is called a *benign misdiagnosis*. Missing the detection of a faulty unit and treating it as fault-free is hazardous to system integrity and so it is called a *malign misdiagnosis*.

Appendix B

Probability models

Probabilistic methods differ in the probabilistic parameters used and the probability model employed to define the likelihood of the different syndromes. A probability model is characterized by defining the probability space, which is a triple (Ω, Θ, P). In this triple Ω is the *sample space*, Θ is the *event space*, and P is a probability measure.

The most often used probability model is called the *common* probability model. In this model, the sample space Ω is defined to be the set of all possible syndrome and fault set pairs in a given testing graph $G = (U, E)$. Formally,

$$\Omega_G = \{\, (S, F) : F \subseteq U \text{ and } S \text{ is a function from } E \text{ to } \{0, 1\} \,\}$$

The event space Θ_G consists of the set of all possible subsets of Ω_G. The probability measure P is determined using the probability parameters describing the behavior of fault-free and faulty units, assuming that all the units in F are faulty and all the units in $U - F$ are fault-free. In the case of the 0-information tester model, every possible syndrome and fault set pairs can have a nonzero probability value. However, for the partial tester or complete testing models certain syndrome and fault set pairs will have zero probability. For example, in these testing models a situation of $u_i, u_j \in U - F$ and $S(t_{ij}) \equiv a_{ij} = 1$ can never occur, so the probability of this syndrome and fault set pair is 0.

Blough introduced a more general probability model. He modeled the actions of faulty units by assuming that they behave in any arbitrary manner, even the most detrimental to the diagnostic algorithm. The sample space Ω_G is the same in the Blough model as for the common model, but the set of basic events is different. Since test outcomes by faulty units are random values, in general no probability value can be assigned to a syndrome, fault set pair. Therefore the basic events of the model consist of syndrome, fault set pairs that have the same fault set and whose syndromes are identical except the test results of faulty units. Thus, a syndrome, fault set pair (S', F') is contained in a basic event B is defined as

$$B = \{\, (S, F) : F = F' \text{ and } \forall t_{ij} \in T \text{ with } u_i \in U - F, S(t_{ij}) = S'(t_{ij}) \,\}$$

Let B_{set} denote the set of basic events, then the event space Θ_G is composed of all subsets of B_{set}. The probability of an algorithm producing a correct and complete diagnosis using the model can be calculated as the minimum of the probabilities assigned to the syndrome, fault set pairs in the basic event B, for which the set of faulty units F is the set of units classified as faulty and B contains the actual syndrome observed.

Lee and Shin introduced another generalization of the common probability model. Since most probabilistic algorithms use less than the global syndrome information for diagnosis, the authors assumed a distributed self-diagnosis method in which each unit declares itself faulty-free or faulty. For this, units use only a limited form of the global syndrome information, referred to as *partial* syndrome. Since the information analyzed during diagnosis varies for every unit, each of them uses a different probability space. For a given unit u_i, let A_i represent the partial syndrome used in the diagnosis of u_i, and A be the set of all such partial syndromes. Then, the sample space for unit u_i is defined as

$$\Omega_i = \{\, (A_i, f(u_i)\,) : A_i \in A, f(u_i) \in \{f_i^0, f_i^1\}\,\}$$

The event space Θ_i is the set of all possible subsets of Γ_i. The definition of of the probability measure P_i depends on the partial syndrome information used.

All of the probability models presented have advantages and disadvantages. The common probability model provides exact calculation of the a posteriori fault probability of each fault set given a syndrome. In this model an algorithm that produces the most probable diagnosis can be formulated. Such an algorithm is also proven to be optimal in diagnostic accuracy [105]. The Lee and Shin model makes it possible to generate the most probable diagnosis for a certain partial syndrome, assuring locally optimal algorithms (i.e. these algorithms are optimal in diagnostic accuracy among all methods analyzing the same partial syndrome information). While the problem of optimal diagnosis in the Lee and Shin model can be solved in polynomial time, the same problem has exponential computational complexity in the common model. Also a serious drawback of both models is that the probability parameters describing the operation of units most be known or estimated before any probability analysis can be done.

Here lies the main advantage of the Blough probability model: the behavior of faulty units do not need to be parameterized before its application. On the other side, this causes many diagnosis algorithms that generate different diagnostic output to be evaluated as being equal under the model. Thus, the Blough model can not distinguish between two methods, one of which may perform significantly better in terms of diagnostic accuracy. Consequently, exact analysis and optimal diagnosis is not possible using this model, but the asymptotic or upper-bound characteristics of the algorithms can be estimated.

Appendix C

Simulation of the LID algorithms

The analysis of probabilistic diagnostic algorithms is not straightforward. The traditional verification method, a *correctness proof*, is of no use in this case, since probabilistic algorithms are inherently inaccurate. There are other analytical approaches to provide some insight into the operation of such algorithms, Section C.1.2 reviews some of these. While such analysis is feasible for a homogeneous system with units having the same, single test invalidation model, the evaluation in a heterogeneous testing situation is much more complex. Theoretical analysis of this kind is therefore exceeds the scope of this document. Still, it is necessary to have some information about the performance and applicability of the presented algorithms. Our method of choice for providing this information is *simulation* and *measurement.*

C.1 Measured characteristics

This section introduces the decisions behind the selection of activities in the measurement process by aswering the three main questions: *what to measure? in the quantity of which parameter(s) are the measurements to be varied?* and *how to measure the chosen values?* The answers to the above questions are given from the practical aspect of the user, who is interested in application-related information. The theoretical aspect is presented by reviewing existing attempts to analyze certain probabilistic diagnostic algorithms and discussing the relationship of these algorithms to our work.

Before one could start experimenting with a diagnostic algorithm, he must (1) identify the most important *characteristics* of the algorithm to be examined, (2) determine the *variable parameters* of the algorithm to which the measurement results can be related, and (3) invent a method for measuring the chosen characteristics. This section outlines this process for the algorithms subject to our document.

C.1.1 Practical aspects

The first, and most important, question to answer is: what to measure? From the practical viewpoint of the user two main characteristics can be identified: *diagnostic accuracy* and *algorithm complexity.* Diagnostic accuracy captures the user's confidence in the diagnostic subsystem. It measures the number of incorrectly classified units, in our model this means

Table C.1: Measured characteristics

Class	**Characteristic**	**Form**
Diagnostic accuracy	Number of misdiagnosed units	incorrect/all ratio
	Type of misdiagnosis	benign, malign
	Arrangement of misdiagnosed units	isolated, accessible
Algorithm complexity	Time complexity	processing time
	Space complexity	memory allocation

fault-free units to be diagnosed to be faulty, and vice versa. The number of misdiagnosed units in itself is not really adequate to characterize the diagnostic performance of a probabilistic algorithm. Better measures are the number of misdiagnosed units *in the ratio of the system size* or rather *in the ratio of the number of faulty units.*

Perhaps an even more important attribute is the *type* of the diagnostic inaccuracy. When the algorithm mistakes a fault-free processor for a faulty one, then it reduces the amount of resources available for a further reconfiguration, but does not harm the system integrity. Since in this case the diagnostic error occurs for the benefit of availability, we call this type of inaccuracy *benign misdiagnosis.* On the other hand, mistaking a faulty processor for a fault-free one does not decrease the amount of resources, but leaves unreliable components be part of the system. This type of inaccuracy undermines system integrity, therefore we call it *malign misdiagnosis.*

Also, the *arrangement* of incorrectly classified units can be of significance. In most massively parallel systems the processing units communicate with each other via direct, processor-to-processor links, driven by a communication controller chip built-in the PE (e.g., transputers of the Parsytec GCel, see Section 3.1), or included as a separate component (e.g., Cmille unit of the APEmille, see Section 4.1). Consequently, the failure of the processor/communication controller usually means the loss of the corresponding communication links. In the event of several processor failures, certain areas of the processing network may become completely *isolated* and unavailable for cooperation with the rest of the system. In such situations the diagnostic mistakes within the inaccessible region are insignificant; until the faulty units blocking the communication are not repaired these processing elements are useless for the system anyway.

The *complexity* of any algorithm largely determines its applicability. This is especially true for diagnostic algorithms, as they are background tasks, and the execution of diagnosis should have as small impact on system performance as possible. At the same time, fast diagnosis is desirable to decrease error propagation. Complexity comprises two distinct measures: *time complexity* evaluates the processing time and load required to run the algorithm, while *space complexity* quantifies the amount of memory used in the diagnostic process. Space complexity may as well affect time complexity: using a larger amount of memory than the available physical memory requires virtual memory management, which will generate many additional processor and disk storage operations.

Table C.1 summarizes the measured characteristics of the developed algorithms.

Knowing the measured values (dependent variables) one must decide the set of *vary-*

ing parameters (independent variables) of the measurements. These belong to three main categories: the attributes of the *fault-free system* and of the *fault injection mechanism*, as well as the variable parameters of the algorithms. In the case of diagnostic accuracy, the system *size*, *topology* and the *test invalidation model* of the units are the essential system attributes. Although the designed algorithms are topology independent, we restricted our investigations to the most commonly used regular system topologies. Only the testing arrangement is taken into account, because we do not simulate the application-related message traffic and the algorithms do not perform diagnosis of the interconnection links. One of the most important topological properties is the *connectivity*, which is composed of two quantities: *fan-in* and *fan-out*. For testing graphs these quantities correspond to the number of testers of, and the number of units tested by a certain processor.

Size and topology are closely interrelated. Due to the regular nature of the considered system topologies certain unit amounts are not valid (e.g., prime numbers for a two-dimensional grid topology). Also, the number of units is limited by the resource requirements and the duration of the simulation. A significant feature of the topology if it has borders; the two-dimensional grid and torus are illustrative examples for this kind of difference. (More precisely, bordered topologies are not really regular, but nevertheless, they are also quite commonly used in parallel computers.) Large, non-bordered topologies can be regarded as being "infinitely" large, provided that the number of faults is relatively small.

A notable advantage of the presented algorithms that they are also invalidation independent, or in other words, the test invalidation of the units is also a variable parameter. Obviously, it does not mean that the algorithms perform equally well for each test invalidation model. The simplest counter-example is the $\mathrm{T_T}$ (consistent liar) model, which requires a simple but specific diagnostic strategy, and therefore most of the general-purpose algorithms produce poor results for this model. Beyond measuring the algorithms in different homogeneous test invalidation situations, it is just as well interesting to examine heterogeneous setups, where the invalidation varies from unit to unit.

Fault injection can be characterized by the *size* and *arrangement* of the generated fault patterns. Similarly to misdiagnosis, the number of faulty units is a less relevant measure than the ratio of the fault pattern size to the whole system size. The main attribute of the fault pattern arrangement was already mentioned: it is important to know whether the faulty units *isolate* a part of the system or not. The probability of isolated areas can be increased when the fault injection mechanism attempts to place faulty units close to each other. In this case we speak of one or more *fault group(s)*.

Certain algorithms, like the LID algorithms presented in this document, also have variable parameters. In our case, we can influence the amount of diagnostic information processed by modifying the local area from which inferences are collected. This is determined by the number of *iteration steps*. Our general expectation is that the more iteration steps the algorithms perform the better the diagnostic results will be.
Table C.2 summarizes the varying parameters of the measurement experiments.

So long we regarded the evaluation of the diagnostic accuracy. As it can be seen in Table C.2, certain parameters are discarded from the measurement of the algorithm complexity. Such is the test invalidation model. Invalidation affects algorithm performance via

Table C.2: Varying parameters

Measured	**In the function of ...**		
characteristic	**Class**	**Parameter**	**Form**
Diagnostic accuracy	System attributes	Size	number of units, border
		Topology	(test) connections
		Test invalidation	heterogeneity
	Fault injection	Number of faulty units	all/faulty ratio
		Arrangement of faulty units	scattered, group
	Algorithm parameters	Number of iterations	iteration steps
Algorithm complexity	System attributes	Size	number of units, border
		Topology	(testing) connections
	Fault injection	Number of faulty units	all/faulty ratio
	Algorithm parameters	Number of iterations	iteration steps

the number of enabled inference rules. In our simplified diagnostic model the maximum set of inference rules is relatively small, therefore there is little variance in the number of available implications from model to model. In the current implementation of the algorithms this means that invalidation has little impact on the run-time and absolutely no effect on the memory requirements. Another parameter left out of consideration is the arrangement of faulty units. The algorithms work with implications, which depend much more on the number of faults than their placement in the system, therefore the influence of fault pattern topology on the algorithm performance is insignificant.

The last open question is: how to measure the chosen values? Most of the selected attributes are quantitative, thus easy to measure. These are automatically recorded by the simulation environment. However, the control or observation of the following attributes and parameters raises some problems:

Arrangement of misdiagnosed units. Although the number and type of the diagnostic mistakes is recorded by the measurement tool, this information does not relieve anything about their arrangement. It is a complex task to automatically detect when a misdiagnosed unit is located in an isolated region, thus this feature is not built-in the simulation environment. Instead, the user can define appropriate fault patterns and can use the *selective statistics* option to separately inspect specific areas of the system.

Arrangement of faulty units. The problem of fault arrangement is similar: the measurement tool provides injection mechanisms to create fault groups, but it cannot guarantee that the resulting fault pattern really isolates a part of the system. Moreover, this approach is inadequate to examine situations where the isolated part is mainly composed of fault-free units. The problem can be solved again by manually defining fault patterns and selectively inspect the interesting areas.

Time complexity. When measuring the performance of the algorithm, several circumstances must be taken into account. The time complexity cannot be evaluated by

Table C.3: Measurement methods

Measured characteristic	**How to measure?**
Number of misdiagnosed units	Count differences of the real and diagnosed state
Type of misdiagnosis	(Selectively) count benign/malign misdiagnosis
Arrangement of misdiagnosed units	Custom fault pattern, selective statistics
Time complexity	Single task operating system, debugging tool
Space complexity	Debugging tool
Varying parameter	**How to set?**
System size	Simulate n units
System topology	Simulate chosen (testing) arrangement
Test invalidation	Constrain test results to comply with invalidation
Number of faults	Inject ϕ faults (drawing or probabilistic method)
Fault arrangement	Evenly distributed, grouped, or custom fault sets
Number of iterations	Run the algorithm in k iterations

the simulation environment, since the code added for this purpose would distort the results. An appropriate profiling tool must be used for the purpose, which is able to distinguish the time spent in the simulation environment from the time required for executing the algorithm. Even the initial phase (allocating and setting up the data structures) and the actual diagnostic phase of the algorithms must be handled separately. And the measurements must take place in a single-tasking, single-user operating system, on order to prevent possible background tasks to falsify the results.

Space complexity. The memory requirements of the implemented algorithms are both hard to measure (it is difficult to separate the memory used by the simulator from the memory used by the simulated algorithm), and the measurement results largely depend on the compiler/operating system used to create the executable code. Anyway, the theoretical considerations give good enough estimations of the real memory requirements. Consequently, we decided to skip the verification of space complexity.

Table C.3 summarizes the measurement and parameter setting methods.

C.1.2 Theoretical aspects

In this section the theoretical analysis methods of the existing deterministic and probabilistic diagnostic algorithms are reviewed. we also examine the relevance of these analysis methods in correspondence with our algorithms.

Deterministic diagnostic methods

Deterministic diagnosis algorithms are expected to produce a correct and complete diagnostic image of the system under certain well-defined circumstances. The primary objective of theoretical analysis is the derivation of the necessary and sufficient conditions under which correct and complete diagnosis is possible. This evaluation phase is called the *diagnosability* *analysis.* Then, the analysis process takes the form of a *correctness proof* to show that the algorithms are really accurate in the existence of the derived conditions.

The traditional Preparata system model identified two different maintenance policies and thus two distinct diagnostic goals. In the first approach one wants to know the location of every faulty unit at once, in order to replace all of them with spares. The conditions of this diagnostic goal are provided by the *one-step diagnosability* analysis. Another approach is to find only one faulty unit in one step. The found faulty unit is repaired and the same diagnostic process is repeated. This procedure continues until every unit becomes fault-free. The circumstances under which this maintenance strategy succeeds are revealed by the *sequential diagnosability* analysis.

The common property in both one-step and sequential diagnosis is that the set of identified faulty units must exactly correspond to the set of actually faulty units. Friedman [53] proposed that a less ambitious goal could be reached with less stringent requirements. His idea was that replacing a *set of processors* (including some that could be fault-free), might be acceptable when correct and complete diagnosis is impractical.

Probabilistic diagnostic methods

Deterministic diagnosis identifies the entire fault set (or a predefined subset of all faults) from the syndrome, provided that certain restrictions on the testing structure and the behavior of the units hold. By contrast, probabilistic diagnosis methods try to locate the faulty units only with *high probability.* They require no restrictive assumption on the set of faulty units or the structure of the testing assignments, but on the other hand they cannot guarantee a correct and complete diagnosis in every situation. Thus, the concept of diagnosability is not applicable in this case. Since diagnosability analysis is an inappropriate method to evaluate probabilistic diagnosis algorithms, they must be studied by other means. The three existing approaches used to evaluate probabilistic diagnosis are:

1. Analyzing the diagnosis accuracy produced in certain situations.

2. Finding the necessary conditions to guarantee that the set of most likely faulty units given the syndrome is found.

3. Proving that the diagnosis accuracy will approach 100 percent as the number of units in the system grows to infinity.

The first method is the most straightforward, end it is also our method of choice. However, we do not attempt to obtain the required information through mathematical analysis due to the complexity of the generalized test invalidation theory, instead, we measure the different aspects of diagnostic accuracy and algorithm performance using simulation.

The second approach may seem particularly appealing from a practical viewpoint. Note, that this analysis method is the counterpart of the deterministic diagnosability analysis in the probabilistic environment. Recall, however, it has been shown that finding the most probable diagnosis based on the global syndrome information is an NP-hard problem [105, 65].

The third argument is easier to calculate than the first two. Still, from the same point of view, is insufficient since an acceptable diagnosis accuracy is required for finite systems. In spite of this, asymptotically correct and complete diagnosis is a desirable property of any probabilistic diagnosis algorithm due to the significance of automated diagnosis for large and massively parallel systems. Obviously, we cannot simulate an infinite system (in fact, the number of units must be kept relatively small to achieve acceptable execution times and memory requirements). Therefore, only the trend of misdiagnosis relative to the increasing system size can be examined.

C.2 Measurement results

The algorithms were implemented for measurement purposes in a dedicated simulation environment. The simulation environment was developed by the author for evaluation of centralized fault diagnostic algorithms [106, 6], including but not restricted to, local information diagnosis based algorithms. As a standard setting, the simulations were performed in a two-dimensional toroidal mesh topology containing 12×12 processing elements. Random fault patterns of various sizes were injected, then the system was diagnosed by the chosen algorithm and statistical data was collected. Measurements consisted of 512 such subsequent simulation rounds. Although several homogeneous and heterogeneous invalidation schemes were involved in the measurements, for conciseness the presented results in most cases correspond to only the symmetric (PMC) test invalidation model, which is the most pessimistic (providing the least diagnostic information) test invalidation model.

For the measured parameter values we introduce a shorthand notation to be used in tables and figures. This notation and the computation of the statistical data is described in Table C.4. The most important global parameters are: the *size of the system* (the number of simulated processing elements) which is by default $n = 144$ in out measurements, the *fault set size* (the mean number of the randomly injected processor faults), the connectivity of the testing graph in the given system topology, and the number of simulation rounds. There are also momentary values respective to the actual simulation round r_i, these are: the current number of faulty units, and the number of incorrectly diagnosed fault-free and faulty units in the generated diagnostic image. Momentary data is collected in $\rho = 512$ simulation rounds and the following statistical parameters are computed: the number of simulation rounds in which at least one processor was incorrectly diagnosed (which is proportional to the *probability* of incorrect diagnosis), the maximum number of incorrectly diagnosed units (which is a *worst-case* measure of diagnostic accuracy), and the average number of incorrectly diagnosed units expressed in the percentage of the system size. None of these values describes completely the diagnostic properties of the studied algorithms, therefore additional parameters (standard deviation, confidence interval, etc.) were also computed. Yet, for volume constraints all of the measured parameters cannot be presented here. We tried to select the most characteristic statistical parameter for each presented property.

Table C.4: Measured parameter values

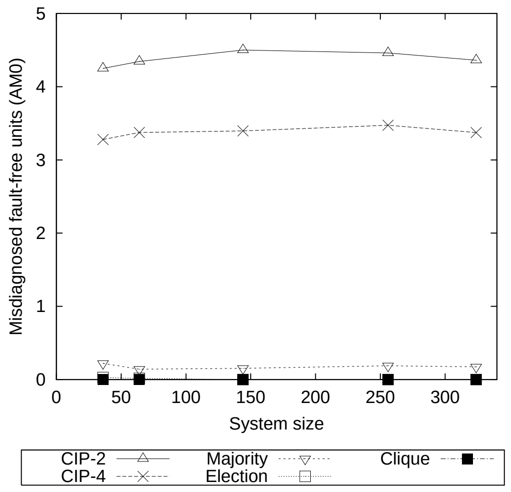
Global parameters		
Name	**Description**	**Computed as**
n	System size (number of units)	$\|U\|$
ν	Connectivity of the testing graph	$\min_{u_i \in U} \|\Gamma(u_i)\|$
ρ	Number of simulation rounds	$\|R\|$
Momentary parameters (corresponding to round r_i)		
Name	**Description**	**Computed as**
ϕ_i	Fault set size (number of faulty units)	$\|F_i\|$
$\mathrm{M0}_i$	Number of misdiagnosed fault-free units	$\|U_i^1 \cap (U - F_i)\|$
$\mathrm{M1}_i$	Number of misdiagnosed faulty units	$\|U_i^0 \cap F_i\|$
Statistical parameters		
Name	**Description**	**Computed as**
MR	Number of rounds with incorrect diagnosis	$\| \{r_i : r_i \in R, (\mathrm{M0}_i + \mathrm{M1}_i) > 0\} \|$
MM1	Maximum number of misdiagnosed fault-free units	$\max_{r_i \in R} \mathrm{M0}_i$
MM1	Maximum number of misdiagnosed faulty units	$\max_{r_i \in R} \mathrm{M1}_i$
AM0	Average percentage of misdiagnosed fault-free units	$100 \cdot \left(\sum_{r_i \in R} \mathrm{M0}_i / n\right) / \rho$
AM1	Average percentage of misdiagnosed faulty units	$100 \cdot \left(\sum_{r_i \in R} \mathrm{M1}_i / n\right) / \rho$

C.2.1 Comparison of fault classification heuristics

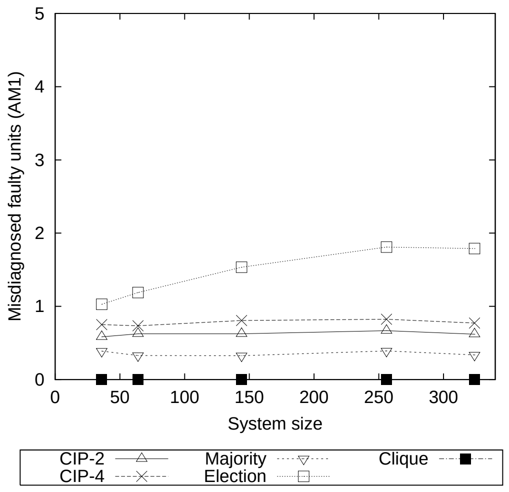
In this section we compare the performance of the developed inference propagation based and scalar LID algorithms. The diagnostic results of the LMIM, DIL, and LTC methods are not considered separately, as in Section 2.2 they were proven to generate the same implication set when run in an appropriate number of iterations. The measurement experiments have confirmed that these methods produced identical diagnostic results with the same fault classification heuristic employed. Therefore, only the CIP-2 and CIP-4 scalar methods and the Majority, Election and Clique heuristics are presented from the viewpoint of diagnostic accuracy; while LMIM, DIL, and LTC are considered separately from the viewpoint of time complexity. In the rest of this section we study the effect of the parameter variations as specified by Table C.2.

System size. To illustrate the tendency of diagnostic accuracy as the system size increases, we varied the number of units in the simulated two-dimensional toroidal mesh topology. Five system sizes were simulated, having respectively 6×6, 8×8, 12×12, 16×16, and 18×18 units. In each system 50 percent of the processors were faulty. The results, shown in Figure C.1, represent the average number of incorrectly diagnosed units in the percentage of the system size. Two-step implications were used in the fault classification. The figure well indicates the expected convergence of the diagnostic accuracy to a constant value. In toroidal systems the constant value is approximated from below, as the wrapped around test connections provide gradually less diagnostic information compared to the size of the system.

The Clique heuristic did not make any diagnostic mistakes in the examined measurement range. However, this impressive diagnostic accuracy also has its price, namely the large amount of *unknown* units. To illustrate this attribute of the Clique heuristic, the average



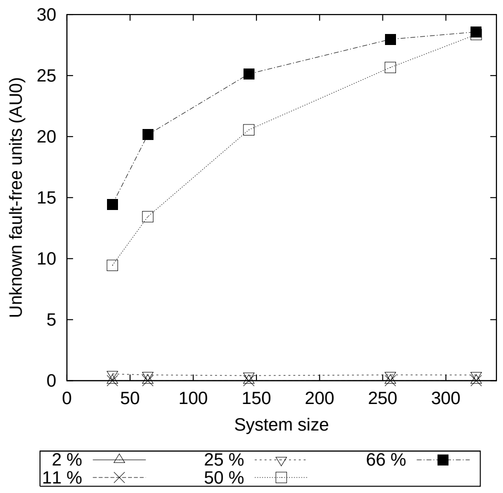
(a) Average misdiagnosed fault-free units



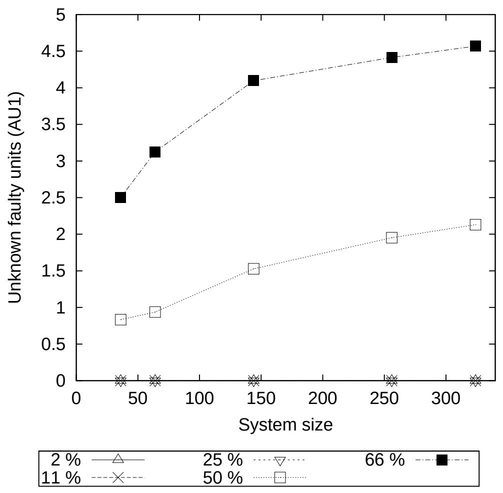
(b) Average misdiagnosed faulty units

Figure C.1: Average misdiagnosed units versus system size

number of unknown fault-free units (AU0) and unknown faulty units (AU1) is presented in the percentage of the system size in Figure C.2, for various fault probabilities ranging from 2 percent to 66 percent. The AU0 and AU1 values are computed analogously to the AM0 and AM1 values presented in Table C.4. Clearly, the total number of unknown units (AU0+AU1) considerably exceeds the total number of units incorrectly diagnosed (AM0+AM1) by the other methods. On the other hand, the Clique heuristic is the only option in those applications which do not tolerate diagnostic mistakes of any kind.



(a) Average unknown fault-free units

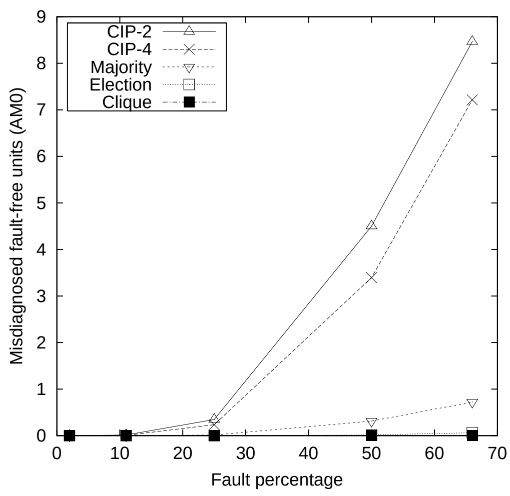


(b) Average unknown faulty units

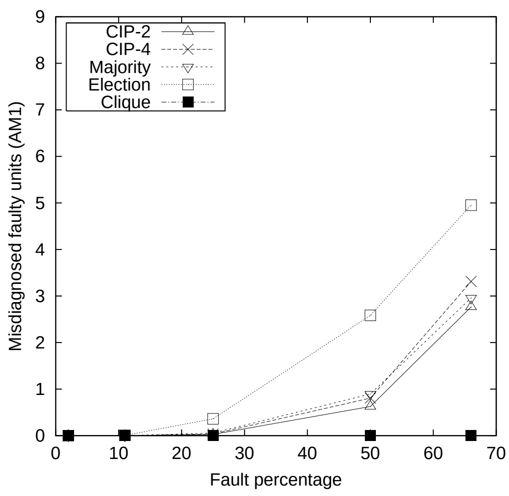
Figure C.2: Average unknown classified units by the Clique heuristics

Fault set size. The effect of the fault set size on diagnosis accuracy is shown in Figure C.3.

The figure shows the average number of incorrectly diagnosed units in the percentage of the system size for various fault percentages. It can be seen, that diagnostic accuracy drops when the amount of faulty units reaches 50 percent of the system size. However, diagnosis is quite accurate (less than 0.2 percent of the total 144 processors) in the practically relevant 0–25 fault percent. The Clique heuristic did not make any diagnostic mistakes in this range.



(a) Average misdiagnosed fault-free units



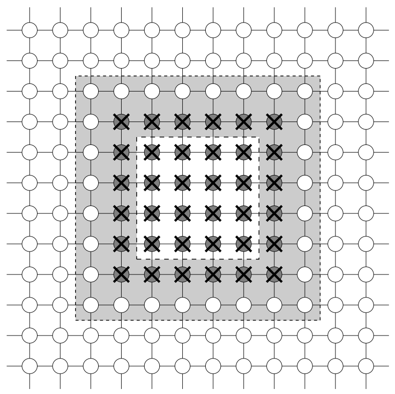
(b) Average misdiagnosed faulty units

Figure C.3: Average misdiagnosed units versus fault set size

Fault arrangement. For analyzing group faults we used the user-defined static fault patterns shown in Figure C.4. The measurement results for the whole system can be seen in Table C.5. The diagnostic algorithms computed the implication sets 1, 2, and 4-step implication, but as we expected, there was no significant improvement using 4-step implications over 2-step implications. Note, that for Pattern (a) neither algorithms diagnosed incorrectly the fault-free units, and even for Pattern (b) only the CIP-2 algorithm made such mistakes. The Clique again did not make any diagnostic mistakes. Also note, that the maximum number of incorrectly diagnosed faulty units is surprisingly small compared to the fault set size. This is due to the sure fault classifications provided by the detected contradictions.

The advantage of user-defined fault patterns is that the simulation environment can selectively prepare statistics to a subset of chosen units. We used this feature to verify the behavior of the diagnostic algorithms on the fault group borders. For this purpose the border region indicated by grey background Pattern (a) was chosen for selective statistics. Diagnostic mistakes occurred only for isolated faulty units, none of the units in the border area have been misdiagnosed.

Diagnostic information. Like Table C.5 shows, inference propagation (increasing the length of implication chains) improves the diagnostic performance. Figure C.5 presents this effect in the case of randomly injected fault patterns consisting of 25 and 50 percent faulty units. The number of simulation rounds with incorrect diagnosis is shown in the function of inference propagation iterations. The diagnostic information used in the fault classification were


(a) Closed fault pattern

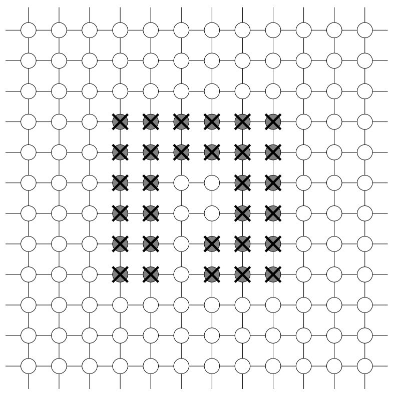

(b) Open fault pattern

Figure C.4: Fault patterns to examine diagnosis on the fault group borders

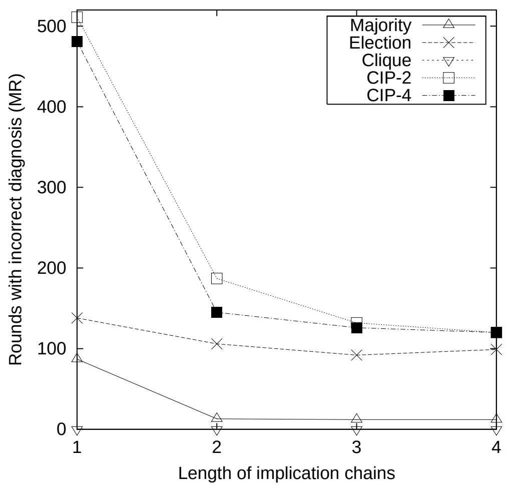
Table C.5: Maximum misdiagnosed units in group fault patterns

		Majority	Election	Clique	CIP-2	CIP-4
Pattern		MM0 / MM1	MM0 / MM1	MM0 / MM1	MM0 / MM1	MM0 / MM1
(a)	1-step	0 / 9	0 / 7	0 / 0	0 / 9	0 / 7
	2-step	0 / 6	0 / 6	0 / 0	0 / 8	0 / 6
	4-step	0 / 5	0 / 6	0 / 0	0 / 8	0 / 6
(b)	1-step	0 / 8	0 / 5	0 / 0	4 / 6	4 / 5
	2-step	0 / 2	0 / 4	0 / 0	3 / 6	6 / 3
	4-step	0 / 2	0 / 4	0 / 0	3 / 5	6 / 2

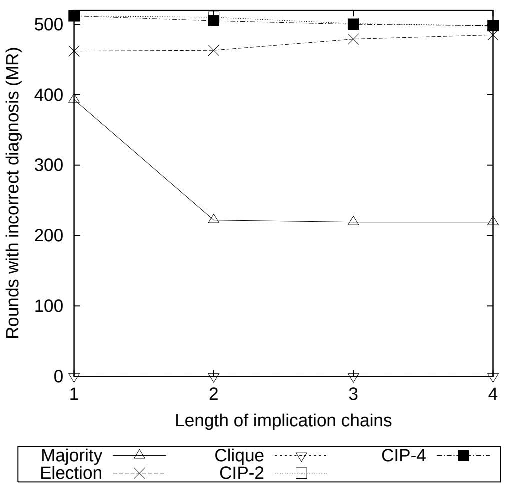
increased gradually from one-step implications to four-step implication chains. The results justify our assumption: the measured values quickly converge to a lower limit and subsequent inference propagation iterations improve diagnostic accuracy less and less.

Topology. We also examined the effect of system topology on diagnostic accuracy. Three regular interconnection topologies were simulated as illustrated in Figure C.6: (a) hexagonal toroidal grid with three connections, (b) 2-dimensional toroidal mesh with four connections, and (c) triangular toroidal grid with six connections. Figure C.7 plots the number of simulation rounds with incorrect diagnosis in the function of connectivity. Random fault patterns consisting of 25 and 50 percent faulty units were injected in the system. As it can be seen, all of the heuristics perform better regardless of fault set size as the number of connections increases. In a completely connected system each heuristic would provide a correct and complete classification.

Time complexity versus diagnostic information. As seen in Table C.6, the LID methods have linear time complexity with respect to the number of transitive propagation iterations, except the LTC algorithm. In the LTC algorithm the amount of considered diagnostic information depends on the local environment from which the implications are collected. The complexity of the method is $O(n\nu_k^3)$ in the function of the ν_k size of the local environment. The table contains the total simulation time of the 512 rounds in seconds, for the default system composed of 144 units. The duration of a single diagnosis with three iterations



(a) Incorrect rounds at 25 percent faults



(b) Incorrect rounds at 50 percent faults

Figure C.5: Simulation rounds with incorrect diagnosis versus implication length

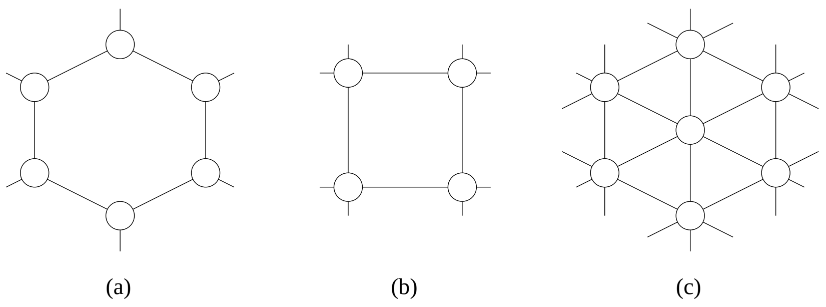
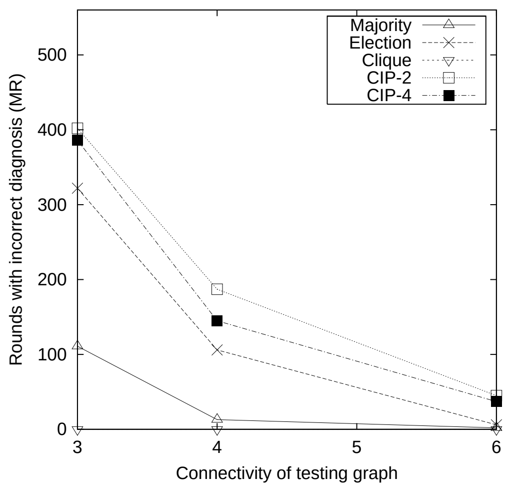


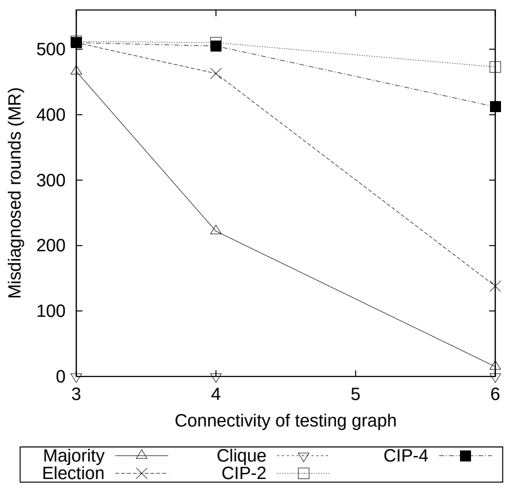
Figure C.6: Simulated system topologies

ranges from less than 5 milliseconds (for the fastest CIP-2 algorithm) to 8 seconds (for the slowest LTC algorithm), which are more than adequate for practical applications.

The results were obtained using a single processor system equipped with Pentium Celeron processor running at 333 MHz clock frequency. At the time of writing this configuration is regarded as a low-class personal computer. The code of the simulating environment and the simulated algorithm are linked together into a single executable. Unfortunately, the employed development software did not support the detailed examination of program modules, therefore we could only measure the total simulation time. As a consequence, the time measurements contained a constant, uniform error: the time spent in the simulator. To eliminate this error we created a *null* diagnostic program, which simply copied the set of physical units states into the set of diagnosed unit states, furthermore did nothing. By modifying the time measurement results with the delay of the null program we could eliminate the measurement error.



(a) Incorrect rounds at 25 percent faults



(b) Incorrect rounds at 50 percent faults

Figure C.7: Simulation rounds with incorrect diagnosis versus connectivity

Table C.6: Time complexity versus number of iterations

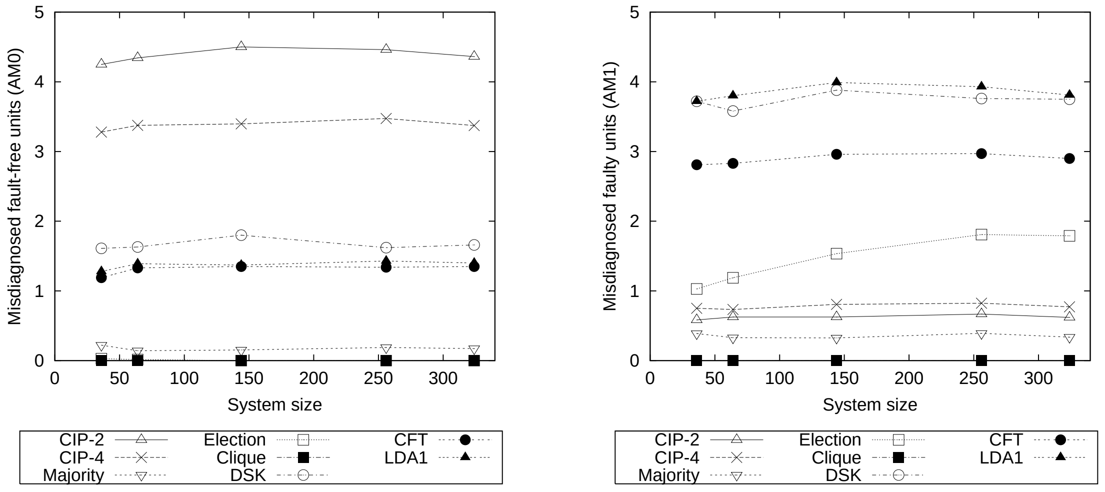
	Simulation time (seconds)				
Iterations	**LMIM**	**DIL**	**LTC**	**CIP-2**	**CIP-4**
None	4.3	7.3	4.8	1.4	1.5
1	796.4	17.3	170.5	1.8	1.9
2	1582.3	27.6	1215.6	2.1	2.5
3	2233.1	38.2	4275.3	2.4	2.9

C.2.2 Comparison to existing probabilistic methods

In order to compare the performance of the developed probabilistic methods to the existing approaches, we implemented several known algorithms taken from the literature, namely: the Blough, Sullivan, and Masson (BSM) algorithm [62], the LDA1 algorithm of Somani and Agarwal [63, 64], and the Dahbura, Sabnani, and King (DSK) algorithm [67, 68]. Also included in this comparison is our CFT algorithm, which despite that it was developed by the author is treated as an external algorithm as it does not fit into the framework of local information diagnosis. The mentioned algorithms are all included in Section 1.2.2, which outlines their fault classification mechanism.

During the measurements we experienced that the diagnostic accuracy provided by the BSM algorithm is so low, that it is simply not comparable to the other methods. Therefore, the BSM method was omitted from the diagrams.

System size. In Figure C.8 the average number of incorrectly diagnosed units in the percentage of the system size are displayed for all of the compared algorithms. The measurements were carried out under the same circumstances as in the previous section. Clearly, the inference propagation based LID methods have the best overall diagnostic accuracy. The scalar LID methods provide worse accuracy than the DSK, CFT, and LDA1 algorithms with regard to *fault-free to faulty* (benign) diagnostic mistakes, but they perform much better with regard to *faulty to fault-free* (malign) diagnostic mistakes. The higher benign misdiagnosis only reduces the amount of available resources, while the lower malign misdiagnosis implies that the CIP algorithms are less prone to endanger system integrity by missing faulty processors. Therefore we say that scalar methods have more favorable diagnostic properties than the methods taken from the literature, despite that they have similar overall (AM0+AM1) diagnostic inaccuracy.

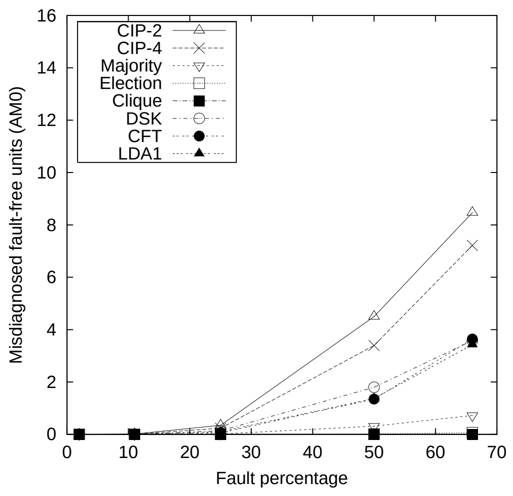


(a) Average misdiagnosed fault-free units (b) Average misdiagnosed faulty units

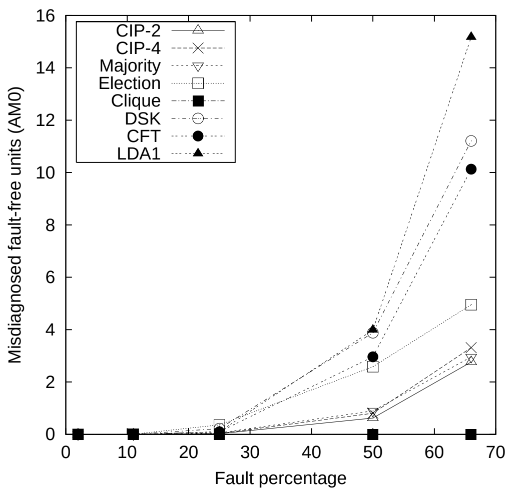
Figure C.8: Average misdiagnosed units versus system size

Fault set size. The developed and the existing algorithms are compared with regard to fault set size in Figure C.9. The figure presents average number of incorrectly diagnosed units in the percentage of the system size, measured with the same settings as in the previous section. It can be seen that for smaller fault sets all the probabilistic algorithms perform quite well (less than 1 percent of the total number of units is misdiagnosed under 30 percent fault probability!), but the strength of the inference propagation approach is obvious above 25 percent fault probability. The difference in the diagnostic accuracy increases in parallel with higher fault percentage.

Invalidation. A notable advantage of local information diagnosis based methods is that they are based on generalized test invalidation and therefore are capable of utilizing the additional diagnostic inferences contained in non-symmetric invalidation models. To verify the improvement the extra implications yield, we studied the behaviour of the algorithms in the homogeneous symmetric, four homogeneous non-symmetric, and a heterogeneous test arrangement. The heterogeneous situation was imitated by randomly selecting the test invalidation of each unit with equal probability from the set of non-equivalent testing models listed in Table 1.1. The simulated system contained 144 processors arranged in a two-dimensional toroidal mesh topology, of which 50 percent was faulty. The results show that the diagnostic accuracy of LID methods is in general several times higher than that



(a) Average misdiagnosed fault-free units



(b) Average misdiagnosed faulty units

Figure C.9: Average misdiagnosed units versus fault set size

of the algorithms taken from the literature. For certain invalidation models: T_{0X} and T_{X0} the difference can be more than one order of magnitude, but also in the heterogeneous case it is quite significant. We must also mentioned that although the inference extraction and propagation procedures generate all of the available implications, the employed fault classification heuristics are invalidation-independent. Taking into consideration the characteristics of the different non-symmetric invalidation models could result in much better diagnostic performance in certain cases (e.g., the T_{X1} model). However, our aim was to present a general diagnostic methodology, therefore we intentionally did not make adjustments for improving the methods in particular situations.

Table C.7: Average misdiagnosed units in various invalidation models

	PMC, T_{XX}	**ST**, T_{0X}	**HK1**, T_{1X}	T_{X0}	**BGM**, T_{X1}	**Random**
	AM0 / AM1	AM0 / AM1	AM0 / AM1	AM0 / AM1	AM0 / AM1	AM0 / AM1
Majority	0.15 / 0.33	0.00 / 0.02	1.89 / 0.06	0.00 / 0.00	0.02 / 1.56	0.01 / 0.85
Election	0.00 / 1.53	0.00 / 0.02	0.00 / 1.58	0.00 / 0.00	0.02 / 2.02	0.04 / 0.81
Clique	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00	0.00 / 0.00	0.00 / 0.63	0.01 / 0.25
CIP-2	4.50 / 0.63	0.00 / 0.19	5.66 / 0.88	0.47 / 15.25	0.19 / 0.00	2.26 / 1.33
CIP-4	3.40 / 0.81	0.00 / 0.19	4.12 / 1.47	1.51 / 14.20	0.19 / 0.00	2.08 / 1.22
BSM	2.54 / 13.71	0.00 / 13.21	15.57 / 13.21	2.54 / 34.47	2.54 / 0.00	2.33 / 11.92
DSK	1.80 / 3.88	0.00 / 2.17	8.23 / 9.49	5.95 / 9.52	1.24 / 3.55	1.81 / 3.98
CFT	1.35 / 2.96	0.00 / 1.85	7.33 / 7.83	6.29 / 9.40	1.85 / 2.03	1.27 / 3.01
LDA1	1.37 / 3.99	0.00 / 2.97	7.53 / 6.42	3.49 / 21.99	1.37 / 0.00	1.30 / 3.77

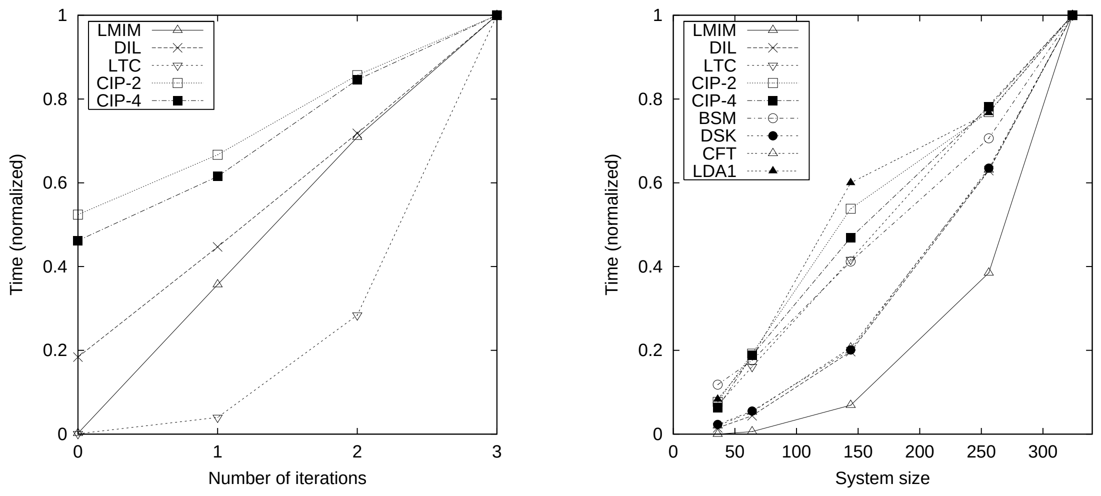
Time complexity versus system size. Table C.8 lists the simulation time of the compared algorithms. The table contains the total time of the 512 simulation rounds in seconds for 25 percent fault probability. The LID methods were executed in one iteration. The duration of a single diagnosis ranges from less than 80 μseconds (for the fastest BSM algorithm) to

22 seconds (for the slowest LMIM algorithm). The 22 second delay for a system containing 324 processors may be acceptable in practical situations, but it is obvious that the $O(n^3)$ time complexity of the LMIM algorithm makes it inapplicable for larger systems.

Table C.8: Time complexity versus system size

	Simulation time (seconds)								
Size	**LMIM**	**DIL**	**LTC**	**CIP-2**	**CIP-4**	**BSM**	**DSK**	**CFT**	**LDA1**
36	4.4	1.3	28.0	0.2	0.2	0.04	0.08	0.08	0.05
64	69.5	3.8	65.6	0.5	0.6	0.06	0.19	0.21	0.11
144	796.1	17.0	170.2	1.4	1.5	0.14	0.70	0.82	0.36
256	4428.7	54.4	320.8	2.0	2.5	0.24	2.21	2.51	0.46
324	11505.2	86.5	408.6	2.6	3.2	0.34	3.48	3.98	0.60

To make the time complexity of the various algorithms easy to evaluate, we plotted the simulation time against the number of iterations and against the system size in Figure C.10. The diagrams had to be scaled down by the maximum value in order to be comparable. The polynomial time complexity of the LTC algorithm with regard to iteration number, and the polynomial time complexity of the LMIM and DIL algorithms with regard to system size are clearly visible. Interestingly, the CFT algorithm also shows polynomial time complexity, but it is due to the constant fault probability, since CFT has quadratic time complexity respective to the fault set size.



(a) Time complexity versus iterations (b) Time complexity versus system size

Figure C.10: Measured time complexity

Appendix D

Communication protocols for distributed diagnosis

This chapter presents the synchronous and asynchronous communication protocols employed in the initial and working phase of the developed distributed diagnostic algorithm. These protocols are integral parts of the diagnostic procedure, as in distributed algorithms the reliable broadcast of the local test result report messages is essential for the correct operation.

D.1 Communication in the initial stage

The initial stage of the algorithm is executed immediately after switching on the Parsytec GCel machine. The purpose of the initial stage is to obtain an initial diagnostic image of the system, therefore the information exchange takes place in only one diagnostic round (i.e., every processor generates and distributes its local syndrome only once). This means that the employed communication protocol must implement a high-performance fault-tolerant global multicast. The characteristics of the required communication protocol can be summarized as: test result report messages are sent from each fault-free processor to every other reachable fault-free processor, the system may contain multiple faulty units, synchronous communication must be used for fast message transfer, the protocol must use "dummy messages" to handle irregularity in the interconnection topology, communication terminates when each message at every unit was delivered, a separate termination stage is necessary.

The choice of synchronous communication for the initial stage is motivated by the following considerations. Since messages are exchanged between every connected fault-free processors, the message volume can be very high depending on the size of the system. The accurate number of messages in a two-dimensional mesh topology having $x \times y$ processors—assuming that processors forward the messages to all of their fault-free neighbors and faults are permanent—can be written as:

$$(x+y)\big(4xy - 2(x+y) - 4\phi + \phi_b + 2\phi_c\big)$$

where ϕ is the number of faulty units, ϕ_b is the number of faulty units situated on the borders of the mesh, and ϕ_c is the number of faulty units located in the corners of the mesh. The fastest possible communication model must be employed to minimize the distribution

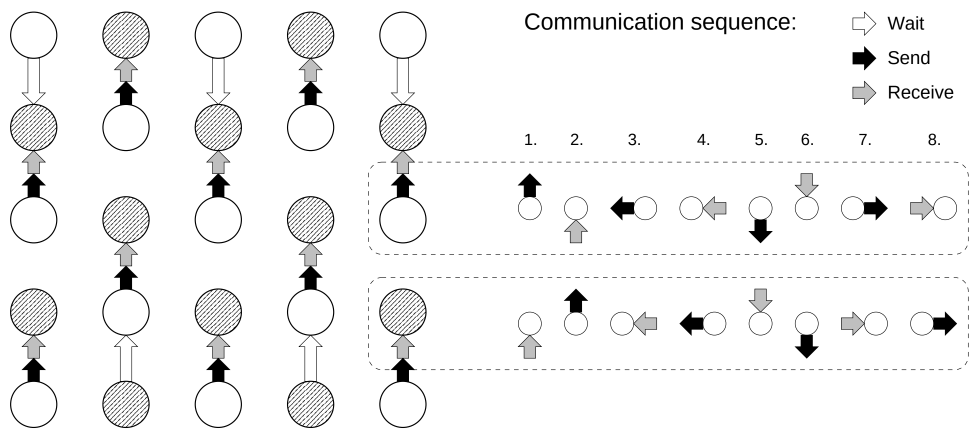


Figure D.1: The synchronous communication protocol

time of messages. The synchronous communication model has low-level supported built-in the transputer hardware, thus it is much faster than the asynchronous transfer methods. For this and other reasons [107] synchronous communication is preferred when large information volume needs to be exchanged.

The probability of multiple simultaneous fault occurrences is considerable after switch-on, requiring protocols tolerating even large amount of faulty units in the initial stage. For this purpose our protocol has a simple routing mechanism which does not execute separate fault-handling code. On the contrary, the larger the number of faulty units in the system the faster the protocol will be (because the number of transmitted messages is decreased). The operation of the protocol in a 5×5 two-dimensional bordered mesh topology is illustrated in Figure D.1 (the processor-to-processor interconnections are not shown).

The protocol is composed of a set of communication actions executed in a predefined schedule. The communication steps are described in the right side of the figure. Black arrows represent a send, gray arrows a receive activity. The sequence of communication actions, however, is not identical for every unit. Two types of units exist, represented by white and shaded nodes that are placed on the mesh in a chequered pattern. The white nodes correspond to units that use the upper communication sequence. These units in the first step of the protocol send in the north direction, then in the second step receive from south, then in the third step send to west, etc. The shaded nodes use an inverted sequence, created by swapping the role of communicating parties (senders become recipients and vice versa) and accordingly, the direction of transmission. These communication actions are listed in the lower row. Figure D.1 also illustrates the first step of the protocol in the left side. Note, that except some units on the border that are forced to wait, practically all of the processors are transferring messages. In the second step the direction of message transfer is inverted, so again almost every processor will have a communicating pair and different units on the mesh border will have to wait. So continues the execution of the protocol until the eighth step, when the whole sequence is started again. The protocol is applicable for torus topology as well as bordered mesh topology, in fact in a toroidal topology there are no waiting nodes at all.

The above mechanism forwards incoming messages to each of the fault-free neighbors of a certain unit (except of course the sender of the actual message). This ensures that each fault-free processor belonging to the same strongly connected component will eventually obtain every test result report message. However, the protocol is not minimal in terms of messages transmitted. Some processors may receive the same information two or even more times. Additional copies of the same, already received message are ignored. Nevertheless, the time spent by transferring these redundant TRR messages is lost. More complicated routing algorithms can reduce or even eliminate this redundancy. Two main reasons motivated the choice of the simplest possible routing method:

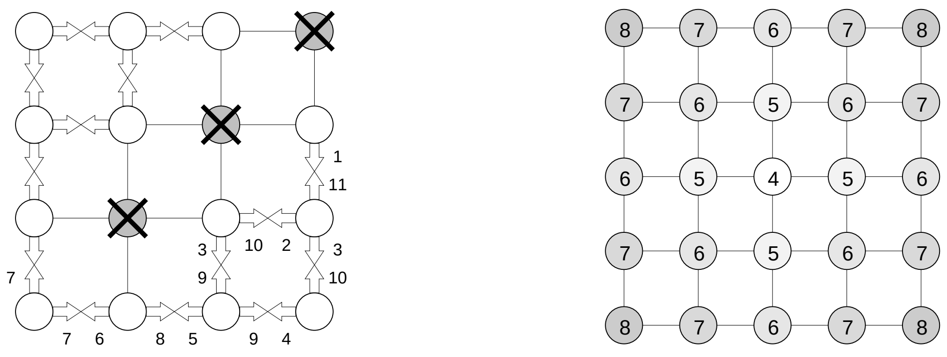
- An improved routing algorithm may require the knowledge of the global diagnostic image of the system. This system-wide diagnostic image can be needed for the routing algorithm in order to construct an optimal message distribution scheme. However, this global diagnostic information is not available during the initial stage.

- A smaller message count does not necessarily imply the reduction of the execution time. The execution of a complex routing algorithm to optimize the transmission can be time-consuming. Furthermore, the trade-off between the two choices, that is whether to use a simple method with fast execution time and some redundant messages or a more complex and slower method with no redundant messages, is application-dependent.

Some messages transmitted during the synchronous communication process are *dummy messages*, that is they do not represent valuable information. They are only required to maintain the continuous operation of the synchronous protocol. The necessity of dummy messages is illustrated in Figure D.2(a). The numbers written on the arrows show the amount of diagnostic messages that the adjacent unit can send to the respective direction. As it can be seen, the number of messages transmitted in different directions are usually not equal. Consequently, after a certain number of communication rounds the processors will have no more information to send in one direction while still several messages are waiting to be forwarded in other directions. Naturally, processors cannot ignore the "depleted" directions, since then a deadlock, breaking the execution of the protocol and blocking the entire system, would occur. As a solution, empty "dummy messages" must be transferred to all directions in which no new information is available.

The synchronous communication terminates when all of the TRR messages have been delivered. If the system is cut up into several strongly connected components by barriers formed of faulty processors, only the members of the same strongly connected component are reachable and communication with units in other components is impossible. Therefore, the knowledge of the node accessibility graph is necessary in order to detect when the termination criterion is fulfilled. Note, that although a global diagnostic image (from which the node accessibility graph may be derived) is not available during the communication, by the time the last TRR message is received and processed the diagnostic image will be gradually completed (recall, that diagnosis is integrated with message transfer), and hence the termination criterion can be correctly evaluated.

Not all processors terminate simultaneously, as units are located in various distances from each other. This phenomenon is illustrated in Figure D.2(b). The numbers written



(a) Necessity of dummy messages (b) Necessity of a termination phase

Figure D.2: Handling irregularity of the interconnection graph

into the nodes indicate the maximum distance of the respective unit to any other unit. Messages travel one direct connection per one communication round, thus units in more central positions may receive every TRR message earlier than outer units. Immediate termination of the earlier finished unit could cause a deadlock, because its neighbors are unaware of the fact that the unit has fulfilled its termination criterion. Two solutions can be applied to this problem:

1. Processors can maintain a record about the messages received by their neighbors. They are informed about these messages, since either they sent a certain information to the neighbor, or the neighbor received the information from another unit and forwarded it to them. A processor can detect if any of its neighbors is terminating by examining the maintained records.

2. A separate *termination phase* can be employed. The finishing processor must perform one additional communication round. During this final round, the processor receives pending messages from its neighbors (all of these messages are either dummy or duplicate, otherwise the termination condition would not have been met) and sends special messages to each neighbor notifying them about its intention to finish. The termination phase method does not require memory for storing message record opposite to the first solution. It also does not imply longer execution time of the communication protocol, as the last finishing processor—without active neighbors to inform about the termination–will not perform the additional termination phase.

Based on these considerations the second solution was integrated in the developed distributed diagnostic algorithm.

D.2 Communication in the working stage

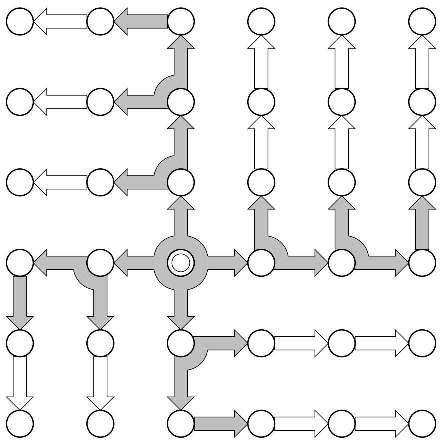
The working stage of the algorithm begins after the initial stage has terminated. During further operation the diagnostic algorithm remains in the working stage (except when two isolated parts of the system are united by a processor repair and the algorithm returns to the initial stage). As the majority of potential faults were detected during boot-time, infrequent fault events with a small number of simultaneous fault occurrences are expected. Therefore, only the differences from the initial system-wide diagnostic image need to be distributed and processed. Messages are disseminated with a communication protocol having the following main characteristics: asynchronous communication model, lower message count than in the initial stage, more complex routing method, distributed message routing based on local fault patterns, no termination phase is necessary.

As after the initial stage a global diagnostic image is available and only a moderate amount of new diagnostic events (fault occurrences and unit repairs) are expected, an *event-oriented*, *incremental* test result report distribution strategy can be used to significantly reduce the communication load. However, the communication model used in the initial stage is not suitable for this purpose. Synchronous transmission works well for intensive communication and large message count. The infrequent communication in the working stage would cause the system to be generally in a state waiting for synchronization, since the sending process would be blocked until the receiving process is ready. For low intensity message transfer the *asynchronous* communication model, performing the low-level send/receive actions in the background and allowing the processor to execute other tasks as well, is much more appropriate.

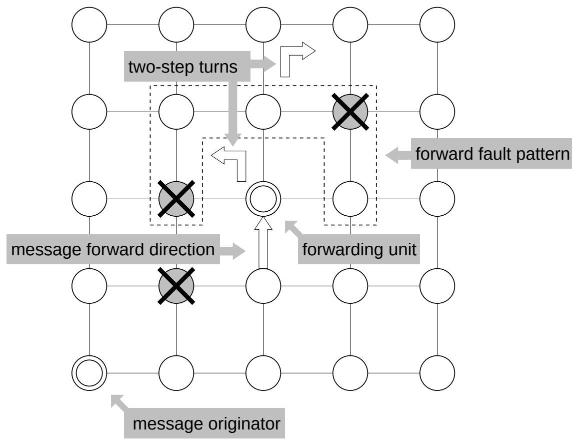
Because of the entirely different nature of communication, a new protocol with distributed message routing was developed for the working stage. Instead of choosing between the two alternatives: whether to employ a simple but redundant method or a more complex routing procedure with less redundant messages, a combined mechanism was developed. A simple routing method was created to provide an optimal message dissemination in the fault-free system structure. Routing is based on the relative position between the unit forwarding the incoming message and the originator of the message. However, the method does not function when faulty units are present in the system. Therefore, a supplemental procedure which compensates for the effect of faulty units was added. The characteristics of the resulting routing algorithm are optimal information transfer in a fault-free system, and low amount of redundant messages if faults are present. The routing process is executed independently on each processor using local test results, and it is automatically adapted to the possible changes of the fault situation.

The routing protocol of the working stage implements a one-to-all multicast, fulfilling two main requirements: (1) it guarantees that every fault-free unit reachable by the message originator eventually receives the information, and (2) the generated message volume is kept as low as possible (in the optimal case the processors receive every message only once).

The developed routing algorithm is based on three different source of knowledge: (1) relative position from the message originator, (2) local fault pattern, and (3) additional directions included in the message. The relative position between the message originator and the unit forwarding the message are used to create an optimal distribution scheme for the fault-free case. An example optimal distribution path is presented in Figure D.3(a). When a processor receives diagnostic message, it calculates the relative position from the unit starting the message. In the so called "minor" relative positions (northwest, southwest, southeast, northeast areas marked by white arrows) messages are forwarded in only one direction. In the "major" relative positions (northern, western, southern and eastern line of



(a) Routing in a fault-free system



(b) Routing based on local information

Figure D.3: The asynchronous communication protocol

processors) two copies of the message are forwarded to two different neighbors. The method is optimal for the fault-free mesh, that is all units receive the messages only once.

If the system contains faulty units as well, the previous method in itself is not sufficient. Faults are obstacles in the way of message distribution, so employing the presented method some processors can be left-out of the communication. Additional messages must be generated which can get "behind" the faulty units to compensate for the inhibitive effect. The two-dimensional mesh structure containing faulty processors may be considered as a maze. Then, the problem is to find a path in the maze from the entrance (the message originator) to every point (fault-free reachable processor) avoiding the walls (failed units).

For this purpose, an idea based on the famous *right-hand rule* of solving maze problems is used. The rule is utilized in the algorithm in the following way. Every unit examines the local fault pattern in the *forward direction* (e.g., if the message was received from south, then the forward direction is north). The local fault pattern is formed by the fault state of the five nearest units in the forward direction (see Figure D.3(b)). When there are faulty units in the forward local fault pattern, units add new messages to the distribution scheme. There are 32 different message addition rules, as there are five units and 2^5 fault-free/faulty combinations in the forward fault pattern. For brevity's sake we omit the description of these rules. In general, the additional messages travel along the sides of the "walls", like in the right-side rule. Therefore, messages must be able to "turn" left or right on the corner of an obstacle. A turn can be accomplished only in two steps, so routing information must also be included within the message. Two-step moves can be realized by considering these routing directions at the recipient unit. As messages travel on their distribution path and encounter obstacles formed by faulty units, with two-step distribution rules they can get behind the obstacles, and from that point the original forwarding algorithm again provides their optimal dissemination.

Appendix E

Reliable storage in APEmille

A reliable device used to store persistent information or intermediate data structures of compound operations is called a *stable storage* [94, 108]. The two main properties of this abstract device can be summarized as follows:

- Resistance against external hardware or software failures (processor failures, invalid storage accesses) and internal errors such as decays, and
- Atomicity of read and write operations.

The stable storage system we propose for recording checkpoints and other persistent data resides entirely on the disk storage of the Local Host computers. Only a part of the disk area is devoted to this purpose, the rest holds a normal UNIX file system. There is a special process, called the *Storage Manager* (SM), which has an exclusive access to the stable storage area. We assume that the resource protection and process isolation services provided by the OS are adequate to prevent any failed process to corrupt the data in the stable area.

Besides, the storage system must handle the following types of faults: *external hardware faults* such as the erroneous behavior of the host processors, *external software failures* that manifest themselves at the user interface of the Storage Manager, and *internal errors* of the SM process and the disk drive/controller. The stable storage is designed to hold objects that can have the size of a single or multiple memory pages. We are not interested in the internal structure of an object, but we require that a large object of many pages should occupy a contiguous space in the memory. There are two operations defined on the objects: a `write` command to create or update an object in the stable storage, and a `read` command to retrieve the last written value of the object.

The techniques proposed to realize the stable properties are listed in Table E.1. Soft internal errors originated in the disk drive and controller are partly handled by the hardware itself; modern storage systems are equipped with powerful coding mechanisms to detect and correct decay errors. To tolerate soft errors during disk access we employ the *careful disk operations*, as described in [109]. A *careful read* operation repeatedly performs a normal disk read until it gets a good status or a predefined limit of retries is exceeded. This eliminates soft read errors. A *careful write* operation repeatedly performs a normal disk write followed by a read until it returns a good status with the data being written. This eliminates null

Table E.1: Realization of the stable storage properties

Desired property	Requirement	Realization technique
Fault tolerance of	resilience	careful disk operations
hardware/software failures	stability	available copy replication
		Replica Majority Commit
	consistency	access control
		semantic integrity checking
Atomicity of	indivisibility	shadow updating
read/update operations	serializability	two-phase locking (in ACR)
		three-phase commit (in RMC)

writes and bad writes to good addresses. Address value problems are discovered by the Storage Manager process using access control and a simple semantic check on the objects.

Ensuring storage stability

The techniques that help the Storage Manager process to detect the interaction errors at its user interface we inherit from [108]. The purpose of these detection techniques are two-fold:

1. they ensure a proper initialization of the transactions by checking that the first access to the object is valid, and

2. they enforce that the read and write transactions respect their predefined semantics.

The checking of the first access is performed by a key control mechanism. Any transaction that changes an object must first provide a *key*, which is inspected by the SM. For the purpose of the key we suggest to use a checksum, generated from the entire data content of the object. The checksum mechanism should be inexpensive with respect to the computation, but must be capable of verifying the integrity of the object. The checksum is stored together with the object. At the beginning of the transaction the SM compares the stored and the provided checksums. If they match, then the access to the object is granted to the requesting process. When the transaction successfully terminates, the Storage Manager computes a new checksum based the changed data contents and stores it as the new key to the object. On the next access the requesting process must present the new key to gain control of the object.

After passing access control the requested transaction can take place. Interaction problems during the execution of the transaction are detected by checking the semantics of the performed operation. The following two properties characterize the semantics of atomic read and write transactions:

P access: any access to an object implies accessing all of the pages it is composed of once and only once. This property encompasses the unitary ("all or nothing") characteristic of the atomic transactions on the object.

C access: all the pages that constitute an object are accessed in the ascending order of their logical address. This property helps to filter out incorrect accesses to the object.

These properties fit well the purpose of storing checkpoints in our stable storage, since the scientific computations running on APEmille typically use large contiguous data structures such as vectors and arrays. Although the semantic checking power they provide is not very strong, they detect the most common malfunction during the stable storage access [108], the address value failure. And they can be verified quite easily by a simple counter. More powerful semantic checking mechanisms (such as application-specific data acceptance tests) can be developed with a deeper knowledge of a certain application.

In order to implement stability, there must be at least one intact and up-to-date instance of the data deposited in the stable storage at any time. Permanent internal faults and external hardware/software failures can make the disk storage of any host computer unavailable. Therefore, the stable data area must be maintained at multiple host computers. The different copies of the stable data area are called *replicas*, they a supervised by a *replica manager* (in the APEmille the Storage Manager process can fulfill this role) which uses a *replica control protocol* to keep the available copies up-to-date and organize the accesses to the replicated objects user-transparently, as if it would be a single, highly available storage system. Two protocols were selected for APEmille depending on the system configuration: the *available copy replication* (APC) for smaller systems, and the *Replica Majority Commit* (RMC) protocol for complex configurations. Details of these replica control protocols are presented in the next section.

The atomicity of the read/write transactions can be guaranteed by fulfilling two basic conditions. The simplest method for ensuring the unitary or indivisibility property of write transactions is *shadow updating* [110]. This technique allocates two buffers of identical size to record the object. One of the buffers has the actual data of the object, the other one is called the *shadow* buffer. A binary pointer indicates which buffer plays the active/shadow part at a certain point of time. A write operation is carried out in two steps: (1) the new value of the object is written in the shadow buffer; (2) the pointer is inverted, i.e., the role of the active/shadow part is exchanged. If an error occurs during the first step, then the second step is omitted, thus the changes are not reflected in the state of the object. The second step is so simple, that it can be implemented as an atomic action using a read-modify-write (RMW) or test-and-set (TAS) instruction found in most modern processors.

The serializability property means that atomic transactions must always be carried out according to a serial schedule, or in other words, there must exist a partial ordering of read and write events. This property also plays an important role in the multiple copy update problem, therefore serializability is inherent to the replica control protocols mentioned above: available copy replication employs the two-phase locking technique, while Replica Majority Commit contains a modified three-phase commit protocol.

Management of the replicated data

An ideal replication control protocol should guarantee the consistency of the replicated data in the presence of any arbitrary combination of non-Byzantine failures, while providing the highest possible data availability and requiring the lowest possible overhead. Data consistency in a replicated storage system means that the distributed processes accessing the stored information experience an identical view of the global state of the replicated data at any point of time. This notion incorporates two aspects: the *mutual consistency* of different copies, and the *internal consistency* of each copy. Copies of the data are mutually consistent if they are identical; since this is impossible to achieve for every instant of time this constraint is relaxed to require that multiple copies must converge to the same final state as all access activities cease.

Internal consistency of the data refers to the information content and involves both the *semantic integrity* of the stored object and the *atomic property* of the update operations. Semantic integrity stresses the need for the stored data to reflect accurately the state of the real world object it describes. Atomic transactions guarantee that the storage system reflects either none or all of the actions caused by a create/update transaction. Since the transaction is committed only if it does not violate semantic integrity, atomic transactions guarantee the internal consistency of the data, and so does any *serial schedule* of atomic transactions. Thus for a replicated storage with the possibility of concurrent transaction processing, mechanisms must be provided to generate serializable transaction schedules.

In the absence of network partitions the consistency of the replicated data can be ensured by the *available copy replication* (ACR) protocols [111]. These replication control protocols were designed to provide better fault tolerance characteristics than voting methods in environments that preclude partial communication failures. The operation of ACR protocols is based on: (1 imposing a total ordering on all writes so that all replicas receive these requests in the same order, (2) broadcasting these writes to all available replicas, and (3) requiring that replicas residing on nodes recovering from a failure remain unavailable until they are brought up-to-date. Using this mechanism read requests never need to access more than one available replica, because *all* available replicas are ensured to be valid and contain the latest version of the data. Furthermore, the replicated data can be accessed as long as there is at least *one* available replica.

The situation is quite different when network partitions (i.e., normally connected and logically coherent parts of a network are separated by a communication failure) must be taken into account [112]. This situation may occur in APEmille when two complete configurations are connected together to form an even more potent computing environment. A major limitation of the available copy replication strategy in this case is that it requires *reliable failure detectors.* As network failure detection is usually implemented by time-outs (which is an unreliable failure detection method), partitioning may lead to falsely suspect a correct but inaccessible process. For such a case we propose the use of the RMC (Replica Majority Commit) replication control protocol [113]. The RMC protocol was introduced to solve the *update majority* problem in the presence of unreliable failure detectors. The update majority problem is the task of updating a replicated object according to the majority voting strategy. Majority voting refers to a replication protocol where a read or write operation on a logical object must always access some majority of the replicas:

- On reading the object: the transaction accesses some majority of the replicas, chooses the one with the highest version number, and returns the contents of the selected replica.

- On writing the object: the transaction accesses some majority of the replicas, determines the highest version number in the majority, generates a new version number by increasing the highest version number, and updates all replicas in the accessed majority with this new version number.

The update majority problem consists for a set of replica managers to agree on the outcome of a transaction. A proper algorithm for solving the update majority problem has the following three properties: (1) *uniform agreement*: no two failure-free replica managers decide differently, (2) *non-blocking*: every failure-free replica manager that starts the protocol eventually decides, and *uniform validity*: every failure-free replica manager decides identically.

Bibliography

[1] F. P. Preparata, G. Metze, and R. T. Chien, "On the connection assigment problem of diagnosable systems," *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 848–854, Dec. 1967.

[2] T. Bartha and E. Selényi, "Probabilistic system-level fault diagnostic algorithms for multiprocessors," *Parallel Computing*, vol. 22, pp. 1807–1821, Feb. 1997.

[3] T. Bartha, "Effective approximate fault diagnosis of systems with inhomogeneous test invalidation," in *Proc. of the 22nd Euromicro Conference*, (Prague, Czech Republic), pp. 379–386, Euromicro, Sept. 1996.

[4] T. Bartha and E. Selényi, "Probabilistic fault diagnosis in large, heterogeneous computing systems," *Periodica Polytechnica*, 2000. Accepted for publication.

[5] T. Bartha and E. Selényi, "System-level diagnosis based on local test results," in *Proc. of the 1996 Mini-Symposium*, pp. 35–36, Dept. of Measurement and Information Systems, Jan. 1996.

[6] T. Bartha, *Rendszerszintű diagnosztikai algoritmusok multiprocesszoros rendszerekhez.* Dept. of Measurement and Information Systems, 1999. Mérési útmutató.

[7] T. Bartha and E. Selényi, "Efficient probabilistic diagnosis algorithms for large systems," in *Proc. of the 3rd European Dependable Computing Conf. (EDCC-3)*, (Prague, Czech Republic), Sept. 1999. Fast abstract and poster presentation.

[8] T. Bartha and E. Selényi, "On classification heuristics of probabilistic system-level fault diagnostic algorithms," in *Proc. of the Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS-2000)*, (Balatonfüred), Sept. 2000. Accepted for publication as full paper.

[9] T. Bartha and E. Selényi, "System level diagnosis of multiprocessors using local information," in *Proc. of the Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS-96)*, (Miskolc), pp. 183–190, Oct. 1996.

[10] T. Bartha and A. Pataricza, "Distributed system-level diagnostic algorithm," in *Proc. of the 14th Scientific Workshop*, (Budapest), Kandó Kálmán Technical College, May 1994.

[11] T. Bartha, A. Pataricza, and J. Altmann, "On integrating error detection into a fault diagnosis algorithm for massively parallel computers," in *Proc. of the IEEE Int. Symp. on Parallel and Distributed Systems (IPDS-95)*, (Erlangen, Germany), pp. 154–164, IEEE Computer Society Press, Oct. 1995.

[12] T. Bartha, A. Pataricza, and J. Altmann, "An event-driven approach to multiprocessor diagnosis," in *Proc. of the 8th Symp. on Microprocessor and Microcomputer Applications (μP'94)*, vol. 1, (Budapest), pp. 109–118, Budapest University of Technology and Economics, Oct. 1994.

[13] T. Bartha, "Diagnosis algorithms of multiprocessor systems," Master's thesis, Budapest University of Technology and Economics, Oct. 1993.

[14] T. Bartha and P. Maestrini, "Backward error recovery in the APEmille parallel computer," in *Proc. of the 3rd European Dependable Computing Conf. (EDCC-3)*, (Prague, Czech Republic), Sept. 1999. Fast abstract and poster presentation.

[15] T. Bartha, "A proposal for the recovery subsystem of the APEmille parallel computer," tech. rep., Istituto di Elaborazione della Informazione, Pisa, Italy, 2000. Accepted for publication.

[16] T. Bartha, "Utilizing backward error recovery to achieve fault tolerance in a SIMD supercomputer," in *Proc. of the SafeProcess '2000 Conference*, vol. 2, (Budapest), pp. 961–966, MTA SZTAKI, June 2000.

[17] T. Bartha, "A backward error recovery scheme for the APEmille parallel computer," in *Proc. of the 11th European Workshop on Dep. Comput. (EWDC-11)*, (Budapest), June 2000. Electronic publication (http://domino.inf.mit.bme.hu/EWDC-11.nsf).

[18] J.-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology," in *Proc. of the 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, (Ann Arbor, USA), pp. 2–11, IEEE Comput. Soc. Press, June 1985.

[19] M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, J. Honig, W. Hohl, E. Michel, and A. Pataricza, "Fault tolerance in distributed shared memory multiprocessors," *Lecture Notes in Computer Science*, vol. 732, pp. 31–43, 1993.

[20] D. Pradhan, ed., *Fault-Tolerant Computing, Theory and Techniques.* Englewood Cliffs: Prentice Hall, 1986.

[21] J. C. Laprie, ed., *Dependability: Basic Concepts and Terminology*, vol. 5 of *Dependable Computing and Fault-Tolerant Systems.* Wien: Springer, 1992.

[22] R. K. Iyer, "Experimental evaluation," in *Special Issue, 25th Int. IEEE Symp. on Fault-Tolerant Computing*, June 1995.

[23] S. Mallela and G. Masson, "Diagnosis without repair for hybrid fault situations," *IEEE Transactions on Computers*, vol. C-29, pp. 461–470, June 1980.

[24] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, pp. 56–78, Feb. 1991.

[25] L. Lamport, R. Shostak, and M. Pease, "Tha Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, June 1982.

[26] C. Kime, "System diagnosis," in *Fault-Tolerant Computing: Theory and Techniques* (D. Pradhan, ed.), vol. 2, ch. 8, pp. 577–626, Englewood Cliffs, N. J.: Prentice-Hall, 1986.

[27] A. D. Friedman and L. Simoncini, "System-level fault diagnosis," *IEEE Computer*, vol. 13, pp. 47–53, Mar. 1980.

[28] S. Kreutzer and S. Hakimi, "Adaptive fault identification in two new diagnostic models," in *21st Allerton Conf. on Communication, Control and Computing*, (Urbana, Ill.), pp. 353–362, University of Illinois, 1983.

[29] F. Barsi, F. Grandoni, and P. Maestrini, "A theory of diagnosability of digital systems," *IEEE Transactions on Computers*, vol. C-25, pp. 585–593, June 1976.

[30] M. L. Blount, "Probabilistic treatment of diagnosis in digital systems," in *7th IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 72–77, June 1977.

[31] J. Kuhl and S. Reddy, "Distributed fault-tolerance for large multiprocessor systems," in *7th Ann. Symp. on Computer Architecture*, pp. 23–30, 1980.

[32] A. K. Somani, V. K. Agarval, and D. Avis, "A generalized theory for system level diagnosis," *IEEE Transactions on Computers*, vol. C-36, pp. 538–546, May 1987.

[33] E. Selényi, *Generalized Theory of System-Level Diagnosis.* D.Sc. thesis, Hungarian Academy of Sciences, Budapest, 1985. In Hungarian.

[34] J. Maeng and M. Malek, "A comparison connection assignment for self-diagnosis of multiprocessor systems," in *11th IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 173–175, 1981.

[35] C. Kime, "An analysis model for digital system diagnosis," *IEEE Transactions on Computers*, vol. C-19, pp. 1063–1073, Nov. 1970.

[36] J. Russell and C. Kime, "System fault diagnosis: Closure and diagnosability with repair," *IEEE Transactions on Computers*, vol. C-24, pp. 1078–1089, Nov. 1975.

[37] J. Russell and C. Kime, "System fault diagnosis: Masking, exposure, and diagnosability without repair," *IEEE Transactions on Computers*, vol. C-24, pp. 1155–1161, Dec. 1975.

[38] S. H. Maheswari and S. L. Hakimi, "On models for diagnosable systems and probabilistic fault diagnosis," *IEEE Transactions on Computers*, vol. C-25, pp. 228–236, Mar. 1976.

[39] R. Gupta and I. Ramakrishnan, "System-level diagnosis in malicious environments," in *17th IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 184–189, 1987.

[40] S. Hakimi and A. Amin, "Characterization of connection assignment of diagnosable systems," *IEEE Transactions on Computers*, vol. C-23, pp. 86–88, Jan. 1974.

[41] G. Sullivan, "A polynomial time algorithm for fault diagnosability," in *25th Symp. on the Foundations of Computer Science*, pp. 148–156, 1984.

[42] V. Raghavan and A. Tripathi, "Improved diagnosability algorithms," *IEEE Transactions on Computers*, vol. 40, pp. 143–153, Feb. 1991.

[43] V. Raghavan and A. Tripathi, "Sequential diagnosability is co-NP complete," *IEEE Transactions on Computers*, vol. 40, pp. 584–595, May 1991.

[44] H. Fujiwara and K. Kinoshita, "On the computational complexity of system diagnosis," *IEEE Transactions on Computers*, vol. C-27, pp. 881–885, Oct. 1978.

[45] G. Sullivan, "An $o(t^3 + |e|)$ fault identification algorithm for diagnosable systems," *IEEE Transactions on Computers*, vol. 37, pp. 388–397, Apr. 1988.

[46] A. Dahbura and G. Masson, "An $o(n^{2.5})$ fault identification algorithm for diagnosable systems," *IEEE Transactions on Computers*, vol. C-33, pp. 486–492, June 1984.

[47] A. Dahbura and G. Masson, "A practical variation of the $o(n^{2.5})$ fault diagnosis algorithm," in *14th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 428–433, IEEE, 1984.

[48] S. R. McConnel, D. P. Siewiorek, and M. M. Tsao, "The measurement and analysis of transient errors in digital computer systems," in *9th IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 67–70, 1979.

[49] A. Pataricza, K. Tilly, E. Selényi, and M. Dal Cin, "A constraint based approach to system level diagnosis," Internal Report 4/1994., University of Erlangen-Nürnberg, Erlangen, June 1994.

[50] T. Bartha, A. Pataricza, P. Urbán, J. Altmann, and A. Petri, "Constraint based system-level diagnosis of multiprocessors," in *Proc. of the 2nd European Dependable Computing Conf. (EDCC-2)*, Lecture Notes on Computer Science, (Taormina, Italy), pp. 403–420, Springer-Verlag, Oct. 1996.

[51] C. Liaw, S. Su, and Y. Malaiya, "Self-diagnosis of non-homogeneous distributed systems," in *12th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 349–352, IEEE, 1982.

[52] S. Huang, J. Xu, and T. Chen, "Characterization and design of sequentially t-diagnosable systems," in *19th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 554–559, IEEE, 1989.

[53] A. Friedman, "A new measure of digital system diagnosis," in *5th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 167–169, IEEE, 1975.

[54] T. Bartha, *System-Level Fault Diagnosis.* TEMPUS Parallel Project, 1996. Electronic publication (http://mazsola.iit.uni-miskolc.hu/tempus/parallel/fault).

[55] M. Barborak, M. Malek, and A. Dahbura, "The consensus problem in fault-tolerant computing," *ACM Computing Surveys*, vol. 25, pp. 171–220, June 1993.

[56] T. Kameda, S. Toida, and F. Allan, "A diagnosing algorithm for networks," *Information and Control*, vol. 29, pp. 141–148, 1975.

[57] G. Meyer and G. Masson, "An efficient fault diagnosis algorithm for symmetric multiple processor architectures," *IEEE Transactions on Computers*, vol. C-27, pp. 1059–1063, Nov. 1978.

[58] K. Nakajima, "A new approach to system diagnosis," in *19th Ann. Allerton Conf. on Communication, Control, andComputation*, pp. 697–706, Sept. 1981.

[59] S. Hakimi and K. Nakajima, "On adaptive system diagnosis," *IEEE Transactions on Computers*, vol. C-33, pp. 234–240, Mar. 1984.

[60] S. Lee and K. Shin, "Probabilistic diagnosis of multiprocessor systems," *ACM Computing Surveys*, vol. 26, pp. 121–139, Mar. 1994.

[61] E. Scheinerman, "Almost sure fault tolerance in random graphs," *SIAM Journal of Computing*, vol. 16, pp. 1124–1134, Dec. 1987.

[62] D. Blough, G. Sullivan, and G. Masson, "Intermittent fault diagnosis in multiprocessor systems," *IEEE Transactions on Computers*, vol. 41, pp. 1430–1441, Nov. 1992.

[63] A. Somani and V. Agarwal, "Distributed syndrome decoding for regular interconnected structures," in *19th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 70–77, IEEE, 1989.

[64] A. Somani and V. Agarwal, "Distributed diagnosis algorithms for regular interconnected structures," *IEEE Transactions on Computers*, vol. 41, pp. 899–906, July 1992.

[65] S. Lee, *Probabilistic Multiprocessor and Multicomputer Diagnosis.* PhD thesis, University of Michigan, Ann Arbor, Mich., 1990.

[66] D. Blough, G. Sullivan, and G. Masson, "Efficient fault diagnosis of multiprocessor systems under probabilistic models," *IEEE Transactions on Computers*, vol. 41, pp. 1126–1136, Sept. 1992.

[67] A. Dahbura, G. Masson, and C. Yang, "Self-implicating structures for diagnosable systems," *IEEE Transactions on Computers*, vol. C-34, pp. 718–723, Aug. 1985.

[68] A. Dahbura, K. Sabnani, and L. King, "The comparison approach to multiprocessor fault diagnosis," *IEEE Transactions on Computers*, vol. C-36, pp. 373–378, Mar. 1987.

[69] S. Lee and K. Shin, “Optimal and efficient probabilistic distributed diagnosis schemes,” *IEEE Transactions on Computers*, vol. 42, pp. 882–886, July 1993.

[70] D. Fussell and S. Rangarajan, “Probabilistic diagnosis of multiprocessor systems with arbitrary connectivity,” in *19th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 560–565, IEEE, 1989.

[71] S. Rangarajan and D. Fussell, “Probabilistic diagnosis algorithms tailored to system topology,” in *21st IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 230–237, IEEE, 1991.

[72] S. Rangarajan and D. Fussell, “Diagnosing arbitrarily connected parallel computers with high probability,” *IEEE Transactions on Computers*, vol. 41, pp. 606–615, May 1992.

[73] M. Malek and Y. Maeng, “Partitioning of large multicomputer systems for effective fault diagnosis,” in *12th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 341–348, IEEE, June 1982.

[74] S. Hosseini, J. Kuhl, and S. Reddy, “A diagnosis algorithm for distributed computing systems with dynamic failure and repair,” *IEEE Transactions on Computers*, vol. C-33, pp. 223–233, Mar. 1984.

[75] S. Hosseini, J. Kuhl, and S. Reddy, “On self fault-diagnosis of the distributed systems,” in *15th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 30–35, IEEE, 1985.

[76] R. Bianchini and R. Buskens, “An adaptive distributed system-level diagnosis algorithm and its implementation,” in *21st IEEE Int. Symp. on Fault-Tolerant Computing*, pp. 222–229, IEEE, New York 1991.

[77] R. Bianchini and R. Buskens, “Implementation of on-line distributed system-level diagnosis theory,” *IEEE Transactions on Computers*, vol. 41, pp. 616–626, May 1992.

[78] M. Stahl, R. Buskens, and R. Bianchini, “On-line diagnosis of general topology networks,” in *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, (New York), pp. 114–121, IEEE, July 1992.

[79] D. Blough, G. Sullivan, and G. Masson, “Fault diagnosis for sparsely interconnected multiprocessor systems,” in *19th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 62–69, IEEE, 1989.

[80] P. Maestrini and P. Santi, “Self diagnosis of processor arrays using a comparison model,” in *Symposium on Reliable Distributed Systems (SRDS ’95)*, (Los Alamitos, Ca., USA), pp. 218–228, IEEE Computer Society Press, Sept. 1995.

[81] B. Andrásfai, *Graph Theory.* Polygon Könyvtár, Szeged: Polygon Kiadó, 1997. in Hungarian.

[82] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Comp. Sci. and Inf. Proc., Reading, Mass.: Addison-Wesley, 1975.

[83] D. Stauffer and A. Aharony, *Introduction to Percolation Theory.* London: Taylor and Francis, 1994.

[84] Parsytec Computer GmbH, Aachen, Germany, *Parsytec GC Technical Summary*, 1991.

[85] M. D. Cin and A. Pataricza, "Increasing dependability in multiprocessors," in *Proc. of the 8th Symp. on Microprocessor and Microcomputer Applications (μP'94)*, vol. 1, (Budapest), pp. 55–64, Budapest University of Technology and Economics, Oct. 1994.

[86] L. Lamport, "Time, clocks and the ordering of events in distributed systems," *Communications of the ACM*, vol. 21, no. 7, pp. 558–564, 1978.

[87] P. Ciompi, F. Grandoni, and L. Simoncini, "Distributed diagnosis in multiprocessor systems: The MuTeam approach," in *11th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 25–29, IEEE, 1981.

[88] S. Rangarajan and D. Fussell, "A probabilistic method for faul diagnosis of multiprocessor systems," in *18th IEEE Int. Symp. on Fault-Tolerant Computing*, (New York), pp. 278–283, IEEE, 1988.

[89] I. Fekete, T. Gregorics, and S. Nagy, *Introduction to Artificial Intelligence.* Budapest: LSI Oktatóözpont, 1990. in Hungarian.

[90] A. B. et al. (The APEmille Collaboration), "APEmille: A parallel processor in the teraflop range," internal report, INFN, Mar. 1995.

[91] F. Aglietti, A. Bartolini, C. Battista, and S. Cabasino, "The teraflop parallel computer APEmille," *Lecture Notes in Computer Science*, vol. 1225, pp. 991–998, 1997.

[92] F. A. et al. (The APEmille Collaboration), "An overview of the APEmille parallel computer," *Nucl. Instr. and Meth. in Phys. Res.*, vol. A 389, pp. 56–58, 1997.

[93] S. Chessa, B. Sallay, and P. Maestrini, "Diagnostic model and diagnosis algorithm of a SIMD computer," in *Proc. of the Third European Dep. Comp. Conf. (EDCC-3)*, pp. 283–300, Sept. 1999.

[94] B. Lampson, M. Paul, and H. Siegert, eds., *Distributed Systems - Architecture and Implementation. An Advanced Course*, vol. 105 of *Lecture Notes in Computer Science*, (New York, NY), Springer-Verlag, 1981.

[95] B. Sallay, P. Maestrini, and P. Santi, "A comparison-based diagnosis algorithm tolerating comparator faults," technical report, CNR - Istituto di Elaborazione della Informazione (Pisa), 1999.

[96] S. Chessa and P. Maestrini, "Correct and almost complete diagnosis of processor grids," technical-report, CNR - Istituto di Elaborazione della Informazione (Pisa), 1999.

[97] K. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[98] T. Bartha, "Rollback recovery in distributed systems," Tech. Rep. B4-12-06-98, Istituto di Elaborazione della Informazione, Pisa, Italy, 1998.

[99] J. Planck, *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton University, Jan. 1993.

[100] J. Plank, "Efficient checkpointing on MIMD architectures," Tech. Rep. TR-406-93, Princeton University, Computer Science Department, June 1993.

[101] G. A. Gibson, *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* PhD thesis, University of California at Berkeley, Dec. 1991. also available from MIT Press, 1992.

[102] D. Johnson and W. Zwaenepoel, "Sender-based message logging," in *17th Int. Symp. on Fault-Tolerant Computing (FTCS-17)*, (Washington DC, USA), pp. 14–19, IEEE Comput. Soc. Press, 1987.

[103] L. Alvisi, B. Hoppe, and K. Marzullo, "Nonblocking and orphan-free message logging protocols," in *Twenty-Third Int. Symp. on Fault-Tolerant Computing (FTCS-23)*, (Los Alamitos, CA, USA), pp. 145–54, IEEE Comput. Soc. Press, Aug. 1993.

[104] L. Alvisi and K. Marzullo, "Deriving optimal checkpoint protocols for distributed shared memory architectures," in *Proc. of the Fourteenth Ann. ACM Symp. on Principles of Distributed Computing*, (New York, NY, USA), p. 263, ACM, 1995.

[105] D. M. Blough, *Fault Detection and Diagnosis in Multiprocessor Systems.* PhD thesis, The John Hopkins University, Baltimore, 1988.

[106] T. Bartha, *Többprocesszoros rendszerek központosított diagnosztikájának vizsgálata.* Dept. of Measurement and Information Systems, 1994. Mérési útmutató.

[107] P. Behr, W. Giloi, and W. Schröder, "Synchronous versus asynchronous communication in high-performance multicomputer systems," in *Aspects of Computation on Asynchronous Parallel Processors*, pp. 239–247, North-Holland, 1989.

[108] M. Banatre, G. Muller, and J. Banatre, "Ensuring data security and integrity with a fast stable storage," in *Proc. IEEE Intl. Conf. on Data Eng.*, (Los Angeles, CA), pp. 285–293, Feb. 1988.

[109] B. Lampson, "Atomic transactions," in *Distributed Systems–Architecture and Implementation*, vol. 105 of *Lecture Notes in Computer Science*, pp. 246–265, New York, NY: Springer-Verlag, 1981.

[110] F. Cristian, "Reaching agreement on processor group membership in synchronous distributed systems," *Distributed Computing*, vol. 4, pp. 175–187, 1991.

[111] J.-F. Pâris and D. Long, "The performance of available copy protocols for the management of replicated data," *Performance Evaluation*, vol. 11, pp. 9–30, 1990.

[112] B. Parhami, "A multi-level view of dependable computing," *Computers & Electrical Engineering*, vol. 20, pp. 347–68, July 1994.

[113] R. Guerraoui, R. Oliveira, and A. Schiper, "Atomic updates of replicated data," in *European Depend. Comput. Conf. (EDCC-2)*, pp. 365–381, LNCS, 1996.

[114] T. Bartha and E. Selényi, "The mathematical model of integrated diagnostics," in *Proc. of the 1994 Mini-Symposium*, pp. 3–4, Dept. of Measurement and Information Systems, Jan. 1994.

[115] T. Bartha and E. Selényi, "Considering syndrome decoding as a classification problem," in *Proc. of the 1995 Mini-Symposium*, pp. 1–2, Dept. of Measurement and Information Systems, Jan. 1995.

[116] L. Harmat, *Self-Checking and Self-Diagnosis of Multiprocessor Structures.* C.Sc. thesis, Hungarian Academy of Sciences, Budapest, 1980. In Hungarian.

[117] G. Gruber, *Design of Multi-Microprocessor Self-Diagnosing Systems.* Doctoral thesis, Technical University of Budapest, Budapest, 1984. In Hungarian.