A CONSTRAINT-BASED ALGORITHM FOR SYSTEM LEVEL DIAGNOSIS

Diploma Thesis

ifj. Petri András

Technical University of Budapest

1994.

Acknowledgements

The author wishes to express his sincere gratitude to Prof. Dr. Mario Dal Cin for enhosting him at the Institute of Mathematical Machines and Data Processing at the Friedrich Alexander University of Erlangen-Nürnberg and to Dipl.-Math. Wolfgang Hohl for supporting the organizational aids of this diploma work.

Many thanks to Dipl.-Inf. Frank Balbach and Dipl.-Inf. Jörn Altmann who always found enough time for consulting and their valuable ideas and pieces of work greatly enhanced the completed task.

Special thanks to Prof. Dr. András Pataricza who made this study trip possible and constantly inspired the author to make his best (or even more).

This diploma work was carried out under the TEMPUS JET 3815/93. project.

Brief summary of the work

The latest years brought new ideas to the field of system-level self diagnosis. The growing practical importance of massively parallel multiprocessor architectures (consisting a few hundred or thousand computing elements) and the constant development of electronics made the application range of the traditional models and algorithms smaller.

Contrary to the traditional diagnosis models (like PMC, BGM etc.) which use strictly graph-oriented methods to determine the faulty components in a system, these new theories prefer AI-based algorithms for higher efficiency and greater flexibility. **Syndrome decoding**, the basic problem of self diagnosis, can be easily transformed to restrictions (*constraints*) between the state of the tester and the tested components, taking the test results into account. Well-elaborated tools from the field of AI can be used to find the possible results of these constraints, thus to find the possible fault state combinations of the system elements.

This approach has many advantages over the classical methods:

- applicability for **inhomogeneous** systems (built from different components with various test invalidation considerations) as well;
- the diagnosis algorithm itself can be practically *independent* from the actual **system topology** and the test invalidation model;
- the level of diagnosis can be **adaptively adjusted** after receiving the test results and extracting all the useful information from them.

Therefore, the diagnosis algorithm can be derived from a special constraint solving algorithm. The "benign" nature of the constraints (all their variables, representing the fault states of the components, have a very limited domain; thus the constraints are simple and similar to each other) reduces the algorithm's complexity so it can be converted to a powerful diagnosis method, suitable also for distributed diagnosis, with a minimal overhead. The primary goal of this diploma thesis was to evaluate these concepts in the practice.

An experimental algorithm was implemented for a Parsytec GC massively parallel multiprocessor system at the Institute of Mathematics and Computer Data Processing at the Friedrich Alexander University of Erlangen-Nürnberg. A simplified fault model was used, based on the available standard testing methods developed in the Institute [4][31]. The algorithm applied centralized diagnosis due to some useful special features of the Parsytec hardware architecture. The low-level testing mechanism ran on the transputers of the Parsytec system and the actual syndrome evaluation took place on a separate host machine, on a Sun workstation. It initiated the test sequence on the Parsytec, collected the syndrome bits and ran the CSP solver algorithm.

The Parsytec part of the diagnostic system was written in C, using the PARIX kernel. The syndrome decoding routines were implemented in ANSI C on a Sun SPARCstation under UNIX (SunOS 4.1.3) environment.

Table of Contents

CHAPTER 1. Introduction 6

1.1. Basic Terms and Principles of Dependability 6
1.2. Methods for Creating Dependable Systems 8
1.3. Classical Models and Algorithms in System-level Diagnosis 11
1.3.1. Fault-Test Relationships and Fault Models 11
1.3.2. Diagnostic Algorithms 18
1.3.3. Distributed Diagnosis 21
1.4. Problems in Traditional Methods 24
1.4.1. Requirements of a General Purpose Diagnosis Algorithm 26
1.5. AI-based Methods 27

CHAPTER 2. Constraint Satisfaction Problems 28

2.1. Formal Definitions 28
2.2. CSP Solution Methods 29
2.3. Consistency Algorithms 30
2.4. Similarity to the Self-diagnosis Problem 33
2.5. Ideas from an "AI-like" Traditional Algorithm 34

CHAPTER 3. Implementation Environment 35

3.1. Classification of Multiprocessor Systems 35
3.2. Hardware Overview 39
3.2.1. The Parsytec GC/GCel Machine 39
3.2.2. The INMOS T9000 Transputer 42
3.3. Software Environment 45
3.3.1. Programming Model 46
3.3.2. The PARIX Operating System Kernel 47
3.3.3. Development Tools 49
3.4. Differences between T9000 and T805 Transputers 50

CHAPTER 4. The Developed CSP-based Diagnosis Algorithm 52

4.1. Fault Model 52
4.2. Assumptions 55
4.3. Transformation into a CSP 56
4.4. Implementation Details 60
4.4.1. Low-level Testing Mechanism 60
4.4.2. CSP Solver 61

CHAPTER 5. Test Results 65

5.1. Measurement Methods and Considerations 65
5.2. Preformance Curves of Typical Test Runs 67

CHAPTER 6. Conclusions 69

6.1. Experiences 69
6.2. Future Work 70

Bibliography 72

APPENDIX: Program Source Listings ....

I. Low-level tester on the Parsytec transputers A-1
II. CSP solver on the Sun SPARCstation A-10
III. Common declarations A-37

1 Introduction

1.1. Basic Terms and Principles of Dependability

One of the basic expectations on a computer system is *dependability*: the behavior of a computer (the service it delivers) should be reliably identical to the specifications, from the viewpoint of another systems interacting with it (including human users). Dependability is a highly abstracted term and it strongly depends on the current application, as the expected service is different.

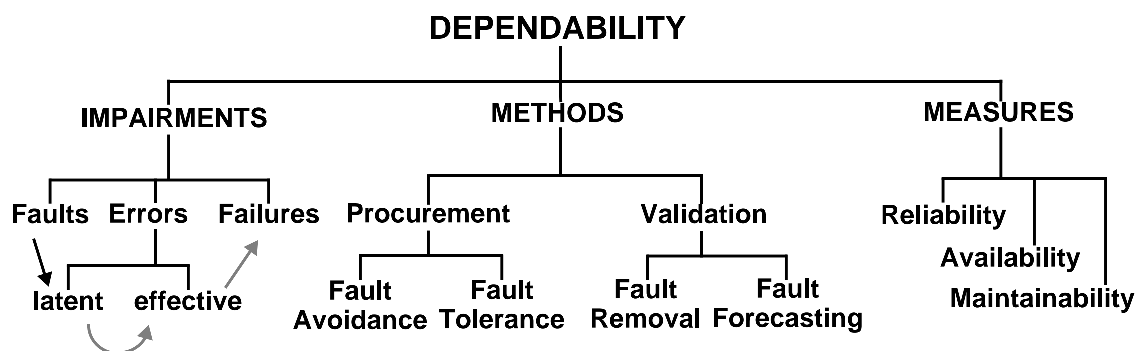


Figure 1-1. Terms of Dependability

Lack of dependability means that no reliance can be justifiably placed on the service of the system. These impairments are the following [30]:

- a *failure* appears when the actual service of the system is not identical to the expected (previously specified) service; this is the consequence of (one or more)
- *errors*: system states in which the possibility of a failure holds;
- *faults* are the phenomena in the system that are responsible for occurring of errors.

(An example: the pollution of the LED diodes in an opto-electronic mouse is the fault that causes an error of mouse cursor stopping; this effect can cause a failure in an application as its operation cannot be controlled by the mouse.)

Faults can be categorized into the following groups (fault classes):

- *physical* faults, results of physical phenomena within the system (internal) or its environment (external physical faults);
- *human* faults, committed during system design and implementation (design faults) or during operation and/or maintenance (interaction faults).

However, not each fault results in an immediate failure. An occurrence of a fault initiates a *latent* error that does not modify the behavior of the system; the error becomes *effective* only when it is activated and only an active error may cause a failure. (In the above example, if the mouse becomes faulty during a long calculation, it causes only a latent error that will be activated only when the user wants to use the mouse to control a further operation of the program. A failure occurs if the desired operation cannot be activated due to the erroneous mouse.) The actual latency of errors depends on the circumstances: the type of the fault, the system's utilization and other parameters.

Some effective errors still do not cause a failure due to the **redundant** construction of the system (redundancy can be *explicit*, i.e. previously designed specifically in order to avoid failure, or *implicit*, i.e. some unintentional features in the system may result similar failure avoidance as explicit redundancy) or the user's **unique definition** of failure (if the user can achieve its goal despite of the error occurred, he does not judge the situation as a failure; possibly he does not observe the error at all).

In the field of computer system diagnosis, from the two different basic approaches - the I/O oriented (*black box*) and the hierarchical (*white box*) concept - obviously only the second one can be used effectively. The 'black box' approach supports *fault pathology* only in a very limited extent. The basic rules governing the "life cycle" of faults, errors and failures on a system are the following ones:

1. A fault creates latent error(s) in the system component(s) where it appears. Physical faults can directly affect only the components in the physical layer; human faults can affect any components.

2. A latent error becomes effective when it is activated. An error can alternate between latent and effective state.

3. An effective error may (in the practice, usually does) *propagate* from one component to others; during propagation, *new errors* are created.
   Therefore an effective error in a component can be either a result of a latent error activation in that component or a side effect of an error propagation from another component(s).

4. A component failure occurs when an effective error changes its delivered service (responses to requests from interacting systems become different from the specification).

The basic measures of dependability are based on the concept that the operation time of a system can be divided to two parts: *service accomplishment* (when the system effectively delivers the specified service) and *service interruption* (when it does not). The system alternates between these states; the causes of the transitions are failures and repairs. The three basic measure qualifies the dependability by quantifying the state transitions:

- *reliability* is the measure of continuous service accomplishment (i.e. the time to failure from an initial reference state);
- *availability* is the measure of a successful service completion with respect to the alteration of accomplishment and interruption periods;
- *maintainability* is the measure of continuous system interruption (i.e. the time to repair).

As the processes leading to failure and repair are stochastic, these measures can be represented only as probabilities. Their exact mathematical definition is presented in [12].

1.2. Methods for Creating Dependable Systems

Constructing dependable systems requires a combined set of different design methods that can be grouped around two basic concepts: *fault procurement* to minimize the probability and/or seriousness of failures that can be resulted from faults, and *fault validation* to analyze the system layout for the possibility of faults and determine the confidence level of the system).

The main methods of fault procurement are:

- **fault avoidance** (prevention of fault occurrence by careful and systematic design and implementation);

- **fault tolerance** (providing the ability of the system to deliver the specified service even if errors have occurred, using built-in or explicit redundancy).

Fault validation includes the following methods:

- **fault removal** (checking for latent errors in the system and elimination of them so they cannot activate);
- **fault forecasting** (calculation of the probability and possible consequences of faults).

Fault tolerance has the greatest practical importance from this methods as this is the only *active* preventive action that operates during system operation; the other three methods have much more influence on the design and construction phase than the everyday running of the system. Moreover, fault tolerance can uphold (or at least reduce the loss of) dependability even in the case when the operational circumstances of a computer system has been altered since they were considered in design time.

Fault tolerance can be achieved by *detection* and *processing* of the errors already appeared in the system, desirably before they cause a failure, for preserving integrity of the system. Processing of effective errors is the more important task as these errors may cause failure in any moment. The possible methods of processing effective errors are:

- **error recovery**: replacement of the current (erroneous) state of the system with a previous, known error-free state, called checkpoint (*backward* recovery) or a new , presumably error-free state that never occurred before (*forward* recovery)[1];
- **error compensation**: implementing sufficient redundancy to the system component affected by an error that it could deliver its expected service, despite of the error.

Latent errors also have to be processed to prevent their activation. In the practice it means making them passive (i.e. excluding the erroneous component from the system) and changing system configuration to cope with the lack of some components.

When applying error recovery, the erroneous state of the system (and the faulty components themselves, if possible) need to be identified as soon as possible; it is the purpose of *error detection*. In the case of error compensation, errors are “hidden” from the other

1. Backward and forward recovery are not necessarily mutually exclusive; they can be applied together. (Backward recovery restores an earlier state of the system, therefore it decreases efficiency and may cause synchronization difficulties in cooperation of other components; forward recovery does not cause efficiency losses but careful damage investigations are required when finding the new system state.)

components and the user by *error masking*, so error detection is optional in principle (as long as the system components can comply with their specification, we can ignore the latent errors in them, at least those ones that were considered possible during the design process); however, omitting the error detection requires systematic application of error compensation which generally results an unwanted increase of redundancy and loss of efficiency. Practical implementations of error masking therefore contain error detection as well.

After detecting the erroneous components, the obvious way to handle them is their elimination from the system; this process is called *maintenance*. It can be *corrective* (removal of the errors that became effective and already has been processed by e.g. recovery) or *preventive* (immediate removal of errors in their latent state).

Fault procurement methods, however, obviously cannot protect the system from arbitrary error; they can achieve toleration of those error classes only which they were designed for. The tolerable error classes can be derived from the considered fault hypotheses. Moreover, fault tolerance can be implemented only by adding new components to the system so these components also need protection against errors, otherwise the fault-tolerant operation itself would be unreliable. Detailed description of reliable fault-tolerance devices (replicated voters, self-checking state observers, stable memory for recovery checkpoints etc.) is presented in [13].

The level of redundancy can be decreased if the system to be protected has special properties, e.g. some sort of structural regularity (error correcting codes, multiprocessors, special local area networks, simple robust data structures etc. [14]); in this case the tolerable error classes strongly depend on these system properties as the considered fault hypothesis is also based on these features.

Another important issue of implementation is the *time overhead* caused by error processing. This overhead determines the error latency and basically affects the system performance decrease resulting from applying fault tolerance. In the case of error recovery, the time overhead depends on whether an error is effective or latent (recovery from effective errors necessitates preparation of checkpoints; the finer the user time granularity is, the more checkpoints need to be used); when using error compensation, the time overhead is constant, and the duration of compensation is generally much shorter. In general, the relation between operational time overhead and the amount of redundancy is simple: the more redundancy is applied in the system, the less the overhead will be.

1.3. Classical Models and Algorithms in System-level Diagnosis

The first theoretical publications on the field of system-level diagnosis were published in the mid-sixties. From that time, numerous studies and experiments have been made, many implementations appeared and many practical experiences have gathered. In this section a general overview is given on the most well-known models and methods, especially paying attention to their known problems and limitations.

1.3.1. Fault-Test Relationships and Fault Models

A diagnostic model is a symbolic representation of erroneous states and/or components in a system and the methods used for detecting them. Many different fault models were developed in the last thirty years; they are usually unrelated to other models so composition of a general foundation for system diagnosis modeling is a complex task. A generalized model, presented by Russel and Kime in [15][16], covers all the existing diagnosis techniques and the contemporary approach as well. However, this model uses a slightly different terminology than the one described in Section 1.1 so the terms "fault" and "test" have to be used with different definitions.

The model is based on a hierarchical viewpoint: it considers a hierarchy of faults in a system. The cause of the erroneous behavior on the lowest level is a *physical defect* of an atomic component (e.g. a short circuit between the connections of a transistor). This defect is manifested as a *logical fault* at the line/device level (e.g. if the defective transistor is an input driver of a NAND gate, it causes a stuck-at-0 fault at the corresponding input line of that gate). This low-level fault results in the faulty operation of the NAND gate; it is a gate level fault. This fault may also result a higher level fault in the unit that contains it (e.g. an instruction decoder of a CPU) and so on (**Figure 1-2.**).

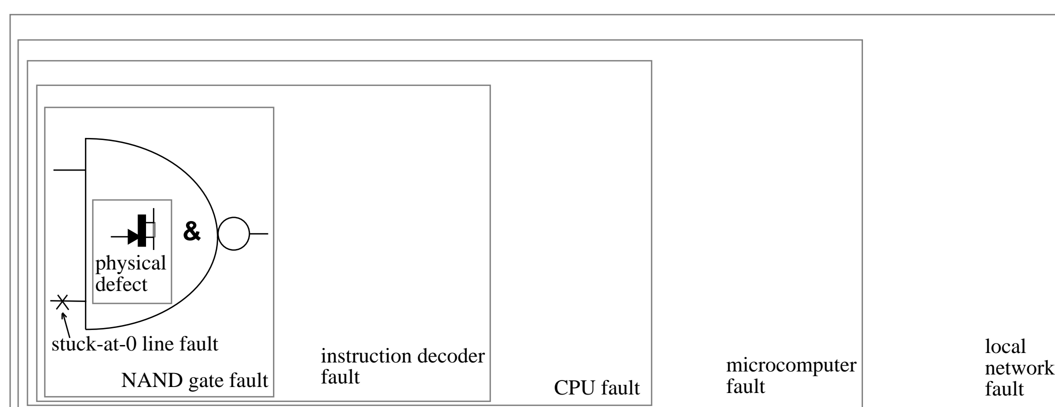


Figure 1-2. Levels of a fault hierarchy

A test in complete generality can be described as the application of some input sequence to a system element and observation of one or more of its output lines. It can take various forms, depending on the current component under test and the testing approach selected (e.g. single stuck-at tests of combinatorial networks can be tested by a single test pattern; a complex sequential network needs a sequence of test input patterns and the checking of the sequence appearing on all output lines.)

Obviously, tests can form a similar hierarchy like faults. The levels of the fault and test hierarchy correspond to each other. The following basic concepts define the correlation between the hierarchies:

- **Tests can be generated for any level in the fault hierarchy.** In the example system (**Figure 1-2.**), it is possible to test all lines and devices in the CPU logic for single stuck-at-0 and stuck-at-1 faults as a CPU is also a (high complexity) sequential network. This test detects faults on the lowest level of the fault hierarchy so it corresponds to the lowest level in the test hierarchy. On higher levels, a test program can be used to check the registers, addressing modes, instruction execution, ALU and other features of the CPU. It corresponds to the functional unit level (the fourth level on **Figure 1-2.**).
- **Certain collections of lower-level tests can be replaced by a higher level test.** If an extensive set of single stuck-at tests is applied to the CPU, it will reveal all the device/line level faults so the faulty CPU component(s) can be determined; the same result can be achieved, however, with tests generated at functional unit level.

  The inverse of this thought is also applicable: combination of the results of an appropriate extensive set of lower-level tests can give a higher level test information, as a higher level fault is considered a combination of lower level faults.
- **The test hierarchy corresponds closely to the diagnostic goal.** Fault detection within a microcomputer can be implemented on microcomputer level by generating pass/fail tests for the whole system. However, in many cases this testing approach does not qualify the faulty computer sufficiently; testing on lower, e.g. component level (separate tests for CPU, RAM, ROM and other components) provides better fault localization. Lower level test may generate detailed test results for more exact determination of faults; this information can be lost if the test results are combined for simulating a higher level test, and it does not appear at all if the testing was made on a high level.

  However, higher detail in test result does not always result better testing mechanisms; e.g. applying numerous stuck-at tests on a faulty CPU is generally useless (the knowledge of which elementary gate is faulty in a circuit containing a few hundred thousand gates is unnecessary). Therefore the level of testing must be adjusted to the diagnostic goals.

A test is *complete* for a set of faults if it fails in the presence of any single fault from the set and passes if none of the faults occur. Moreover, a complete test must detect all faults

on lower levels of hierarchy manifested. However, depending on the system architecture, some lower level faults may be undetectable by high-level tests. *Error coverage* represents the ratio of the faults detected by the test to all possible faults at the corresponding fault hierarchy level.

Error coverage can be increased by the application of low-level tests; this method, however, dramatically increases the complexity of the fault model. Due to limitations of data processing efficiency and/or human cognitive capability, the complexity of the fault model should be kept low, therefore complete testing (100% error coverage) hardly can be achieved in practice.

The set of possible faults is determined by the considered system granularity. The term of *fault pattern* is used for notation of the erroneous states in the system, including the possibility of multiple faults. A fault pattern F^i is a vector ($f_1, f_2, \dots ,f_n$) representing the faults present in the system; it is a subset of all potential faults. Similarly, the term of a *test pattern* can be applied as a subset of all the generated tests in the system (including failed tests). Vector $T^j = (t_1, t_2, \dots ,t_m)$ represents the actually used test pattern of $\boldsymbol{m}$ elements.

The relationship between faults and test patterns is described by the **fault pattern-test pattern event space**, which specifies the test pattern(s) applicable for each fault pattern. The event space can be generated from the fault model of the system: after determining the set of possible fault patterns from combinations of all considered faults (taking other features: failure rates, time interval between subsequent tests, error propagation assumptions etc. into account), the appropriate test patterns can be selected for each fault pattern.

Many different implementations exist for the representation of the event space [17]. The most obvious form is the array of vectors (**Table 1-1.**): rows of the array are associated with the fault patterns. The entries in a row correspond to the test results for the fault pattern.

This description form is quite redundant: one fault pattern requires multiple rows in the array; some tests may have indeterminate results (they can either pass or fail); some test patterns are correlated, have the same results. Therefore the vector array can be compressed into a tabular form (**Table 1-2.**) where one row represents a single fault pattern and the test

Fault pattern	Fault	Test pattern vectors				
		t_1	t_2	t_3	t_4	t_5
F^0	-	0	0	0	0	0
F^1	f_1	0	0	0	1	0
		0	0	0	1	1
		0	0	0	0	1
F^2	f_2	0	0	1	0	0
		0	1	0	0	0
F^3	f_3	1	0	1	1	0
F^4	f_4	0	1	0	0	1
		0	1	1	0	1
		0	1	1	1	0
F^5	f_5	0	0	1	1	0
		0	0	1	1	1
	f_6	1	1	1	1	0
		1	1	1	1	1
		1	0	1	1	1

Table 1-1. Fault pattern-test pattern event space in vector array form

results are compressed by using “don’t care” entries. In this case, every test pattern with n “don’t care” entries represents 2^n different patterns.

Fault pattern	Fault	Test pattern vectors				
		t_1	t_2	t_3	t_4	t_5
F^0	-	0	0	0	0	0
F^1	f_1	0	0	0	X	X
F^2	f_2	0	X	X	0	0
F^3	f_3	1	0	1	1	0
F^4	f_4	0	1	X	X	X
F^5	f_5	0	0	1	1	X
	f_6	1	X	1	1	X

Table 1-2. Event space in compressed tabular form

Fault pattern-test pattern relationships can be formulated as a Boolean expression as well. It can be derived either from the vector array form or directly from the behavior of the system. The transformation is quite straightforward: Table 1-1. is equivalent to the following expression:

$$\boldsymbol{S} = \bar{f}_1\bar{f}_2\bar{f}_3\bar{f}_4\bar{f}_5\bar{f}_6 \cdot \bar{t}_1\bar{t}_2\bar{t}_3\bar{t}_4\bar{t}_5 + f_1\bar{f}_2\bar{f}_3\bar{f}_4\bar{f}_5\bar{f}_6 \cdot \bar{t}_1\bar{t}_2\bar{t}_3 t_4\bar{t}_5 + f_1\bar{f}_2\bar{f}_3\bar{f}_4\bar{f}_5\bar{f}_6 \cdot \bar{t}_1\bar{t}_2\bar{t}_3 t_4 t_5 + \ldots$$

(1st row) (2nd row) (3rd row)

This standard form can be reduced to a simpler formula by use of “don’t care” values and of factorization:

$$\boldsymbol{S'} = \bar{f}_3\bar{f}_4\bar{f}_5\bar{f}_6\bar{t}_1 \cdot (f_1\bar{f}_2(t_4+t_5) + \bar{f}_1 f_2(t_2\bar{t}_3 + \bar{t}_2 t_3)) +$$

$$\bar{f}_1\bar{f}_2\bar{f}_5\bar{f}_6 \cdot (f_3\bar{f}_4 t_1\bar{t}_2 t_3 t_4\bar{t}_5 + \bar{f}_3 f_4\bar{t}_1 t_2(t_4\bar{t}_5 + \bar{t}_4 t_5)) +$$

$$\bar{f}_1\bar{f}_2\bar{f}_3\bar{f}_4 t_3 t_4 \cdot (f_5\bar{f}_6 t_1 t_2 + \bar{f}_5 f_6 t_1(t_2+t_5)).$$

The value of the Boolean expression is 1 if the given faults and test results occur in the fault pattern-test pattern event space and 0 if not.

The fault-test relationships can be represented by directed graphs as well. In this form the emphasis is not on the effective test results on the fault patterns but on other relations like test completeness for a fault and test invalidation by other faults. Directed graph-based representations are described in detail in [18]. The two basic types of graph models (for the example on **Table 1-2.**) are shown in **Figure 1-3**.

When creating fault models, three basic problems arise. The first is *test completeness*; this problem is generally ignored (all tests are assumed to be complete).

The second problem is *test invalidation*: the presence of certain faults may affect the results of tests aiming the detection of other faults. Therefore a test can be considered **valid** if it always fails in the presence of those faults which the test is aimed to detect, and always fails if those faults are absent. Hence in the practice some faults cause tests of other faults to perform incorrectly; in other words, occurrence of these faults **invalidates** the result of the corresponding test. This process must be included in a reasonable fault model.

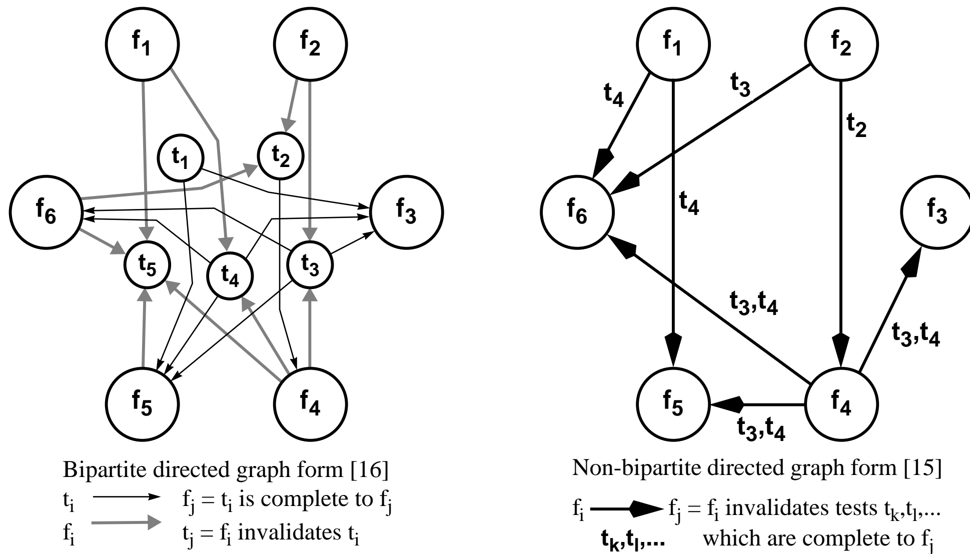


Figure 1-3. Directed graph representations of fault-test relationships

The third problem is the requirement of a proper *compromise* between model complexity and detail level of test results. A simplification in the fault model generally causes loss of information; in our example, tests t_2 and t_3 for fault pattern F^2 can result in [0,1] or [1,0] but their combined results appear as [X,X] instead of $[X_2,\overline{X}_2]$ in the compressed tabular form (see **Table 1-1.** and **Table 1-2.**) Correlated tests may also be uncovered; in **Table 1-1.**, t_2 and t_5 seem to be correlated in the original form and this knowledge is lost during table compression; moreover, incorrect test patterns (e.g. $[t_2=1, t_5=0]$) appear. However, simplification is often necessary due to efficiency limitations.

	Fault pattern	Fault	Test pattern vectors				
			t_1	t_2	t_3	t_4	t_5
Original test pattern vectors	F^5	f_5	0	0	1	1	0
			0	1	1	1	1
Compressed form	F^5	f_5	0	**X**	1	1	**X**

Table 1-3. Effect of a correlated test pair

The most widely known fault models are the following:

- **Preparata-Metze-Chien (PMC) model** [19]. This was the first published conceptual test invalidation model in 1967; however, it is still used. The model involves system components (*units*) capable to perform tests on another units. The units and the test results are considered as binary, pass/fail-type ones. Hence every working unit per-

forms and evaluates tests independently from other units, each test is complete for a single fault (the fault of the unit under test) and each test is invalidated by exactly one fault (the fault of the tester unit). Test results are collected and decoded by a fault-free central observer (*hard-core*).
The applied test invalidation scheme is *symmetrical*: a fault-free tester determines the state of the tested unit correctly, but a faulty tester can produce arbitrary results, independently of the actual state of the unit under test. This assumption is the most pessimistic approach as it does not include any knowledge about the operation of a faulty tester unit.
Terms are introduced for describing diagnosability of the system:

- ***t**-fault detectability*: if at least one tests will definitely fail in the presence of at least one but at most ***t*** faults, the system is ***t***-fault detectable;
- *one-step **t**-diagnosability*: if every faulty units can be unambiguously determined after evaluating the test results, considering at most ***t*** faulty units, the system is one-step ***t***-diagnosable (***t***-diagnosable without repair);
- *sequential **t**-diagnosability*: if at least one faulty unit can be unambiguously determined after evaluating the test results, considering that the number of faulty units in the system is maximally ***t***, the system is sequentially ***t***-diagnosable (***t***-diagnosable with repair). The term "sequential" refers to the obvious maintenance method using this kind of diagnosability: complete repair is achieved by subsequently replacing the faulty units found in the last test round and repeating the tests.

Diagnosability analysis can be performed on the system to determine the effectiveness of a given test set on the actual system structure, i.e. to find the maximal ***t*** values for ***t***-fault detectability (for pass/fail checking of the whole system), sequential ***t***-diagnosability (to find some faulty units) and one-step ***t***-diagnosability (for complete repair). Sufficient and necessary conditions are given for ***t***, the number of units and the number of tests in the system in [18] and [19].

PMC model is the simpler and most general test invalidation model; many of the later models are based on it.

- **Barsi-Grandoni-Maestrini (BGM) model** [27]. This model uses generally the same considerations as the PMC model. The basic difference is the *asymmetrical* test invalidation: faulty units are always tested as faulty, independently from the state of the tester unit. It is a more optimistic (and in many cases, more realistic) assumption; it makes mathematical formulation of diagnosability conditions easier so algorithms based on the BGM model often result better diagnostic.

- **Maheshwari-Hakimi model** [28]. Probabilistic elements are included into this model to cope with test completeness problem and give better description of the reality. It involves units with statistically independent failure rates. A probability p(f_i) is assigned to each fault f_i; the term *probabilistic **t**-diagnosability* is introduced. It means that each test patterns has one or more associated fault patterns F^i with p(F^i) > t (t is a real number in the [0, 1) range). Diagnosability analysis can be performed in this case as well.

  The greatest problem with probabilistic models is that the p(f_i) a priori fault probabilities cannot be previously calculated in the practice: exact measures can be taken only from experimental results and inexact preconceptions may provide totally incorrect diagnostic results; therefore these models have little practical importance.

- **Russel-Kime (RK) model** [15][16]. In their presented general fault model, Russel and Kime introduced a new approach of the system diagnosis problem. Their model involves a wider range of fault-test relationships, not only test invalidation. Instead of system units and tests performed by them, just test and faults are involved. This method allows the inclusion of some possibilities: multiple (cooperative) testing, multiple invalidation (a test can be invalidated by more than one fault), hard-core tests (not invalidated by other faults) etc.,that caused problems in other models.

  Diagnosability analysis is also performed from new viewpoint in this model. The theoretical basis of the analysis is the term of **closed fault pattern**: F^i is closed if no complete test for any fault in F^i exists when F^i appears in the system. Conditions for t-fault detectability and sequential t-fault diagnosability (in the meaning used in the PMC model) can be formulated from determining the cardinality of the smallest closed fault pattern in a system (called *closure index*). An additional term, **exposed fault** (a fault in a fault pattern F^i is exposed if one or more complete tests exist for f_i in the presence of F^i) makes conditions for one-step t-fault diagnosability possible by determining the minimum of the number of exposed faults in the fault patterns (*exposure index*).

1.3.2. Diagnostic Algorithms

System diagnosis is generally consisting of two tasks: generation and execution of tests according to a selected fault model, and evaluation of the collected test output patterns (*syndromes*) to determine and locate the fault(s) in the system. This second task is called *syndrome decoding*; mathematical methods for performing it are diagnosis algorithms.

The simplest diagnosis algorithms are based on the fault pattern-test pattern event space representation. If the actual result of the tests is known, the event space can be searched for fault patterns whose corresponding test pattern matches to the syndrome, so the decoding is transformed to a pattern matching problem. However, if a syndrome matches with multiple fault patterns, there is no way to identify the actual fault(s) without performing additional tests; therefore this **table look-up** algorithm supports directly only fault detection; for fault diagnosis, it must be extended with hierarchical test initiation.

The pattern matching can be performed on an “inverse” of the event space table (*fault dictionary*, containing the possible syndrome values and the corresponding fault patterns) or a tree-based data structure built from the tables (see **Figure 1-4**).

Syndrome detection can also be performed by evaluating the Boolean expression form of the event space. As this expression is built up from the fault values and test results, subsequent substitution of the actual test outcomes will result a simplified expression that contains only those fault patterns that matches to the syndrome. The evaluation can be finished if the expression is reduced to a single product term; the actual fault pattern is the combination of fault values that result the value 1 of the product term.

Syndrome values				Possible faults
t_1	t_2	t_3	t_4	
0	0	0	0	none
0	0	0	1	f_4
0	0	1	0	f_2
0	0	1	1	f_1, f_2 or f_4
0	1	0	0	f_3
0	1	0	1	f_3 or f_5

Table 1-4. Detail of a fault dictionary

Figure 1-4. Tree form of Table 1-4.

The common problem with the above mentioned algorithms is the very low efficiency. Testing of a large-scale system can involve hundreds of faults and thousands of tests; storage and maintenance of huge fault-test tables, fault dictionaries or tree structures limits the applicability range seriously. Similarly, evaluating Boolean expressions results efficiency decrease above a level of complexity; moreover, it requires symbolic data manipulation. Use of tree structures has the additional drawback that the operation of the decoding algorithm depends on the processing order of syndrome bits, so the average time complexity of searching also decreases.

Syndrome decoding can be performed on the graph-based representation of fault-test relationships as well. A basic idea is **bootstrap** testing: knowing the invalidation conditions of tests, those tests are performed first that are not invalidated (hard-core tests). Then the

proven fault-free units test others etc., thus ensuring that any unit is proved to be fault-free before it tests another units. Therefore no test will be invalidated and every failed test will actually discover one or more faults. This method is simple and fast, but obviously applicable only if the graph model of the system does not contain directed cycles (a test is invalidated by a potential fault that is detected - directly or indirectly - with the same test).

A more general diagnosis algorithm for PMC and BGM models, based on bootstrap strategy, was developed by Meyer and Masson [29], supporting one-step t-fault diagnosability. A fault table is built by each unit; the table contains the assumed fault state of other units. Every unit supposes itself as good and the other units as uncertain, then executes tests. The results of the tests are recorded in the table. Then the updated tables are sent to all units that passed the test, tables from those units are received and merged with the local table. The table exchange process is repeated until no new information is gained from an exchange or the t limit of faults in a table is exceeded. The still undetermined units in the tables are recorded as fault-free. After building up tables, an unit is assumed fault-free if it is recorded faulty in no more than t tables, otherwise it is assumed faulty.

A very promising algorithm, allowing arbitrary test invalidation rules, was presented in [2]. The algorithm applies two working phases:

- first all the available information is collected from the system, and some units are qualified by inference. Implications are driven by the actual syndrome and knowledge of the system structure;
- then the fault state of units unqualified in the previous phase is determined, assuming one-step t-fault diagnosability.

The idea behind the use of implications is that some units can be qualified without any further assumptions, merely checking the implication chain for contradiction; if an assumption on the fault state of a unit leads to a contradiction then the assumption is wrong and therefore the fault state can be unambiguously determined.

One-step implication rules between the state of the tester and the tested units are created straightforward from the actual test invalidation model. Two- and more-step implications can be performed by repeating one-step implications subsequently; the final state of implication chain is achieved by computing the *transitive closure* of the implication set. A contradiction occurs if a unit implies the <u>opposite</u> of its own initially assumed fault state during the implication process.

In [2], one-step implication rules are represented by a 2×2 hypermatrix (every element is an $M_{n \times n}$ matrix (n is the number of units in the system), m_{ij} indicates the possibility of one-step implication from the state of unit u_i to the state of unit u_j; the four matrices store the good→good, good→faulty, faulty→good and faulty→faulty one-step implications). After finishing implications, the hypermatrix contains all information about the system that can be extracted from the syndrome without further restrictions.

The great advantage of the algorithm is the independence from the applied test invalidation restrictions as well as the actual system topology. These properties affect only the generation of the initial one-step implication matrices.

1.3.3. Distributed Diagnosis

In the previous sections, collection of test results and application of the diagnosis algorithms were assumed to be performed by a separate unit outside the system, a *central observer*. This approach requires further considerations:

- the operation of the central observer is always correct (no faults occur in it);
- test results can be delivered to the observer reliably, independently from the faults in the system;
- the observer has facilities to maintain the system (i.e. to reconfigure and/or repair faulty units).

Systems using a central observer as diagnostic device are called *centralized systems*. These systems usually contain a highly reliable[1] set of units (also called **hard-core**), dedicated exclusively to perform diagnosis and initiate recovery and/or repair actions. This observer has a communication connection to each unit in the system, dedicated to transfer test results and maintenance action messages.

However, the central observer concept arises serious problems in large-scale systems. A higher number of units requires more tests to generate and more syndrome pieces to evaluate so the time and space complexity of the algorithms running on the central observer increases dramatically. The diagnostic overhead also increases due to the limited bandwidth

1. “High reliability” refers to reliability that is reasonably higher than the reliability of other units in the system (an usual requirement is the failure rate being at least by an order of magnitude lower). It can be achieved by using special, better quality components, careful design and self-checking structures.

of the communication links between the observer and the units and the sequential processing of test results. In scalable systems, where the architecture is expected to be expandable by additional units for greater performance, the central observer may get overloaded. Finally, the reliability of the whole diagnosis is dominated by the reliability of the central observer; its failure may collapse the system or even introduce “malicious” behavior (e.g. the diagnosis may finish detecting some faults, or may randomly signal non-existing ones).

These problems require a completely different approach of system diagnosis. Instead of concentrating the diagnosing capability into a single central device, monitoring and maintenance capabilities should be distributed among all system components. Thus the limitations resulting from centralization can be eliminated while preserving the diagnostic capability.

The concept of *distributed diagnosis* obviously arises other problems. Without a centralized communication network dedicated to diagnostic purposes, all the diagnostic information must transferred via the (potentially faulty) normal interprocessor communication links. Moreover, direct information can be obtained only from neighboring units. Data from any other unit must be considered as uncertain unless reliable information exists on the source of data and the route towards the receiving unit.

Therefore each unit must be able to make localized decisions about the fault state of other units. Moreover, each unit can take only local maintenance actions, also via normal communication facilities. Cooperative maintenance (faulty unit removal) actions, even supported by additional hardware, cannot be executed due to the autonomy of the units and limitations in the communication; similarly, a single unit (or a single external hardware element) cannot be allowed to take complete removal actions as it would make the system sensitive of the fault of that element, despite of the distributed nature of the diagnosis.

The first usable concept of distributed diagnosis was *distributed fault-tolerance* proposed by Kuhn and Reddy [20][21]. The aim was to find a method for isolating faulty units from the system without centralized maintenance. If each fault-free unit can build a correct diagnostic image of the system independently from other units, it can simply cancel any further interaction with the faulty elements so they will be unable to affect the operation of the system. The proposition prefers the self-test of the units. A complete test of a unit by another unit is hardly possible via normal communication links (especially if the units are highly complex structures, e.g. complete microprocessors) and causes enormous commu-

nication overhead. The incorporation of fault-tolerant capability into each unit eliminates the need of intensive communication; moreover, internally fault-tolerant processors can mask or remove some faults without external testing. Thus the number of faults that must be treated by the diagnosis algorithm can be reduced.

Tests in this scheme appear as simple checks on the other unit's fault-tolerant equipment (e.g. a dedicated watchdog processor). This structure requires only a minimal overhead. The fault-tolerant equipment itself still needs to be tested but its complexity is much lower than the complexity to a whole unit. Results of local tests (local syndromes) are sent to other units via the normal communication network; faulty units are allowed, however, either to send erroneous local syndromes or alter diagnostic information routing through them (including the case of not forwarding the incoming messages): the model applies symmetric (PMC) test and message invalidation.

A new term: ***t**-fault self-diagnosability* was introduced. A system is ***t***-fault self-diagnosable if every fault-free unit can determine all the faulty units, providing that the number of faults in the system does not exceed ***t***. Conditions for ***t***-fault self-diagnosability can be calculated from the connectivity of the testing graph (the directed graph describing which units test which other units) [18].

The diagnosis algorithm of the model (named SELF) is similar to the Meyer-Masson algorithm: a fault vector is generated by each unit, containing the fault states of the other units. The fault domain, however, contains three values: <u>good</u>, <u>faulty</u> and <u>unknown</u> (hence the non-neighbor nodes can be qualified only from indirect information). The initial state of the fault vector is also similar: the node supposes itself as good and all the other nodes as unknown. After performing local tests, the results are recorded in the vector and the updated vector is distributed to the neighboring units; vectors from neighbors tested as good is accepted and incorporated to the local fault vector. The process iterates until no more "unknown" records stay in the local vector or ***t*** faulty units are detected; in the latter case the remaining "unknown" units are assumed to be fault-free. The algorithm is based on the assumption that no further faults occur during the information exchange process, thus the analysis of the units reflects the actual and valid state of the system.

A more general model for distributed diagnosis was presented by Holt and Smith [22]. It introduces the concept of **diagnostic information** (that must be produced during the diagnosis process) and **diagnostic sinks** (sets of units that must obtain diagnostic information

and act upon it); this approach allows the description of a wider range of diagnostic goals. From the various possible aims, two of the most commonly used goals are included in the model:

- **diagnosis with repair**: the aim is the determination of at least one faulty unit in the system, and reconfiguration for the exclusion the faults. In the original model, reconfiguration is performed by a special device (*diagnostic controller*). This approach contradicts to the basic concept of distributed fault tolerance as it involves a hard-core into the system. Here the diagnostic information is the identification of the faulty unit(s) and the diagnostic sink is the controller (or a fault-free controller if more separate controllers are applied);
- **graceful degradation**: the aim is the determination of a maximal set of fault-free units that can continue operation, instead of fault detection. In this case, both the diagnostic information and the diagnostic sink is the set of units that are guaranteed to be good. Reconfiguration (exclusion of faulty units) is performed by "intelligent" units of the system (*diagnostic analyzers*); unlike controllers, analyzers are responsible for maintaining only a local part of system. The exclusion algorithms is generally ignore the faulty units because in order to avoid applying additional hard-core hardware circuitry. In the case of graceful degradation, a limit of faulty units as a traditional measure of diagnosability cannot be used anymore. *Surviving curve* is introduced instead; it describes the minimal number of active fault-free units in the system as the function of the number of faults. Obviously, the graceful degradation continually decreases the size of the operational part of the system until it becomes unable to deliver the expected service. The base of measure is to determine this point.

1.4. Problems in Traditional Methods

The majority of the methods described in Section 1.3 were developed quite a long time ago. At that time of development, practical computer systems were significantly smaller and simpler, built from lower quality components. Therefore there methods reflect the level of technology of an earlier time.

Nowadays, however, the development in the field of electronics and computer hardware technology has dramatically speeded up. New systems, components and architectures outdate or even invalidate the implicit considerations included in the earlier models. Moreover, the latest achievements for increasing computational power and system dependability, especially massively parallel systems, have a much greater complexity than other traditional systems.

Traditional methods and algorithms have several serious problems when applied to modern computer systems:

- The *test invalidation* assumptions used in the models are *too pessimistic*. Development in electronic circuitry technology produces computer components with lower fault rates; sophisticated design and production methods can assure a much more deterministic, and thus predictable behavior of a faulty component than earlier system. Therefore simple test invalidation schemes (like PMC) cannot describe real operation of a component with sufficient precisity.

- The other consequence of the lower permanent fault rates of components is the increased practical importance of *transient faults*. The majority of existing fault models handle only permanent faults or at least assumes a longer fault duration than the time needed for testing. In current systems the rate of transient faults is typically by one order of magnitude greater than permanent faults.

- As many methods are derivatives of, or based on the oldest PMC model and test invalidation, they preserved many limitations from it. One of the most important limitations is the requirement that one-step t-fault diagnosability needs at least t other units to test each unit. This approach comes from a concept of a theoretical system, with arbitrary (practically fully connected) system topology.
  This requirement becomes a serious drawback in modern multiprocessor systems, as they apply a *simple physical interconnection topology* (generally a 2- or 3-dimensional mesh) in order to support easier system scalability. Therefore the maximal number of detectable faults in such systems becomes 2 or 3 (units in the corners of the structure has only 2 or 3 neighbors) so the diagnostic power of the traditional algorithms is strongly reduced.

- The use of an *unified test invalidation model* seems to be a reasonable idea for the simplification of the mathematical treatment of the model and the construction of the algorithm. It implicitly assumes that all units in the system have identical diagnostic properties so the system is homogeneous. In the practice, however, *inhomogeneous systems* have a growing importance. Uniformization of the test invalidation models for every unit in an inhomogeneous system looses the detailed **a priori knowledge** about the behavior of different units.

- The *time complexity* of the traditional algorithms are far *too high*, causing unacceptable diagnosis inefficiency. The algorithms were originally developed for systems containing at most only a few tens of units. Modern massively parallel multiprocessor systems, however, contain several hundreds or thousands of processing units.
  The worst-case time complexity of diagnosis algorithms is generally exponential, as the diagnosis problem in its whole generality is NP-complete. Practical algorithms are expected to show good average time complexity. Many of the available algorithms, unfortunately, become inefficient rather quickly when the number of units increases.

- The *level of diagnosis*, i.e. the amount and type of providable diagnostic information is determined strictly from the system structure at design time, *before* the actual syndrome information is received. This property prevents effective information extraction and makes involving *adaptivity* to the algorithm difficult. It should be desirable that the level of diagnosis could be adapted to the syndrome information available and not vice versa.
- The majority of the algorithms starts syndrome decoding after receiving all the test results. After decoding, diagnostic results are sent back in one burst. Both data transfers require a long time and potentially cause local communication overloads. Moreover, the time needed for the collection of syndrome bits decreases the efficiency of the diagnosis algorithm as well. It should be desirable that the diagnosis algorithm could supply diagnostic information as soon as it obtains new syndrome data (*diagnosis on-the-fly*), thus the average efficiency of the diagnosis increases, with a simultaneous decrease of communication overhead.

1.4.1. Requirements of a General Purpose Diagnosis Algorithm

The requirements of a diagnosis algorithm usable on a wide spectrum of practical systems, can be concluded quite straightforward from the problems described in the previous subsection:

- applicability in *inhomogeneous* systems as well, with different units and test invalidations; ability to utilize *a priori knowledge* about the system in full extent;
- ability to handle *both permanent and transient* faults;
- effective *information extraction* for arbitrary topologies, i.e. the algorithm should estimate not only the usual GO/NO GO type qualification for systems and its elements in the special cases of restricted architectures, but in the very general case even *the level of diagnosis* (one-step, sequential diagnosis or only error detection) should be *adaptively estimated* on the basis of the test results;
- use of *efficient mathematical apparatus* with algorithms of low average time and reasonable space complexities, even *for a large number of* (several thousand) *processing elements*;
- (optional) ability to provide *diagnosis on-the-fly.*

These requirements are only partially fulfilled by existing diagnosis algorithms. Moreover, the efficiency of these algorithms can be hardly increased, due to the limitations of the traditional mathematical treatment.

System level diagnosis, however, is not the single field where the necessity of finding algorithms with good average time complexity for generally NP-complete problems arise. These class of problems can handled effectively in many cases by AI-bases algorithms.

1.5. AI-based Methods

The main intention of “artificial intelligence” (AI) methods is to find efficient solutions for difficultly solvable (to be more precise, generally NP-complete) or hard to represent problems. This gives a way to handle many practical but earlier unmanageable problems.

This aim is frequently reached by more sophisticated information management; it is often called “knowledge management” as it represents a high level of abstraction and provides more flexible and efficient information extraction from elementary data. This approach introduces certain symbolic operations, like inference and deduction. This is extremely advantageous if the traditional quantitative methods start to exhaust and prove insufficient for that particular purpose.

Many well-elaborated, efficient and practically tested AI-based algorithms have been developed over the years. A group of them, the CS (Constraint Satisfaction) methods seem especially useful for system level (self-)diagnosis models, as it shows a deep similarity with some existing diagnostic approaches (see Section 2.4 and 2.5).

Application of CS methods in system level diagnosis has appeared as a new idea in the most recent years. Its practical attractiveness has already proven in closely related fields, like automated test pattern generation [3], system safety and risk analysis, etc. These facts motivated primarily the idea of developing a constraint-based diagnostic algorithm.

2 Constraint Satisfaction Problems

A short overview about CSPs and their mathematical apparatus is presented in this chapter, based on [3].

2.1. Formal Definitions

A *constraint satisfaction problem* (CSP) can be formulated as an (X, D, C) tuple where $X = \{X_1,X_2,... X_n\}$ is a set of *variables*, $D = \{D_1,D_2,... D_n\}$ is a set of *domains* (each domain is a set associated with a variable and contains the allowed values of that variable) and $C = \{C_1,C_2,... C_k\}$ is a set of *constraints*. Constraints are *relations* between domains of variables, i.e. they are subsets of the Cartesian product of the affected variables' domains ($C_i \subset D^* = D_p \times D_q \times ... \times D_z$). They represent the allowed value combinations of the affected variables.

A *solution* of a CSP is a vector $x = [x_1,x_2,... x_n]$ of values that satisfies all the constraints (all the constraint relations hold if we substitute the variables with the subsequent values from x). The *constraint satisfaction problem* itself is to find *one solution* or *all solutions* of a given CSP.

CSPs can be represented by a G(X,C) network where the elements of X are represented by the *nodes* and the elements of C by the *edges* of the network. In a special subclass of constraint satisfaction problems (called *binary* CSPs) every constraint affect at most two variables so the network is a simple graph; in the general case, however, the CSP network is a hypergraph. *Loop edges* represent unary constraints (affecting only one variable), *multiple edges* are different constraints affecting the same variables.

Binary CSPs are the most “benign” class of constraint satisfaction problems, i.e. they can be handled with the simplest mathematical apparatus.

A CSP is *discrete* if every D_i has enumerable cardinality, and *continuous* if some D_i-s have continuously infinite cardinality. (Restricting the term of discrete CSPs to those cases when every D_i is a finite set is quite common. The solving algorithms for CSPs with enumerably infinite domains are similar to the methods used for continuous CSPs[1] and radically differ from the algorithms for CSPs with finite domains.) In the field of system-level diagnosis, only discrete CSPs are used as the systems and components are supposed to have a finite number of states.

The CSP is *static* if both the constraint network topology and the constraints themselves are fixed and *dynamic* if they can change during the search for solutions.

2.2. CSP Solution Methods

Solving discrete CSPs is proved to be NP-complete [5], so simple exhaustive algorithms cannot be used to generate all the variable value sets and to select the solutions. Intelligent backtracking algorithms (backjumping, conflict-based backtracking, forward checking etc.) must be used [9][10]; they offer much better average time complexity (their worst-case complexity, however, is still exponential).

We can assume for simplicity without loss of generality that each of the n variable in the CSP has a discrete domain with the same cardinality so the search space $D^* = D^n$. Therefore the worst-case complexity of a trivial exhaustive generate-and-test algorithm is $O(d^n)$ where $\boldsymbol{d}$ is the cardinality of the domains. The complexity can be obviously reduced by decreasing $\boldsymbol{d}$ or $\boldsymbol{n}$.

Decreasing of $\boldsymbol{n}$ is possible only if the CSP contains variables (e.g. the value of a variable can be calculated unambiguously from the value of another variable); it never appears in properly designed CSPs. Decreasing of $\boldsymbol{d}$ can be achieved by preprocessing the problem before starting the solution algorithm; these methods are called *consistency algorithms* [5][6][7]. Consistency refers to the elimination of locally inconsistent value combinations

1. Solving continuous CSPs shows similarity to solving special equation sets; they can be handled by numerical analysis methods [11].

from the variables' domains, as they surely cannot participate in a globally consistent (correct) solution.

Consistency algorithms even reduce the number of "fruitless" backtracks made every time when a locally inconsistent value is found. They work generally only on binary CSPs because every variable in such CSPs can be evaluated independently. Moreover, a subsequent evaluation of the current value of a variable and its neighbors is always sufficient to achieve global consistency. Therefore problem transformation to a binary CSP is preferable.

2.3. Consistency Algorithms

Consistency algorithms can be grouped according to the number of the nodes (vertices) they consider when searching for local inconsistencies.

2.3.1. Node consistency

Node consistency considers only a single vertex at a time; it simply checks unary constraints and deletes all values not allowed by them. As unary constraints can be previously eliminated from a CSP by restricting the domains, this algorithm is used only as a supplementary step in more complex consistency algorithms.

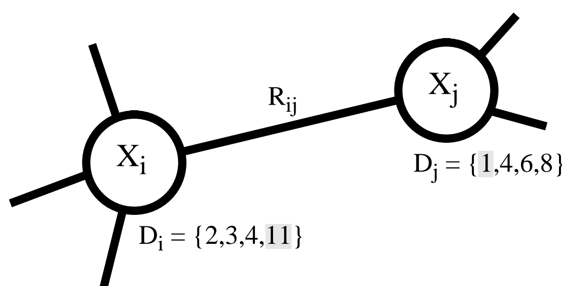
2.3.2. Arc consistency

Arc consistency considers two vertices X_i and X_j at a time, connected by a binary relation R_{ij}. It eliminates every value $\boldsymbol{x}$ from the domain of X_i that has no value pair $\boldsymbol{y}$ in X_j satisfying $R_{ij}(x,y)$. By checking appropriate vertex pairs and relations, full consistency can be achieved.

There are three basic versions of general purpose arc consistency algorithms (in the order of decreasing worst case time complexity) [3][5]:

- **AC-1** updates all the variables whenever any of the variable domains has changed. Its time complexity is $O(d^3nc)$ where $\boldsymbol{c}$ is the number of constraints;
- **AC-3** updates the domains of the variables adjacent to the changed variable. Its complexity is $O(d^2n)$);

- **`AC-4`** updates only those adjacent variables that are affected by the change of a variable domain. It requires some bookkeeping of the relations and the variable domains affected by them. However it is proven to be optimal; its complexity is $O(d^2c)$.



Let's consider that $R_{ij} := (X_i < X_j)$.

In this case:

- 11 can be eliminated from D_i as no value in D_j is greater than 11;
- similarly 1 can be eliminated from D_j as no value in D_i is less than 1.

Figure 2-1. Elimination of values with arc consistency

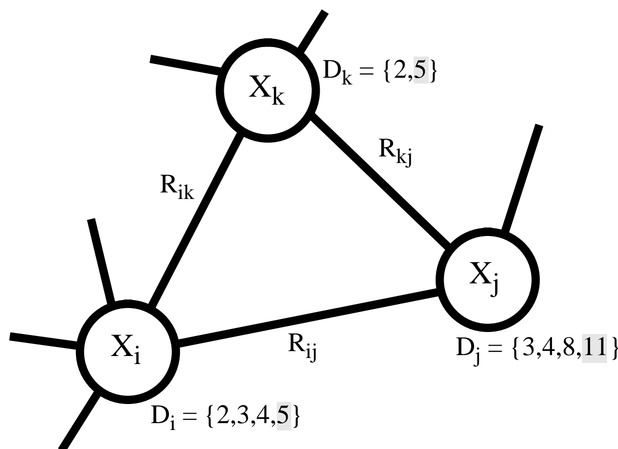
2.3.3. Path consistency

Path consistency between two vertices X_i and X_j connected by a binary relation R_{ij} means that all (x,y) value pairs in a solution of the CSP satisfying $R_{ij}(x,y)$ must be also allowed by all paths between X_i and X_j; i.e. if an X_i-X_k-X_l-...-X_q-X_j path exists then there must be values $\boldsymbol{k} \in X_k, \boldsymbol{l} \in X_l, \ldots, \boldsymbol{q} \in X_q$ satisfying $R_{ik}(i,k)$, $R_{kl}(k,l)$, ...,$R_{qj}(q,j)$. The whole constraint network is path consistent if every pair of directly connected vertices is path consistent.

Full path consistency in a complete constraint graph is equivalent to path consistency for length 2 paths [3]. Since any constraint network can be extended to a full constraint graph with dummy ("always true") constraints, checking path consistency is equal to checking only length 2 paths.

There are also three basic versions of path consistency algorithms; differences among them are similar to the differences among arc consistency algorithms [5]:

- **PC-1** updates domains of every vertex, vertices along every arc and every length 2 path if any vertex has changed. Its time complexity is $O(d^5n^5)$;
- **PC-2** updates domains of those length 2 paths that contain the changed vertex. Time complexity is $O(d^5n^3)$;
- **PC-3** updates only the length 2 path affected by the changes of a vertex domain. It uses similar bookkeeping about the influence of variables and edges like AC-4. It is also proved optimal and its complexity $O(d^3n^3)$.



$R_{ij} := \{X_j \geq 1.5 \times X_i\}$;
$R_{ik} := \{X_i \neq X_k\}$;
$R_{kj} := \{0.4 \times X_j \leq X_k \leq 0.5 \times X_j\}$.

If X_i=5, only $X_j \in \{8,11\}$ satisfies R_{ij}; from these, only X_j=11 and X_k=5 satisfies R_{kj} but it contradicts R_{ik} so the triple (X_i=5, X_j=11, X_k=5) can be eliminated from the domains.

Figure 2-2. Elimination of values with path consistency

2.3.4. k-consistency

A set S_k of k variables is considered at a time. If a consistent subset of value (k-1)-tuples exist on $S_{k-1} \subset S_k$ (with k-1 variables) then any value from the domain of the kth variable can be eliminated that cannot form a consistent value set with any of the consistent (k-1)-tuples. Global consistency can be achieved by successive elimination for increasing values of k until all variables are involved or the some of the domains become empty; in this case the CSP is not satisfiable.

The most well-known k-consistency algorithm is the *invasion* procedure [3]. Every step starts from a consistent subgraph G_i of the CSP graph G. The *front* F_i of G_i (those nodes in G-G_i that are adjacent to G_i) is checked in every step. All values are eliminated from the domains in F_i that cannot form a consistent value set with the allowed value combinations in G_i. F_i is added to G_i in the next step. This algorithm has a time complexity of $O(cd^{f+1})$ where $\boldsymbol{c}$ is the number of constraints and $\boldsymbol{f}$ is the maximal length of the front.

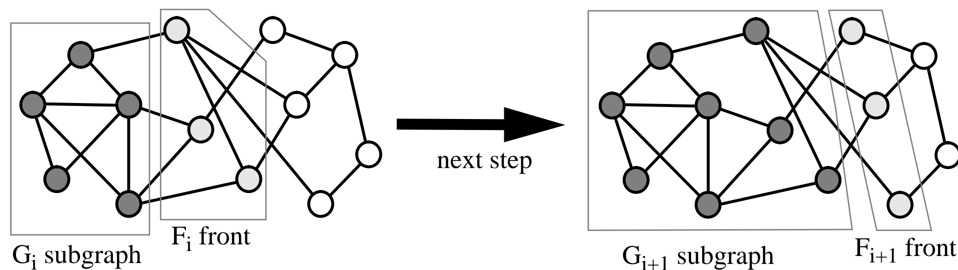


Figure 2-3. The progress of the invasion procedure

2.3.5. Local Propagation

Another very simple but efficient method can be used to solve binary CSPs: it is called *local propagation* of known values [8]. It originates from the data-flow representation of constraint satisfaction. It starts with assigning a value **i** to a variable X_i. Then the domain of every adjacent variable is checked and all the values that are inconsistent with **i** are eliminated. After that, values are assigned to the adjacent variables and the process can be repeated until every variable in the CSP is assigned or a contradiction appears.

This algorithm is the most effective if the CSP graph has a tree topology.

2.4. Similarity to the Self-diagnosis Problem

There are many similarities between self-diagnosis and constraint satisfaction. Actually the final goal is very similar: we want to know the fault state of the system components that conforms to our fault model and the actual test results (syndrome pieces). These restrictions can be represented by binary relations between the state of processors in a test pair. The exact relation is determined by the test result, thus a set of relations can be built from the syndrome information. CSP solution can be applied to find the possible fault states of the system.

The main advantage of the use of *relations* instead of *logical functions* is its elegant expressive power of the diagnostic uncertainty appearing (e.g. a faulty tester unit in the PMC model). The relations can be handled by a uniform mechanism, independently from the actual invalidation rules, system topology and the considered number of faults. So this representation is very flexible and is applicable on a wide range of systems.

Therefore a self-diagnosis problem can be very easily reformulated to a constraint satisfaction problem. The variables of the CSP represent the fault states of the system components. Constraints represent the restrictions from the test invalidation relations and the actual syndrome pieces. If one-pass diagnosis is required, a static binary CSP is produced. In the case of diagnosis on-the-fly each received syndrome element must be processed immediately by successively inserting the corresponding constraint into the set of relations. Every received test result reduces the solution space of possible fault states but the previously constructed relations (constraints) still remain valid, just new constraints have to be added. Therefore a monotone dynamic CSP can represent this case.

This reformulation gives a way to handle self-diagnosis problems very comfortably, with the well-elaborated toolset of CSP solution methods. With a sufficient diagnostic model, a very flexible method can be constructed.

2.5. Ideas from an “AI-like” Traditional Algorithm

The constraint-based approach is also very similar to the approach of the Selényi algorithm [2], whose syndrome decoding process consists of two phases. In the first phase all the deterministic information is extracted from the syndrome. This information contains all possible combinations of the fault states (CSP solution also produces this). Those units are identified in the second phase that remained unclassified in the first phase. This means excluding the unwanted solutions from the set of all possible solutions given in the first phase. It obviously requires further restrictions in the diagnostic model (assumptions on maximal number of faults, exclusion of certain faults, etc.).

The information extraction is based on logical implication rules between the fault state of the tester and the tested component. The concept of creating and using the implication rules originates in the “heuristic” solution of the self-diagnosis problem. This method assumes a fault state of a system component and starts to imply the consequences of this assumption. The conclusions imply fault state assumptions for other system components; the chain of implications can be continued further until all of the system components has and assigned fault state (so a possible solution was found) or some components have no consistent state (so a contradictory value assignment has been made during implication). In the latter case another assignment is necessary and the implication should continue with the new considered fault states.

The above mentioned process is implemented in [2] in a basis of complex matrix operations (e.g. computation of transitive closure). This apparatus requires very a sophisticated implementation. Computational efficiency becomes to a crucial factor in large-scale, massively parallel systems. A CSP-based implementation can be more effective with the same information extraction capability.

3 Implementation Environment

The experimental implementation of the CSP-based diagnosis algorithm took place on a Parsytec GCel supercomputer at the Institute of Mathematics and Computer Data Processing of the Friedrich-Alexander University, Erlangen- Nürnberg. This machine is produced by Parsytec Computer GmbH. The abbreviation GC refers to GigaCube (as it is a massively parallel multiprocessor with a cube-like mechanical design) as well as “Grand Challenges” (problems requiring exceptionally enormous computing power, as the Parsytec GC family is intended to be used for solving them).

For clearer use of terminology in this chapter, some terms and concepts about multiprocessors are described below.

3.1. Classification of Multiprocessor Systems

The term “multiprocessor” generally refers to systems where several computing elements (usually microprocessors, transputers or microcontrollers) cooperate for performing a common task in a system. The main motivation behind multiprocessors is the possibility for a simultaneous increase the computing power in and the dependability of the system. As the cost of microprocessors decreases (especially related to the cost of other computer components), it becomes affordable to apply more of them to multiply the capabilities of a system. Additionally, they have several other advantages as well:

- Multiprocessors can provide better performance/cost ratios than monoprocessor systems as the price of processing elements rises exponentially with the increase of their computation power. The processing capacity of a high-performance monoprocessor can be reached by applying multiple inexpensive processors. Moreover, the increase of the performance of a single processor has serious physical limitations: solid state structures can be realized only within certain size, operating frequency and temperature ranges

making higher performance processors so expensive. Multiprocessors can outstretch these limitations;

- Multiprocessors can provide a performance approximately equal to the sum of its components' performance by optimizing the use of shared resources in a system or separating specific sub-tasks to dedicated processors. Therefore more complex problems and applications can be realized, with a significantly lower cost than with traditional architectures. The overall throughput of a multiprocessor, however, does not always increases proportionally with the number of components. The increasing inter-processor communication overhead and resource allocation complexity reduce the increase of the performance, and after a peak point the system throughput starts to decline.

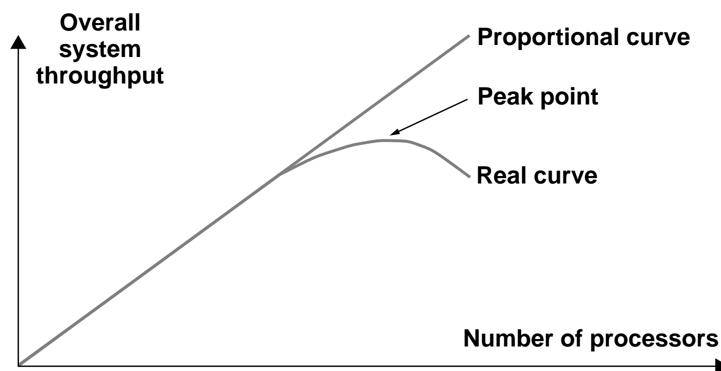


Figure 3-1. System throughput as a function of the number of processing elements

- Multiprocessor systems can be easily configured for different operating environments and tasks, thus they offer greater flexibility and higher overall utilization than monoprocessors;

- By applying several individual components in a system, the reliability can be increased. A multiprocessor contains a reasonable amount of redundancy; one of the main design objectives is fault tolerance. A higher overall MTBF can be achieved than the MTBF of any components by proper design and error maintenance mechanisms.

Multiprocessor systems introduce *parallelism*: it refers to performing independent tasks concurrently (different tasks are executed within the same time interval and they do not affect other tasks unintentionally). Parallelism can appear in many levels of system hierarchy:

- at hardware level: different hardware components perform independent but related tasks concurrently, using a coordination mechanism to prevent interference (co-processor systems);

- inside processor instructions: independent phases of machine instruction execution: opcode fetching, decoding, operand loading, opcode execution are performed simultaneously by different parts of a processor (pipeline architectures);

- among processor instructions: separate processing elements cooperate linked by the input and output data received and produced by them (data flow machines[1]);
- at data level: separate processing elements perform similar operations on separate parts of consecutive data simultaneously (vector-, array- and associative processors).

The communication between processing elements can be realized in three different ways. Elements of **shared memory systems** do not really communicate with each other as data transfer is executed by reading and writing a publicly accessible memory. **Common bus systems** have one publicly available communication facility (the system bus) that is able to broadcast data from one processor to all the others; finally **dedicated links** can be used to interconnect the processors: in this case each link is assigned to be used by exactly two processors. Common memory and common bus systems also need a facility to protect the communication medium from data collisions caused by simultaneous write operations (*memory/bus arbitration*). Dedicated links are organized into an interconnection topology. The same topologies are usual in multiprocessor systems as in computer networks: star, ring, fully connected, hypercube[2] and their combinations.

Multiprocessor systems can be also classified on the applied method of cooperation:

- **Loosely coupled systems** apply communication interfaces between processors, defining precise protocols for data transfer. Each processing element has its own local resources and performs data processing locally; however they can exchange data with other units via the communication network, so they can be arbitrarily long distances from other elements (computer networks are a special case of loosely coupled systems).
- **Tightly coupled systems** apply common shared memory for communication. Other resources are also usually shared but every processing element may have local private resources as well. These structures are commonly *symmetrical*: each processor has identical properties and rights in the system; the resource allocation and communication coordination is supervised by a public operation system, running on the whole set of processing elements. Synchronization is essential in tightly coupled systems due to the common shared memory. The number of processing elements is limited by the

---

1. Data flow machines are special non-Neumannian architectures where the operation of the system is driven by the processed data instead of a program execution structure; each task is initiated by the presence of the necessary input data and the output data of a task, together with other results, initiates other task(s). The name comes from the graphical representation of their operation. See also [23].

2. An n-dimensional hypercube is a generalization of the 3-dimensional cube: it can be obtained from the (n-1)-dimension hypercube by duplicating it and connect the corresponding nodes with new edges (the 0-dimensional hypercube is a single point). Hypercube topology is especially practical in multiprocessor image transformation applications.

increasing complexity of resource sharing, especially memory conflict prevention.

- **Distributed intelligence (moderately coupled) systems** represent a transition between loosely and strongly connected ones. In these typically asymmetrical systems a set of different processing elements, each optimized for a specified task, divide the necessary operations into partly independent tasks and each processor performs its specialty cooperatively. Communication is not centrally coordinated, it is the liability of the processors.

3.2. Hardware Overview

3.2.1. The Parsytec GC/GCel Machine

The Parsytec GC machine is a dependable massively parallel multiprocessor. Massively parallel systems are characterized by the enormous number of processing elements contained by them; Parsytec GC can contain up to 16384 processors. These architectures were introduced to completely fulfill the requirements for high-performance supercomputers:

- The processing capability of the Parsytec system is *scalable*: it can be easily increased by simply adding more processing elements to the system, and the performance increment is directly proportional to the number of processors. The actual performance range of the Parsytec GC is from 1 GFlop to 400 GFlops (containing 64 to 16384 processors) [25].
- Due to scalability, exclusion of some (faulty) processors from the system does not lead to system failure, only to a performance loss. Therefore the Parsytec GC, as other massively parallel multiprocessors, can *tolerate* a reasonable number of *faulty components* by substituting faulty processors with spares or by redistributing the tasks among the fault-free processors. The architecture of the machine includes dedicated spare processors so both of these solutions can be applied.
  This increased fault tolerance compensates the side effects of the large number of processors.
- Despite of the relatively low costs of the processing elements, the overall price of a massively parallel system is still very high due to the huge number of processors needed and the complicated additional hardware components (inter-processor communication facility, fault tolerance-related parts). Therefore the *efficient operation* of such an expensive system is very important. In order to maximize the computing power achievable on the system, a careful utilization of the available system resources is needed.

The hardware architecture of the Parsytec GC system is designed to meet these requirements. It employs MIMD parallelism in a loosely coupled form; each processing element is an INMOS T9000 transputer, equipped with 8 MByte local memory, executing tasks

separate from the other processing elements. User applications consist of several cooperating tasks, forming so-called task forces.

The inter-processor communication network is totally distributed, without a central management unit. Its topology - in physical and logical sense - forms a homogeneous three-dimensional mesh. This topology assures easy scalability as new processing elements can be added in any of the three spatial directions. The 3-D mesh provides a good structure for many practical applications, especially numerical analysis problems. However, the user can interact with a computer only through one centralized interface, and global resources (file systems, terminals etc.) should also be maintained centrally.

Therefore the Parsytec GC applies a *host computer* connected to its inter-processor network. This is liable for downloading and distributing user programs to the transputer nodes, and provides peripheral handling functions (mostly terminal I/O and file system) and user interface for the multiprocessor. The type of the host computer is indifferent; Parsytec GC uses Sun workstations for this purpose. It is possible, however, to attach external peripheral components directly to the Parsytec system as well.

The connection between the host and the multiprocessor is implemented physically by a simple serial interface. The host runs a special program called *server* that provides the interface between the host's resources and the Parsytec systems.

Additionally to the standard task of providing high-speed communication between the processing elements, the inter-processor communication system has other functions as well in the Parsytec architecture like interfacing to other systems, performing fault detection and maintenance. Therefore the communication network is divided into three separate subcomponents:

- the *data network* **(D-Net)** provides the data communication between the processing elements. The basic topology of the D-Net is a three-dimensional grid, although its physical implementation is different in the elementary units of the system. However, D-Net is capable to form an arbitrary *virtual topology* instead of the 3-D grid by reconfiguring the inter-processor data links, to fit to the current application optimally. D-Net consists of high-performance data links and programmable intelligent routing chips. Its fault-tolerant design assures that physical defects of data lines, connectors and even routing chips do not result in a total loss in the communication but only reduces the bandwidth of the network;

- the *control network* **(C-Net)** forms a separate network of processing elements (the control processors), and has dedicated connections to every application processing element. This component monitors the operation of the processing elements and maintain reconfiguration tasks for either optimal task allocation or fault masking. Therefore the C-Net distributes only configuration and diagnostic information; it cannot be used for communication between two application transputers, and it is unavailable for regular user applications;
- the *input/output network* **(I/O-Net)** is responsible for interfacing the global resources of the Parsytec system (mostly the host computer).

The Parsytec system employs a hierarchical modular structure for supporting scalability and decreasing system complexity. The basic module of a Parsytec system, a **GigaCube** is a complete multiprocessor system containing 64 processing elements, their interconnection network, interface connectors to other GigaCubes, a power supply and temperature control unit; it is able to perform stand-alone operation. Its mechanical design ensures interconnection possibility of GigaCubes in three spatial dimensions. The minimal configuration of the Parsytec GC contains one GigaCube; in the full configuration, 256 GigaCubes can be combined into a $4 \times 8 \times 8$ spatial array. The interconnection between GigaCubes provides eight data and one control lines in the 6 spatial directions.

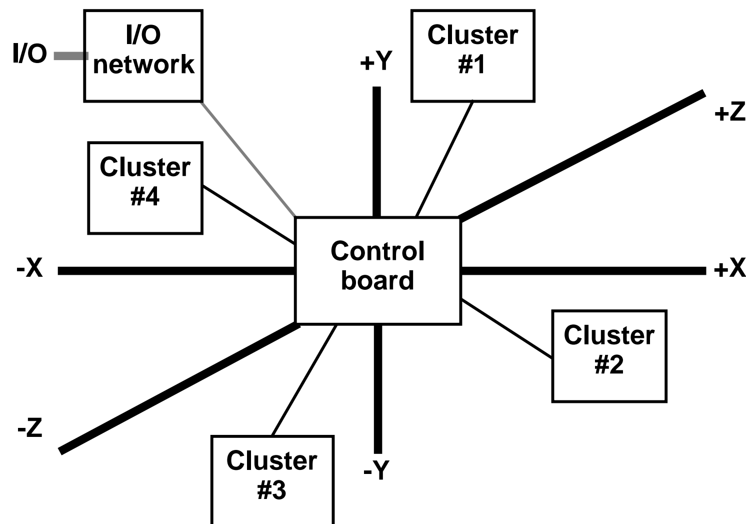


Figure 3-2. Structural diagram of a GigaCube

A GigaCube is formed by four clusters of processors. A cluster contains 16 + 1 processing nodes and 4 routing chips for interconnection. The four clusters are supervised by a single control processor in the C-Net. The task of the control processor is to download user application programs to the processing nodes, to control the I/O network, to monitor the op-

eration of the transputers as well as the power supply and temperature control system, and to perform maintenance actions when necessary.

A **cluster** is the smallest operational unit of the Parsytec system. It is possible to use partially equipped GigaCubes with only one or two clusters. It contains 16 transputers running user applications and the operating system kernel, four INMOS C104 routing chips interconnecting the transputers and a reserve transputer for fail-safe operation. As the reserve transputer can replace any defective working transputer, the overall failure rate of a cluster is lower than the failure rate of a single transputer. The spare replacement process is driven by the control processor in the C-Net.

The internal interconnection scheme of a cluster is a fully connected graph, implemented by C104 routing chips (see **Figure 3-3**.). Each cluster has an external connection facility of 8 data lines in the six spatial directions and 8 connection lines to the I/O network.

The electronic circuit design of a cluster constitutes two processor cards (with 8 transputers and two routing chips each) plugged into the GigaCube backplane. The 17th reserve transputer is placed separately on the common control board of the GigaCube.

Processing nodes of a cluster are also designed for maximal fault tolerance. To minimize the number of components and thus potential faults caused by damaged inter-component data paths and component manufactory defects, most of the supplementary functions of the elements are integrated into an application-specific integrated circuit (ASIC). So a processing node consists of only three active components: a T9000 transputer, DRAM memory chips and the ASIC. Each transputer can have 4, 8, 16 or 32 MBytes of local memory (32- or 64-bit wide). DRAMs are protected from **soft errors** (nondestructive transient faults resulting from e. g. alpha particles) by an error detection correction (EDC) logic that is incorporated to the ASIC.

For optimizing inter-processor data communication and the cache facility of the transputers, a special data link and memory usage monitor is implemented. The data collected by it is available for user applications. Each processing element has various test functions to allow software-based hardware fault detection.

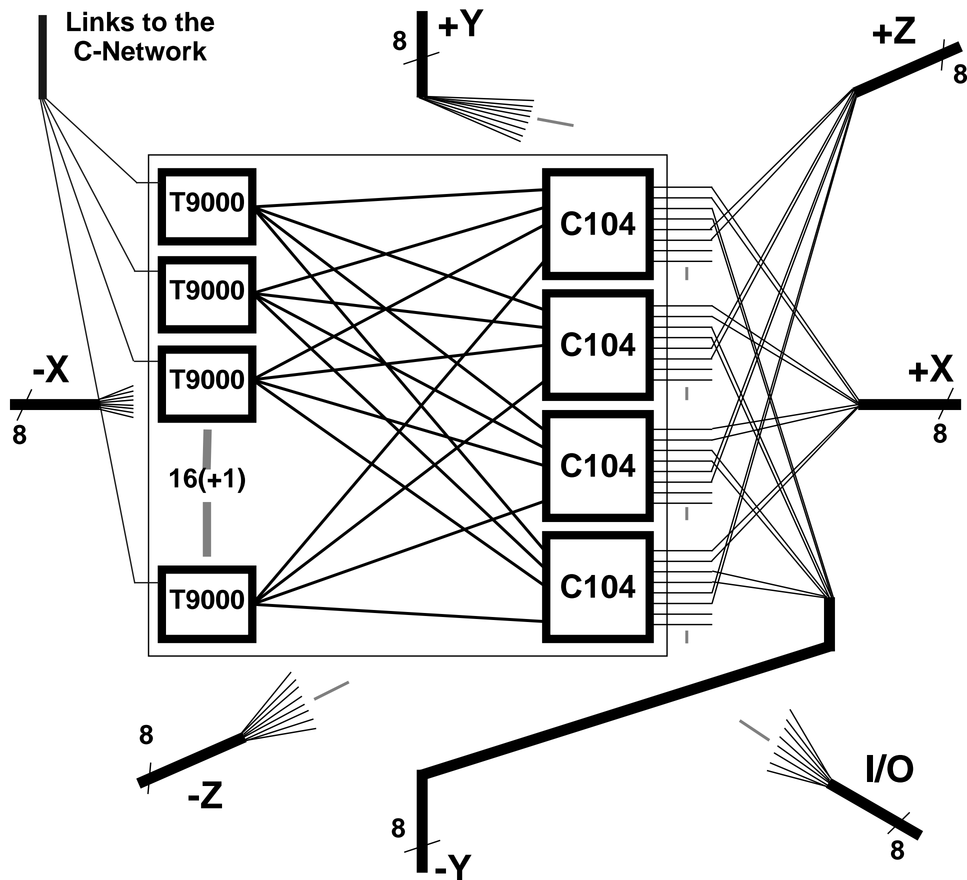


Figure 3-3. Structure of a Parsytec cluster

3.2.2. The INMOS T9000 Transputer

The T9000 transputer is the latest member of the transputer series developed by INMOS, Ltd.,especially for application in massively parallel multiprocessor systems. High-performance communication facilities of the INMOS transputers provide the possibility to design multiprocessor systems with extremely high overall performance.

The block scheme of the T9000 is shown in **Figure 3-4**. The transputer chip contains a 32-bit CPU, a 64-bit floating point unit, 16 KBytes of on-chip memory cache, a hardware scheduler, a communication co-processor and four high-speed serial communication links.

The *32-bit CPU unit* has a superscalar pipeline architecture. The five-stage pipeline and the carefully selected instruction set provides a high execution speed of programs written

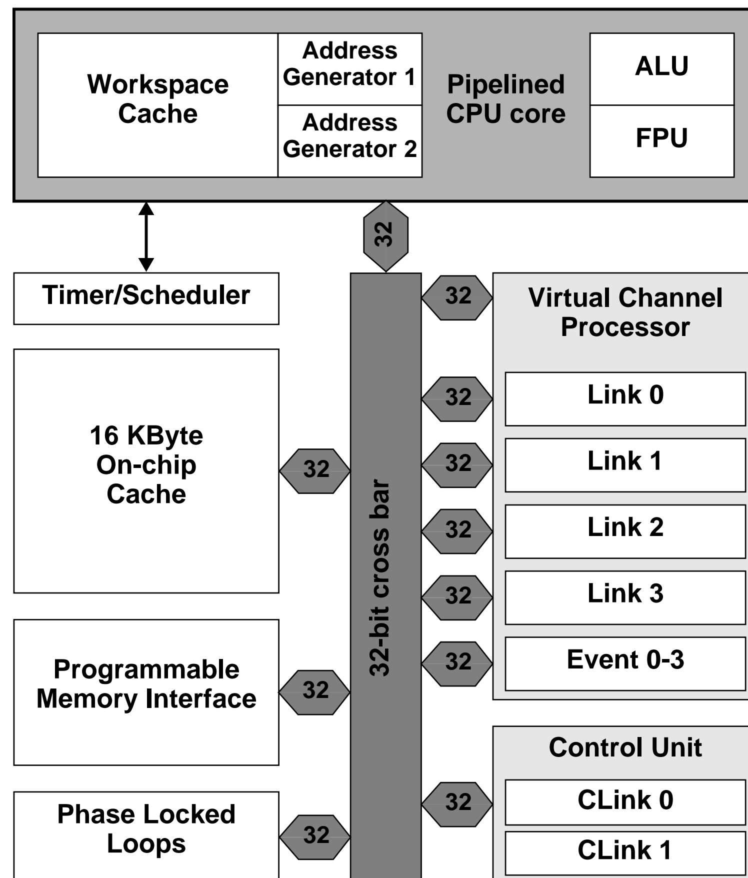


Figure 3-4. Block diagram of the T9000

in high level languages. Moreover, a hardware optimizer is applied to preprocess instruction groups from the running code in order to exploit the maximal pipeline performance.

Programmable *address generators* provide memory access control and privilege level handling by memory address translation in the protected mode of the CPU. An integrated *hardware scheduler kernel* supports multi-thread operation. Typical interrupt response and context switching times are less than 1 µs. Context switching operations are included in the instruction set. These features result a peak performance of 200 MIPS of the CPU and 25 MFlops of the FPU [24].

16 KBytes of unified *data and instruction cache* is implemented on-chip for decreasing the number of external memory operations and achieving single-cycle instruction executions. Moreover, a small additional cache is present for the most frequently used data, e.g.

local variables (*workspace cache*). The peak bandwidth of the on-chip caches is 200 MWords/s.

A wide range of external memory devices can be attached to the T9000 transputer without complicated external logic, via the *programmable memory interface*. It can be adapted for the fastest DRAMs with special (paged, refresh combined etc.) addressing strategies as well as SRAMs and even slow ROM and memory-mapped peripheral devices. Different sets of memory control options can be switched very quickly for supporting heterogeneous memory structures. The word width of the memory can be 8, 16, 32 or 64 bit.

The sophisticated *communication co-processor* offers an elegant solution for incorporating data exchange facilities into application programs: communication between **processes** can be handled with the same mechanism as between **processors** so it is transparent for the user: an application needs not to know if their cooperating processes run scheduled on the same transputer or simultaneously on several transputers. The *virtual channel processor* provides the capability of employing **virtual data links** between processors, thus using an arbitrary number of logical data channels forming various **virtual topologies** over the physical 3-dimensional grid topology. Therefore a multiprocessor built from T9000 transputers can be easily adapted to various types of problems, using the optimal virtual topology for each of them.

The communication subsystem also contains four additional data lines. These inputs (called event lines) can be used similarly to hardware interrupt lines as appearing of data on them results a processor exception. Practically event lines applied as inputs of fault alert signals produced by a testing circuit (e.g. the EDC logic signals an alert if an uncorrectable memory error is detected).

The transputer's *physical connections* provide 100 MBit/s full duplex serial data throughput. As the whole communication co-processor operates concurrently with the CPU core and direct memory access (DMA) is used to transfer data between the data link circuitry and local memory, the amount of communication has no significant influence on the speed of program execution.

The *control unit* of the T9000 transputer is liable for supervision and monitoring of the other units and it performs the general management-related functions like initialization and configuration. The control unit can communicate with other units separately from the stan-

dard data links, by using its dedicated control links. These links are generally reserved for system configuration and maintenance messages, but they can be used optionally for data transfer as well (e.g. application programs can be downloaded this way).

Communication via the control links can be initiated either by the transputer itself or by an external unit. Therefore a higher level of system supervision can be applied over a multiprocessor system consisting of T9000 transputers, as a detailed status information of the transputer is available via the control links and the control unit can receive and execute commands from another unit as well. In the Parsytec GC system, control links of the application transputers are connected to the C-Net.

The *phase locked loop* (PLL) circuitry in the T9000 transputer is a part of the data link and memory usage monitoring subsystem mentioned in the previous subsection.

3.3. Software Environment

Effective use of a multiprocessor system requires a sophisticated working environment providing tools for running user applications on the processors, partitioning the multiprocessor among different users and applications, managing the allocation of system resources and developing programs. The operating environment designed for the Parsytec GC machines is called PARIX (from PARallel unIX). It is a distributed UNIX-like operating system, extended for use with parallel multiprocessor systems. It runs partly on the T9000 application processors of the Parsytec GC, partly on the C-Net control processors and partly on the host machine.

The PARIX system can be divided into four main components:

- **development environment**: standard program development tools (compilers, debuggers, profiler) together with the server interface for downloading programs to the transputers and display its output. These tools run on the host machine;
- **runtime environment**: a small UNIX-like kernel to provide inter-processor message passing facilities, together with higher level communication and other support libraries for building various applications. The kernel is linked to the application running on the Parsytec transputers. Runtime environment also includes drivers and programmer's interface for various peripheral units connectable to the Parsytec machine directly, including the host machine itself;

- **multi-user administration**: a common interface to ensure user interaction with the active applications, management of system resources allocated by the users' applications and controlling the application execution.

  The most important resources in the Parsytec system are the transputers themselves. Separation of jobs is performed by creating different groups of processing elements (called *user partitions*) that belong to a single user and can be used only by his/her applications. Sharing of other resources is generally managed by the host machine as most of the peripherals are usually attached to it instead of the Parsytec GC. The administration tools also run on the host machine;

- **control network software**: the program running on the control processors in the C-Net, responsible for initializing and starting up the Parsytec system correctly, performing regular supervisory and maintenance tasks, ensuring the fault-tolerant operation and reconfiguring the transputers according to the actual partitioning required by the administration software.

The collection of these tools provides maximal exploitation of the potential capabilities of the Parsytec system while offering a comfortable user access.

3.3.1. Programming Model

Hence the system architecture of parallel multiprocessor systems radically differs from the traditional monoprocessor machines, the *programming model* is also obviously different. Multiprocessor systems, however, have also completely different architectural properties so many different programming models were developed for various architectures.

The model applied by the Parsytec GC machine is quite common in massively parallel systems. Its primary intention is to cope with the extremely high number of processing elements without making application development uncomfortably complicated. The main characteristics of Parsytec's programming model are the following:

- The *same binary program code* is downloaded to each transputer when an application is started. This method simplifies program development - there is no need to write several versions of the program for different processors. The PARIX kernel initializes certain pre-defined global system variables identifying the processor at load time; a processor can determine its spatial position by examining this data and is able to perform different tasks according to its place in the topology.

- Besides of self-identification, each processor can obtain information on the *size of the actual user partition*, thus providing the possibility of automatic *software scalability* so an application can adjust its internal configuration to the actual number of processors involved.

- Hence the T9000 transputer has a hardware implemented timer/scheduler facility, application of multitasking is also possible within a single processor node. For maximal performance, the Parsytec system supports *multi-threaded* programming. (Threads - also referred to as lightweight processes - are special processes sharing a common partial context, therefore switching between them in a multitasking environment requires less time and data transfer than switching between “heavyweight” processes when the full process context must be saved and restored.)
- The hardware scheduler supports *priority handling*. Actually two priority levels exist. Low priority threads are scheduled normally so they are periodically interrupted; high priority threads run uninterrupted until they terminate. (This feature is a source of potential danger as infinite loops within a high priority thread cannot be terminated from within the transputer (even hardware interrupts are disabled); breaking such a loop is possible only via the control links of the transputer.)
- *Additional user programs* can be loaded and executed from within an application, with full control on program placement (i.e. the group of the transputers where the new application will run can be also selected). Every new application constitutes a full process context; therefore direct interaction between different contexts is not possible. Threads of the additional application are scheduled together with threads of the original program; the thread initiating the load and execute activity, however, will be suspended until the loaded new application terminates.

Programs developed under this model can exploit the advantages of the massively parallel multiprocessor architecture of the Parsytec GC machine. With additional support of the PARIX operating system kernel and the provided development libraries, it is possible to create powerful applications with relatively small effort as the model allows programmers to concentrate on implementing the effective application algorithms instead of taking care of the architectural details of the system.

3.3.2. The PARIX Operating System Kernel

In a regular UNIX system, the kernel of the operating system is a separate program running continuously on the machine, handling the hardware components of the system and offering various system services to application programs. Due to the different structure of the Parsytec GC system, many of the tasks of a traditional operating system kernel are handled by the host computer. Therefore the kernel of the PARIX is oriented to provide standard UNIX services (system calls) and other extended services related to the Parsytec architecture. It works together closely with the standard development libraries.

The kernel itself is a binary object module that is linked to each application program. Its static code size and dynamic memory allocation was minimized to allow the highest pos-

sible application performance. (Probably this is the cause of that a few well-known and widely used UNIX features were left out from the current version of the kernel.)

The most important set of services provided by the PARIX kernel and the libraries is the support for inter-processor communication. The following features are available for the applications:

- As mentioned in Section 3.2.2, inter-processor and inter-process communication is handled by an *unified mechanism* so the implementation of the inter-process data exchange in a user application can be independent from the physical location of the communicating processes.
- *Virtual data links* (bidirectional unbuffered logical data connections) can be set up between arbitrary processors and processes, totally independently from the actual physical interconnection system. Creation and termination of virtual links is possible at run time as well.
- Sets of virtual links often used for creating a specific interconnection scheme can be organized to a *virtual topology.* Virtual topologies can be established in one step to save the time-consuming effort to build all the necessary virtual links every time when they are needed. A set of pre-defined virtual topologies are supplied with the P ARIX communication libraries, implementing the most common topologies (grid, torus, tree, pipe and hypercube) scalable to the actual application, with optimized virtual data link mapping to physical data links. However, every user can easily build a private virtual topology library. An application can use *more than one* virtual topology at the same time for adjusting the interconnection scheme optimally for different parts of its algorithm.
- Three *basic communication modes* are available:
  - **synchronous link-bound communication**; it is the fastest method, directly supported by the transputer hardware and the routing chips. A virtual link is needed for the communication (it can be a link of a virtual topology but can be created ad hoc as well) or random routing can be applied (in this case a set of temporary virtual links is automatically created between the two communicating processors). The communication *blocks* any of the involved processes if the other process is not ready: they must execute the communication requests cooperatively and their operation resumes only after the data exchange has been finished.
  - **asynchronous link-bound communication**. In this case the sender process can initiate the communication request and continue its operation immediately. Actual data exchange takes place only when the receiver process is ready. Buffering of the data during the wait period can be performed either by the user program or by the kernel, on both sides of the communication. Asynchronous communication requires a virtual link between the communicating processes.

    Asynchronity is implemented in the libraries by using the synchronous communi-

cation facility and starting a new thread dedicated to the communication (so the dedicated thread will be blocked instead of the user process).

A *time-out* value can be assigned to asynchronous *receive requests* as well for automatic termination of the dedicated communication thread if no data has been sent within a given time period. So accumulation of inactive communication threads in the system (consuming valuable resources) can be avoided. Unfortunately, there is no way to terminate an asynchronous send thread after it was initiated.

- **mailbox-based communication**, when messages are passed from the sender towards the receiver using a software router. There is no need for any direct connection between the communicating processes at the time of communication; the software router, however, needs to be initialized on both processors at boot time. This communication mode does not require any form of handshaking: the sender process continues immediately after sending a message, the receiver process can regularly check its own mailbox for incoming messages.

Besides of the communication, the PARIX kernel and libraries provide standard UNIX peripheral handling functions as well. As a Parsytec machine seldom has peripheral units attached directly to itself, handling of the peripheral devices of the host machine is implemented by a Remote Procedure Call (RPC) mechanism. An RPC is initiated by opening a special data link from a process to the server interface running on the host and sending an appropriate message; the server processes the message, activates the corresponding system service on the host machine and sends the result back to the process.

Most of the standard UNIX terminal and file I/O library functions are available via this method; however, due to the need of synchronous communication between the process and the host, response time of an RPC is limited by the transfer speed of the serial connection between the Parsytec and the host machine. Therefore extensive use of RPC-based library functions decrease the efficiency of the application seriously.

The PARIX kernel includes some extra debugging facilities as well to support the operation of the parallel debugger. These functions are only partially implemented in the kernel version 1.1 which was available at the time of the development.

3.3.3. Development Tools

The development process of user programs for the Parsytec machines is based on the *cross-platform development* concept: the effective development tools reside and run on a separate machine and only the binary code is transferred to the Parsytec system. This concept has the obvious advantage that programmers can use their own well-known and cus-

tomized working environment (text editors, source code management tools etc.) hence only the binary code generation parts of the development system needs to be changed. As use of Parsytec GC systems requires the application of another computer system by default, cross-platform development is the only logical way to develop applications.

Compilers for various high level languages (C, Fortran, Pascal, Modula-2) are available or are under development for Parsytec systems. These compilers are developed by ACE B.V., a Dutch software company. At the site of the development in Erlangen only the ACE EXPERT C (an ANSI C compliant C compiler) and the ACE EXPERT Fortran-77 were available. Use of the assembly language of the T9000 transputer is also possible through the common T9000 assembler of the compilers.

Both assembly level and source level debugging is supported by the PARIX operating system. Debugger programs has two parts: a small stub is loaded into the memory of the T9000 transputers (it performs the effective debugging tasks using the '**ptrace**' functionality of the PARIX kernel) and the front-end user interface running on the host machine (it communicates with the debugger stub via RPC calls). Due to development delays, source level debuggers were not available at the time of the development, only a simple one offering the functions of the 'adb' UNIX debugger, with some parallel extensions.

A performance analysis and profiling tool called PATOP, developed at München University of Technology for optimizing application efficiency in highly parallel systems, is also available for Parsytec GC machines.

3.4. Differences between T9000 and T805 Transputers

The constraint-based diagnosis algorithm was developed on a partially equipped Parsytec GCel machine with only one clusters and 16 processing elements. As the machine was installed in 1992, its processing elements were INMOS T805 transputers. (The T9000 series is the successor of the older INMOS T800 transputer family; due to development problems with the new processor design, T9000-based Parsytec systems were not available yet when the development has started.)

The T805 transputer is strongly similar in its internal structure to the T9000 what was described in the previous sections. However, it lacks some new features that were designed specifically for the T9000 series. The most significant difference is that the T805 has no

virtual channel management and it cannot cooperate with C104 routing chips. The functions of the routing chips are emulated by software components incorporated into the PARIX kernel.

Moreover, T805 transputers cannot run in protected mode and their memory management unit has no any protection facilities; therefore an extreme care must be taken of using pointers as writes to the memory via an invalid pointer can effectively destroy the PARIX kernel and thus cause total failure of the system.

For maximal portability, the architecture of Parsytec GCel system and the Parsytec development tools were designed in such a way that every piece of software written for earlier Parsytec systems (with T805 transputers) can be simply recompiled, without any source code modification, for use on Parsytec machines equipped with T9000 transputers. Therefore application developers do not need to wait until T9000 appears and their work will run on later Parsytec systems as well.

4 The Developed CSP-based Diagnosis Algorithm

4.1. Fault model

The primary goal of this diploma work was to validate the concept of constraint-based testing in the practice. Therefore no effort was made to produce a full-featured, commercial quality product, only a small experimental system.

During development of the fault model, a Parsytec GC system - equipped with T9000 transputers and C104 routing chips - was considered for demonstration of the diagnosability of a non-homogenous system, in spite of that the effective algorithm ran on a GCel machine with T805 transputers (as mentioned in Section 3.4). The difference in the hardware structure did not cause significant changes in the algorithm, it affected only the implementation.

The test mechanisms used on the Parsytec processors are simply periodical *mutual <I'm alive> messages* between neighboring processors. A test passed if the next <I'm alive> message was received within a given time interval and it was correct; failed if the message was not sent by a neighbor within the time-out limit, or the internal format of the message was incorrect (the correctness of <I'm alive> messages was checked with a checksum field). Therefore a test could produce three possible results: **good** (test passed), **faulty** (test failed due to invalid <I'm alive> message) and **dead** (test failed due to time-out).

More accurate test methods were not available at the time of development. However, as testing is performed via the normal inter-processor data connections, efficiency of a more accurate test may be undesirably low (see Section 1.3.3 and [20],[21]). Alternatively,

the application transputers can be tested from the C-Net of the Parsytec system (see Section 3.2.2); this kind of test mechanisms are still under development at University of Coimbra.

Besides of the obvious *processor* faults, defects appearing on *data lines* between processors and routing chips and faults of the *routing chips* themselves are also included into the fault model. This makes the model more realistic and the diagnosis more effective.

Determination of the fault states of a processor can be based on considerations of its internal structure. As the faults on the transputer level are assumed to be consequences of lower-level faults, investigation of the possible faults on a lower level helps creation of high level fault states.

Fortunately, the fault-tolerant structure of a Parsytec computing element provides the possibility to detect a considerable amount of faults for an external device (e.g. the control processors in the C-Net) [4]. The detectable physical fault classes of the T9000 transputer are the following:

<table>
<tr><th>Fault class</th><th>Physical cause(s)</th><th colspan="2">Effect on the transputer</th></tr>
<tr><td>PMI Error</td><td>internal defect in the programmable memory interface unit</td><td colspan="2">lockup</td></tr>
<tr><td>VCP Error</td><td>internal defect in the virtual channel processor unit</td><td colspan="2">lockup</td></tr>
<tr><td rowspan="2">CPU Configuration Error - Hardware Exception</td><td>internal defect of CPU configuration registers, illegal memory access, illegal instruction etc.</td><td rowspan="2">the NullTrap routine (a special part of the PARIX kernel) is activated</td><td>lockup (intentional, caused by NullTrap routine)</td></tr>
<tr><td>divide by 0, floating point exception, other software-related faults</td><td>NullTrap routine handles the fault, operation continues</td></tr>
<tr><td>Link0...Link3 Communication Error</td><td>mechanical contact problems, line noise, router faults</td><td colspan="2">no communication via the faulty data link</td></tr>
<tr><td>CLink 1 Error</td><td rowspan="3">mechanical contact problems, C-Net processor faults</td><td rowspan="3" colspan="2">lockup</td></tr>
<tr><td>CLink 0 Protocol Error</td></tr>
<tr><td>CLink 0 Command Error</td></tr>
<tr><td>Event0...Event3 Signal</td><td>various (e.g. unrecoverable memory error alert from EDC ASIC)</td><td colspan="2">depends on the origin of the signal (in the case of EDC memory fault, random behavior)</td></tr>
</table>

Table 4-1. Fault classes of the T9000

These faults are detectable if the appropriate functional unit performing internal fault detection and report (i.e. the control unit of the T9000) is still operational. In the case of a fault in the control unit, no information can be received about the fault state; however, hence the control unit is liable to coordinate the whole operation of the transputer, it can be assumed that the fault of the control unit implies the **total disfunction** of the T9000.

Moreover, as neither the CPU working registers nor the on-chip cache memory is protected against soft errors, external physical phenomena, like alpha particles from cosmic radiation can cause stochastic temporary faults in these units; these faults are manifested in **random behavior** of the transputer (i.e. unpredictable changes in the data or in the control flow).

Not each of the above mentioned lower level faults manifests as an unique fault state at transputer level. Some of the lower level faults are *equivalent* (they give identical results for the tests aimed to detect them) and some of the faults may *dominate* other faults (tests for the dominated fault always fail in the presence of the dominating fault). These relations must also taken into account when creating the fault model [32]; the applied testing method affects the selection of significant transputer-level faults.

Therefore the T9000 transputer can be modeled with three significant fault states: *fault-free* (it operates correctly), *faulty* (it operates incorrectly, gives improper results but its communication facilities are good) and *dead* (it does not communicate with other transputers). These fault states correspond to the three possible results of testing with <I'm alive> messages; this simple test mechanism does not make more sophisticated fault pathology possible.

Fault states of data links are obvious. Routing chips consist of data port circuits connecting to the data link pins of the chip and an internal routing logic. It is assumed that only a single data port is faulty in a given time; faults in the routing logic, however, cause complete failure of the routing chip so no further communication is possible through it.

The complete fault model, including the notations used in Section 4.3, is shown in **Table 4-2**.

Due to the simple, almost pass/fail style tests, PMC (symmetrical) test invalidation was used (a fault-free transputer tests its neighbors well, a faulty transputer delivers random test

Unit	Fault state and its notation		Behavior	Possibly faulty compo-nents(s)
Processor	fault-free	0p	correct operation	-
	faulty	1_p	incorrect test result evaluation	memory
	dead	c_p	no communication	CPU configuration, virtual link, C-Net-work, hardware excep-tions
Data link	live	$L_{p,R}$	correct message trans-fer	-
	broken	$\bar{L}_{p,R}$	no message transfer	wires/connectors, CPU data link circuit
Routing chip	fault-free	-	correct operation	-
	single port fault	$L_{R,p}$	no message transfer via the faulty port	router data port circuit
	dead	m_R	all ports are faulty	internal routing scheme, clock

Table 4-2. The applied fault model

results, a dead tester neither executes tests not it gives any result). More sophisticated tests are needed for using other invalidation schemes; however, changing the test invalidation rules requires only minimal effort and does not affect the operation of the diagnostic algo-rithm significantly.

4.2. Assumptions

The developed diagnosis algorithm itself runs on the host machine and only the lower-level tester functions run on the Parsytec transputers; in other words, the algorithm is based on the concept of *centralized diagnosis*. This approach was selected primarily due to the difficulties with implementing the constraint solver on the T9000 transputers. Centralized diagnosis, however, can be considered usable on a Parsytec machine, due to the following reasons:

- Despite of its symmetrical distributed nature, the host machine is still a centralized ele-ment of the Parsytec GC machine so the failure of the host definitely implies the failure of the whole system; therefore centralizing diagnosis into the host does not decrease the overall dependability of the system;

- Although the tests between processing elements are performed via normal data links, each transputer has a separate data connection: the control links attached between the application transputers and the control processors in the C-Net, so test initialization commands and test results can be transferred separately from the standard inter-processor connections. As the control links are parts of the C-Net, their reliability can be considered higher than data links as they are reliable (among others) for the fault tolerant operation of the Parsytec machine; in this algorithm, they are considered fault-free.

The *asynchronous* communication mode is applied for exchanging <I'm alive> messages. Obviously, the mailbox-based mode is unusable for diagnostic purposes as it is based on software routing (thus implicitly on the correct operation of all transputers). The synchronous communication mode would be ideal due to its high speed. The communication libraries available at the time of development contained time-out option only for receive operations, therefore an occurrence of a dead transputer - that did not attempt to commence the synchronous communication - would block all the sender threads on the neighboring processors, thus bringing the whole system into a halt. The use of asynchronous communication solves these problems, but arises another one: termination of failed asynchronous send requests (or, to be precise, the dedicated threads that implement asynchronity) is not possible in version 1.1 of the PARIX kernel.

Finally, the algorithm developed assumes a single routing chip between any two transputers, as it is within a cluster. This *intra-cluster* approach, however, can be easily extended hierarchically to achieve a system-wide diagnosis; as routers of a cluster are tested good, they can be eliminated from the diagnostic model, i.e. replaced with a set of direct data links and their fault state needs not to be considered again.

4.3. Transformation into a CSP

The developed algorithm is based on the concept of representing the diagnosis problem as a constraint satisfaction problem. This transformation involves creating implication rules on the basis of the system structure, the applied test invalidation model and the actual syndrome bits. These implication rules can then be represented by constraints.

Some of the implication rules can be generated before starting tests as they are independent from test results. Some others must be generated at run time, from the incoming syndrome bits. Therefore the resulting constraint satisfaction problem will be a special dynamic CSP (it special property is that the restrictions in it can only grow stronger; a con-

straint never loosens after it is constructed once). This feature makes application of *static* CSP solution methods (with only marginal modifications) possible.

All constraints are *binary* to achieve maximal simplicity in their treatment: they represent implications between fault states of components only. Test results are not included in the constraints as variables, as these are already known when the syndrome decoding begins. Syndrome bits are used for creating run-time constraints instead.

The implication rules resulting from the system structure (fault domination rules) and the PMC test invalidation are shown in the figure below:

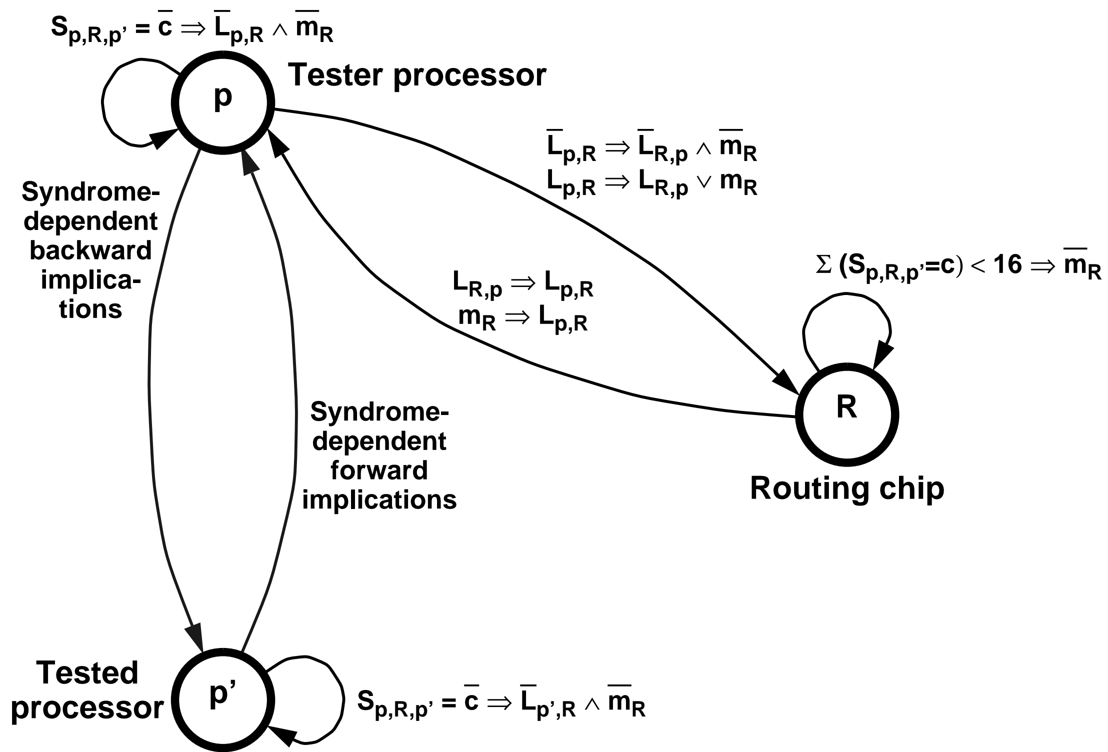


Figure 4-1. The syndrome-independent implication rules

The above implication rules are interpreted quite straightforward (e.g. the implication $S_{p,R,p'} = \bar{c} \Rightarrow \bar{L}_{p,R} \wedge \bar{m}_R$ denotes that if a processing element tests an other processing element as non-dead, then the data link from the tester processor to the involved routing chip and the routing chip itself are good). The constraints representing these rules are created at start-up of the diagnosis algorithm.

Some of the above implications express the fault dominance between faults of different units; e.g. $L_{R,p} \Rightarrow L_{p,R}$ represents that the data port fault of a routing chip implies the fault of the corresponding data link.

The implication rules depending on the test results are the following:

- **Forward implication** (from the fault state of the tester to the fault state of the tested processor)
  - $S_{p,R,p'} = 0 \wedge 0_p \Rightarrow 0_{p'}$ (i.e. if the tester processor is fault-free and the result of the test is good, then the tested processor is good);
  - $S_{p,R,p'} = 1 \wedge 0_p \Rightarrow 1_{p'}$;
  - $S_{p,R,p'} = c \wedge 0_p \Rightarrow L_{p,R} \vee m_R \vee L_{p',R} \vee c_{p'}$ *.
- **Backward implication** (from the fault state of the processor under test to the fault state of the tester processor)
  - $S_{p,R,p'} = 0 \wedge 1_{p'} \Rightarrow 1_p$;
  - $S_{p,R,p'} = 1 \wedge 0_{p'} \Rightarrow 1_p$;
  - $S_{p,R,p'} = c \wedge \overline{c}_{p'} \Rightarrow L_{p,R} \vee m_R \vee L_{p',R} \vee \overline{0}_p$ *.

These implication rules are created upon the receive of syndrome bits.

It must be pointed out that some of the constraints (denoted with an asterisk) are not pure binary constraints. Fortunately, these constraints are all related to m_R, the dead state of a routing chip; as m_R can be quickly eliminated from the possible fault states (due to the constraint $\Sigma(S_{p,R,p'} = c) < 16 \Rightarrow \overline{m}_R$, the first non-dead test via a router implies $\overline{m}_R$), it does not cause serious problems during constraint solving.

The CSP corresponding to the above implication rule set contains 20 variables in the implemented algorithm: 16 variables describe the fault state of the 16 processors and their data links and the other four ones represent the routing chips. The ‘processor’ variables have an initial domain of 48 fault states: the transputer itself can be either fault-free, faulty or dead, and any of its four data links can be live or broken, it results $3 \times 2^4 = 48$. The ‘routing chip’ variables have 18 possible fault states: a router can be good, one of its 16 data ports can be faulty or the router can be dead.

The operation of constraint generation is shown in an example. In Figure 4-2., four processors and three routers are involved in test relations. Domains of the variables are simpli-

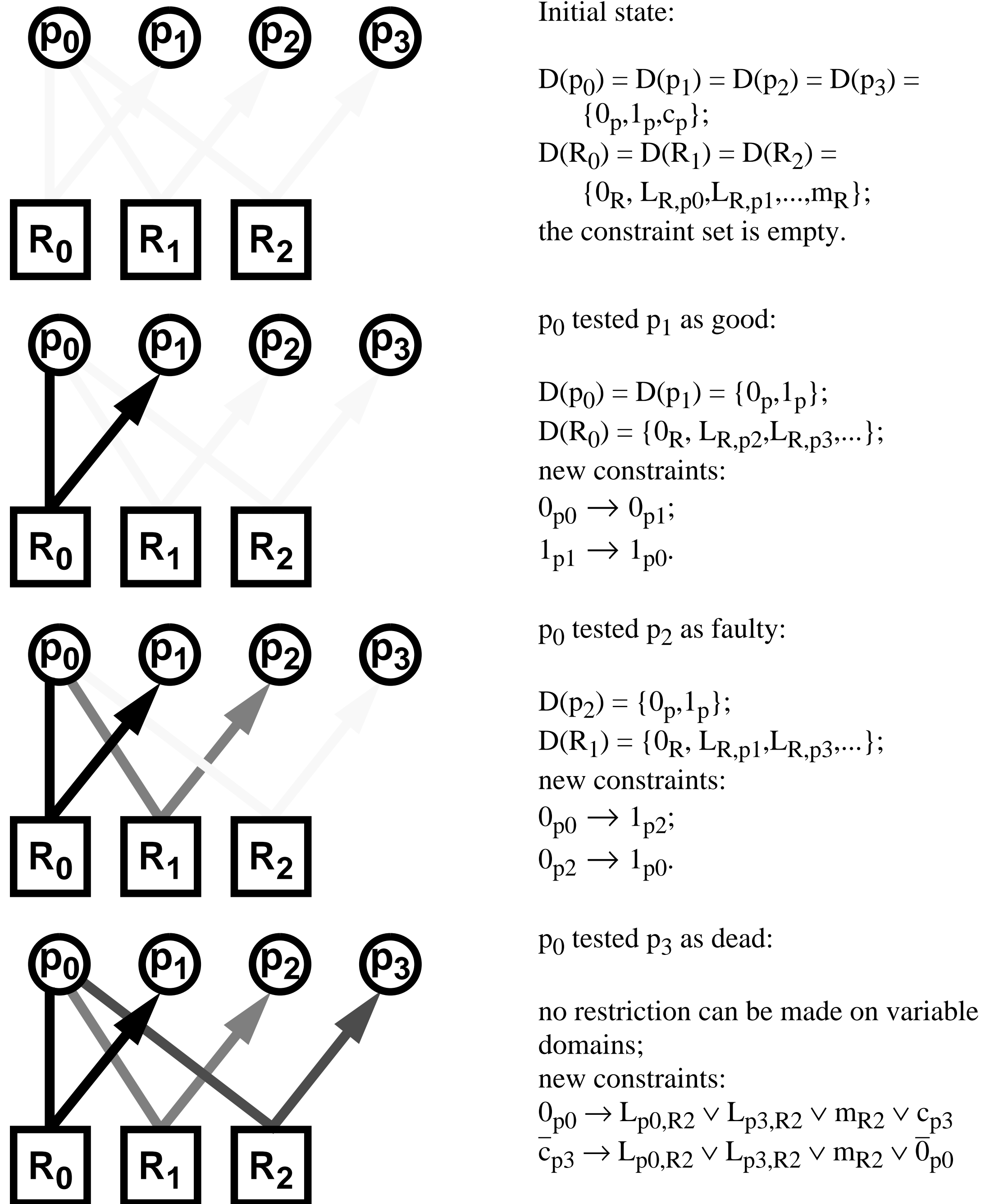


Figure 4-2. Operation of the diagnosis algorithm

fied (fault states of the data links are not shown). The figure shows the processing of the first three test results, demonstrating the application of the pre-generated constraints and creation of the run-time constraints based on the test results.

4.4. Implementation details

4.4.1. Low-level Testing Mechanism

The low-level testing parts of the diagnosis algorithm runs on the Parsytec T9000 transputers. It is written in ANSI C language and compiled with the ACE EXPERT C compiler. The standard communication libraries supplied with the compiler are applied for managing virtual topologies.

The communication scheme is based on a 2-dimensional grid virtual topology, as it can be mapped directly to the existing physical interconnection network. Use of a virtual topology is necessary as the selected asynchronous communication model requires previously created virtual links between the processors and the continuous creation and deletion of temporary virtual links would require a serious overhead. Hence an arbitrary number of virtual topologies can be used at the same time in the Parsytec GC system, an additional virtual topology does not interfere with other applications.

The communication protocol developed by T. Bartha and F. Balbach [18] is applied. Although it is a deadlock-free synchronous protocol, its main feature - the minimized communication delay - makes it usable in the case of asynchronous communication as well, to keep the memory overhead caused by the blocked communication threads low.

The Parsytec part of the program communicates with the CSP solver running on the host machine via standard UNIX sockets (the UNIX file I/O library for the PARIX kernel provides this possibility).

Besides of its obvious tasks, the low-level testing mechanism is also responsible for **fault injection**. Developing an appropriate *physical fault injection* method has proved to be exceptionally difficult due to the slight but - from this point of view - basic difference between T805- and T9000-based Parsytec systems. The main problem is the “emulation” of a dead transputer. The trivial solution for the T9000 is an infinite loop in a high priority thread; as it is impossible to interrupt, it effectively causes the transputer to ignore all communication. In T805 systems, however, an important part of the communication network - the routing chips - are emulated by software; therefore the high priority loop not only kills a transputer but causes a serious, mostly fatal failure in the whole inter-processor communication.

To overcome this problem, *logical fault injection* was employed. The injected faults does not force the actual processors to act as if they were faulty but they alter the operation of the tester mechanism. Fault patterns are generated by a program running on the host machine and they are sent to the low-level tester during initialization of the tester.

If a transputer is marked as **fault-free** in the fault pattern, its low-level tester operates correctly: sends and receives correct <I'm alive> messages, evaluates them and reports correct results to the host machine. A transputer marked **faulty** sends deliberately incorrect <I'm alive> messages to other transputers and reports random test results to the host; finally a transputer marked **dead** does not perform any communication with its neighbors and does not send any test results.

Another problem that is not completely solved is the clean-up after a test round (the complete exchange of <I'm alive> messages and sending reports to the host). As it was mentioned earlier, there is no way (at least no documented way) to terminate communication threads created by cancelled asynchronous communication requests. So at this moment the developed algorithm requires a complete reinitialization (global termination and restart of the tester application) after every test round.

4.4.2. CSP solver

The CSP solver part of the diagnosis algorithm runs on the host machine of the Parsytec GC, a Sun SPARCstation. It is written in ANSI C language and compiled with the GNU C compiler (V2.4.1) available under SunOS 4.1. GNU C was selected for its better code optimization and the application's greater performance compared to the standard C compiler of SunOS, and for its unique feature to optimize a program while retaining full symbolic and line number information for debugging purposes.

The CSP solver is based on a public domain CSP library [9]. This library is supplied in source code and is intended for solving static binary CSPs. Its practical properties, however, makes it usable for the kind of dynamic CSP that appears in the developed model. Moreover, it is written in C language so its adaptation into the algorithm is straightforward.

As a diagnostic purpose CSP solver is expected to be run fast and require a relatively small amount of resources, it should be written in a practically lower level, effectively compilable language. Due to the "benign" nature of the employed special class of CSPs, there is no need to apply a full-featured, general purpose CSP solver system. As the majority of

the known constraint-oriented languages and systems (CLPR, CONSTRAINTS, etc.) are strongly related to resource-hungry, interpreter-based languages like Prolog or LISP, their use does not seem to be reasonable in a diagnostic algorithm - a solver for special CSP classes, written in a more effective language like C, is much more promising.

The CSP library has an extra advantage for experimenting: over a dozen different backtracking strategies are implemented and they can be easily replaced for comparison benchmarks.

Constraints are represented by matrices. Each constraint is a $C_{k \times k}$ matrix where $\boldsymbol{k}$ is the size of the variable domains; c_{ij} is 1 if the $\boldsymbol{i}$th value of the first variable can coexist with the $\boldsymbol{j}$th value of the other variable and 0 otherwise. The solver operates on a fully connected constraint graph; the dummy edges can be replaced by "always-true" constraints. Therefore the full space requirement of the constraint graph is $n^2 \times k^2$ elementary storage units (it depends on the element type of the array containing the constraints; in the developed algorithm, the elementary storage unit is *byte* to avoid time-consuming bit-field operations) where $\boldsymbol{n}$ is the number of the variables.

The solver engine has some enhancements to exploit the special properties of the actual CSP. It considers only those variables that represent system components already engaged in a test relation for limiting the number of variables in the actual constraint problem (obviously considering those components that we have no information about is unnecessary) Moreover, it filters out those solutions that does not provide valuable information, although they conform with the actual set of constraints. For example, as long as a data link of a processor has not been involved in a test, it is unnecessary to care with its state; all the values representing a faulty state of this link can be left out from the domain of the variable representing the processor. Similarly, the state of the links of a dead processor is irrelevant. There are equivalent fault classes: no additional information is provided if each of them appears in the solution, so they can be left out, except one representant. (An example is illustrated in **Figure 4-3**.).

Another important enhancement in the CSP algorithm is the possibility of incorporating assumptions on the number of faulty components in the system. Although the constraint-based test diagnosis has no definite $\boldsymbol{t}$ limit for diagnosability, assuming a maximal number of faults (based on a priori knowledge about the system) can improve the effectivity of di-

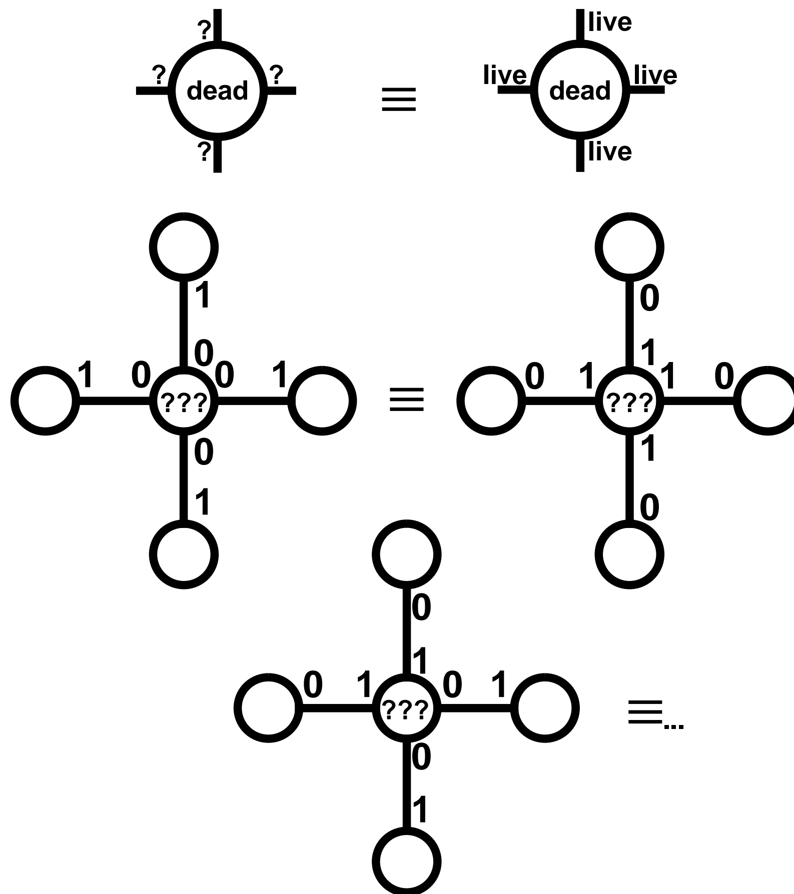


Figure 4-3. Examples for equivalent solution classes

agnosis by vastly decreasing syndrome decoding time. This assumption generates a **global contradiction** if the actual number of faulty units is higher than the assumed value, i.e. the CSP becomes unsatisfiable. This property can be used for implementation of *adaptivity*: the testing can be started with the assumption of a single fault and the limit can be increased if global contradiction was detected. Thus the algorithm becomes faster for low numbers of faulty units while preserving its diagnostic capabilities.

Besides of the effective CSP solution, the program part running on the Sun host is responsible for coordinating the test sequence. It communicates with the low-level tester via UNIX sockets. Upon initialization, a *GET_CONFIG* message is sent to each transputer; it queries the actual size of the Parsytec user partition and adjusts the number of CSP variables for the real number of transputers. A test sequence starts with generation of a fault pattern; it can be fixed for debugging purposes or random with a given maximal number of faults. The fault pattern is downloaded to the low-level part with *SET_ERROR* messages;

the routines running on each transputer adjust their operation according to the actual fault setting.

The Sun component initiates the exchange of <I'm alive> messages with a global *START_TEST* message and starts to receive test results from the Parsytec transputers. Each incoming syndrome bit is processed immediately: new constraints are generated and domains are adjusted, then the CSP solution runs again, displaying the possible solutions. After receiving the last test result and showing the final solutions, the low-level tester part is restarted with an *ABORT* message.

5 Test Results

5.1. Measurement Methods and Considerations

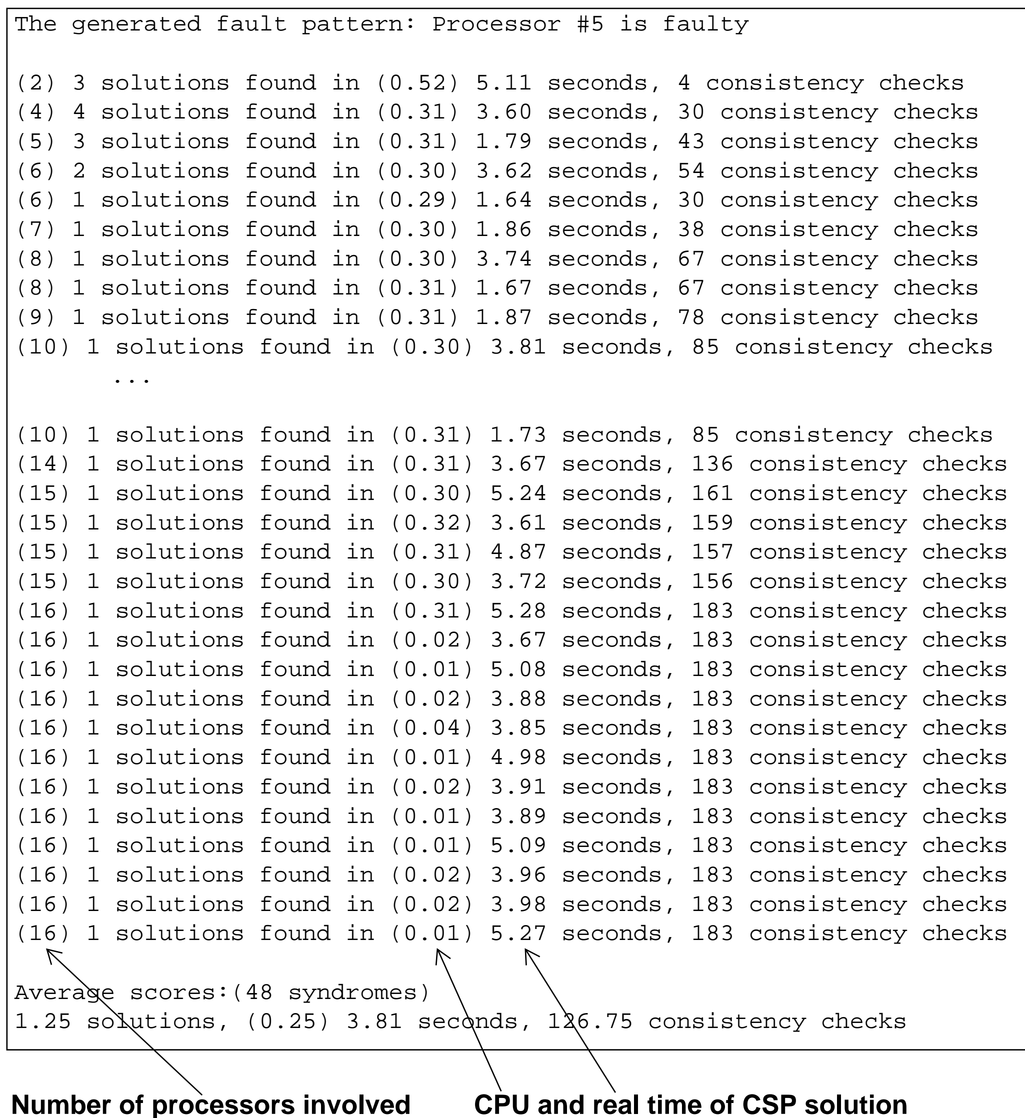
Performance evaluation of a diagnostic algorithm is always a complex task. The actual number of faults, the momentary workload of the diagnosed system and the system topology affect the quantitative characteristics of the diagnosis seriously. Consistent, correct and informative evaluation can be achieved only from a great volume of experimental results under various operational circumstances. Comparison with other reference algorithms is also necessary.

These requirements could not be completely fulfilled during the development. As reference algorithms with similar structural properties were not available, comparison experiments were impossible to perform. Due to the limited time of development and the difficulties resulting from the differences between the considered and the actual system (Section 3.4 and 4.4.1), only a limited number and spectrum of experiments could be performed. Moreover, theoretical analysis of the used CSP methods [10] became available only after finishing the evaluation.

In spite of these problems, some statistical and performance information was gathered from the experiments. The CSP solver part of the algorithm collected the data about the progress of the diagnosis and produced formatted reports (**Figure 5-1.**). These data - although they have only a limited value in absolute qualification - demonstrate the operation of the CSP solver and emphasize the advantageous properties of the whole algorithm quite well.

One of the greatest problems was to find an appropriate **time base** for the evaluation of the experimental results. As the processing of the incoming syndrome bits is performed se-

quentially, in the order of receive, the actual operating time of the constraint solver fluctuates between rather wide limits. The syndrome bit processing time is dominantly affected by the variable workload of the host machine and the relatively slow (4800 baud) serial connection between the host and the Parsytec system, which transferred the messages from the transputers to the CSP solver. The actual usable bandwidth of the serial connection was frequently decreased by the various terminal output messages that were also sent through the same connection. Moreover, the time-out limit of <I'm alive> messages applied for testing the Parsytec transputers was needed to be kept extremely high (2 seconds!) due to the performance limitations of the complex software-based router emulation on the T805 transputers.



```
The generated fault pattern: Processor #5 is faulty

(2) 3 solutions found in (0.52) 5.11 seconds, 4 consistency checks
(4) 4 solutions found in (0.31) 3.60 seconds, 30 consistency checks
(5) 3 solutions found in (0.31) 1.79 seconds, 43 consistency checks
(6) 2 solutions found in (0.30) 3.62 seconds, 54 consistency checks
(6) 1 solutions found in (0.29) 1.64 seconds, 30 consistency checks
(7) 1 solutions found in (0.30) 1.86 seconds, 38 consistency checks
(8) 1 solutions found in (0.30) 3.74 seconds, 67 consistency checks
(8) 1 solutions found in (0.31) 1.67 seconds, 67 consistency checks
(9) 1 solutions found in (0.31) 1.87 seconds, 78 consistency checks
(10) 1 solutions found in (0.30) 3.81 seconds, 85 consistency checks
      ...

(10) 1 solutions found in (0.31) 1.73 seconds, 85 consistency checks
(14) 1 solutions found in (0.31) 3.67 seconds, 136 consistency checks
(15) 1 solutions found in (0.30) 5.24 seconds, 161 consistency checks
(15) 1 solutions found in (0.32) 3.61 seconds, 159 consistency checks
(15) 1 solutions found in (0.31) 4.87 seconds, 157 consistency checks
(15) 1 solutions found in (0.30) 3.72 seconds, 156 consistency checks
(16) 1 solutions found in (0.31) 5.28 seconds, 183 consistency checks
(16) 1 solutions found in (0.02) 3.67 seconds, 183 consistency checks
(16) 1 solutions found in (0.01) 5.08 seconds, 183 consistency checks
(16) 1 solutions found in (0.02) 3.88 seconds, 183 consistency checks
(16) 1 solutions found in (0.04) 3.85 seconds, 183 consistency checks
(16) 1 solutions found in (0.01) 4.98 seconds, 183 consistency checks
(16) 1 solutions found in (0.02) 3.91 seconds, 183 consistency checks
(16) 1 solutions found in (0.01) 3.89 seconds, 183 consistency checks
(16) 1 solutions found in (0.01) 5.09 seconds, 183 consistency checks
(16) 1 solutions found in (0.02) 3.96 seconds, 183 consistency checks
(16) 1 solutions found in (0.02) 3.98 seconds, 183 consistency checks
(16) 1 solutions found in (0.01) 5.27 seconds, 183 consistency checks

Average scores:(48 syndromes)
1.25 solutions, (0.25) 3.81 seconds, 126.75 consistency checks
```


Figure 5-1. Details from a log file of a test run

Finally *the number of the received test results* was selected instead of a time base as it provided the most realistic base for characterizing performance.

The efficiency of a CSP algorithm could not be represented by the solution time, due to the above mentioned causes and the not-so-optimistic implementation. Therefore, as it is usual in the analysis of constraint solving algorithms, the *number of* the performed *consistency checks* was used for efficiency measures.

5.2. Performance Curves of Typical Test Runs

The progress of the CSP algorithm can be traced on the next graphs. They illustrate the number of processors involved, the number of found solutions and the number of consistency checks versus the number of received and processed syndrome bits.

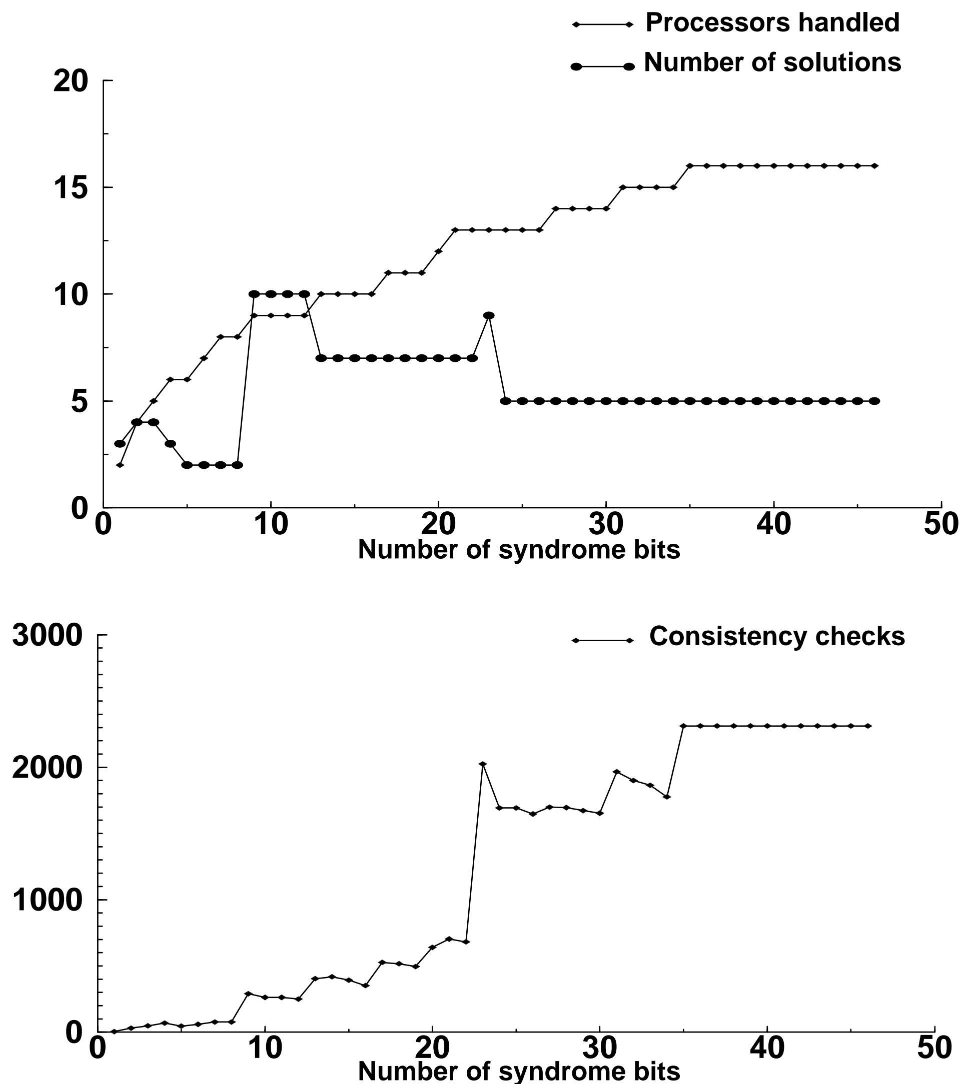
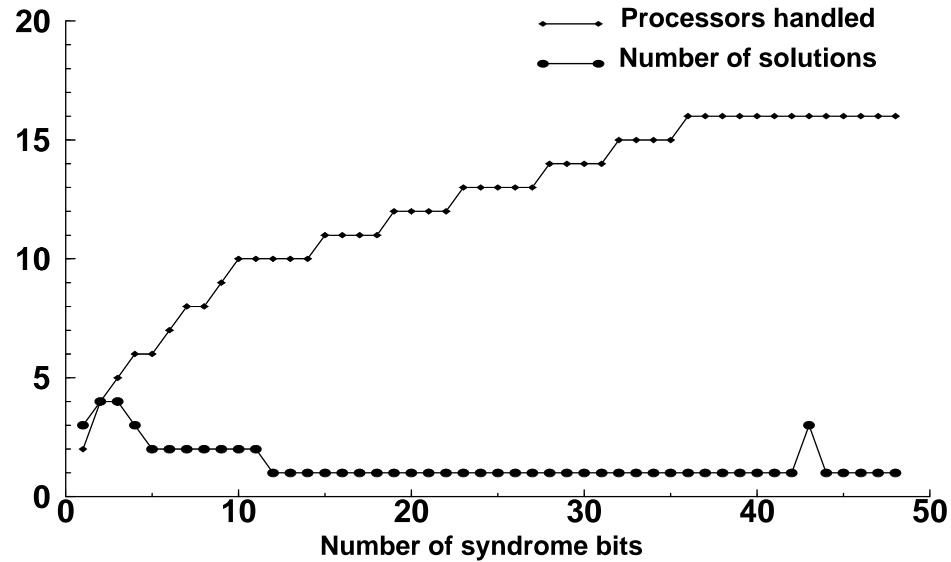


Figure 5-2. Progress curve of the CSP algorithm (1 processor was dead)

In **Figure 5-2.**, the generated fault pattern consisted of a single dead processor. The final number of solutions is 5 (one processor is dead, or a combination of its own data links and its neighbors' data links is broken so one processor is isolated from the system; the nature of the testing mechanism does not make differences between these cases). The fast convergence of the CSP algorithm is showed clearly.



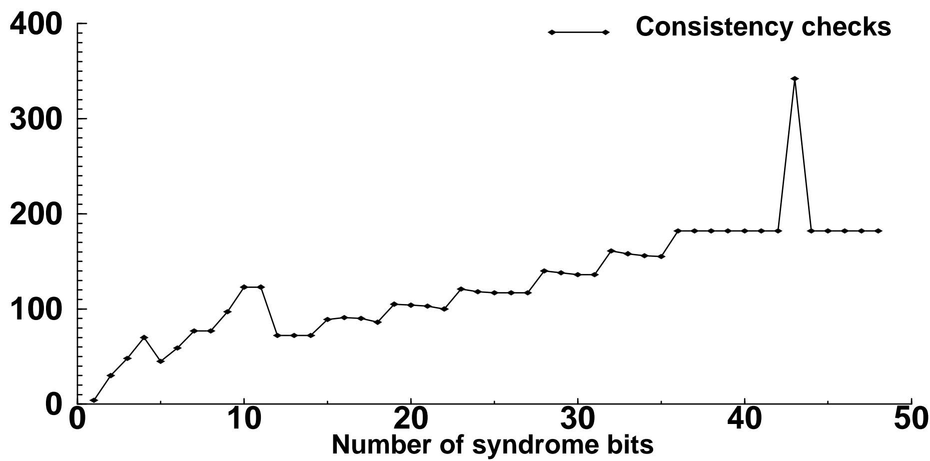


Figure 5-3. Progress curve of the CSP algorithm (1 processor was faulty)

In **Figure 5-2.**, the fault pattern contained a single faulty processor. The convergence of the CSP solver is more intensive because all the links were proven live.

6 Conclusions

6.1. Experiences

In this diploma work, a new concept of constraint-based system diagnosis was introduced. For validation of the theories, an experimental diagnosis algorithm was developed for multiprocessor systems and implemented on a Parsytec GCel machine. During development, emphasis was taken primarily on the following factors:

- maximal diagnostic efficiency (minimal diagnosis time);
- coverage of the widest possible range of faults;
- minimal communication overhead on the inter-processor communication lines;
- small memory requirements on the processors under diagnosis;
- minimal disturbance of other applications running on the same multiprocessor machine.

These factors are considered to improve the practical usability of the diagnosis algorithm. Experiences and usable ideas from earlier researches on the Parsytec system and from the studied literature were also intended to incorporate into the work where it was possible. A limited range of measurement and statistic facilities were also applied for collecting data about algorithm efficiency.

The constraint based diagnostic algorithm was extensively tested on the Parsytec GCel machine. During the tests, it proved its correct and effective operation and thus the correctness of the concepts behind it. The average syndrome processing time of the developed algorithm was about five times lower than a trivial exhaustive diagnosis, due to the

sophisticated backtracking techniques. Comparisons with algorithms based on different concepts, however, could not be performed.

Unfortunately, limitations of the Parsytec GCel system and its PARIX kernel could not be completely ignored. Due to the relatively low number of processing elements in the used system, efficiency could not be verified for greater systems.

6.2. Future work

Naturally, the developed diagnostic algorithm is far from being perfect. Some of the enhancement possibilities are proposed here:

- Testing of the algorithm on a *real T9000-based* Parsytec system, using an appropriate *physical fault injection* mechanism. (The technical conditions for this process was not available at the time of the development; however, hardware and software development efforts are being made towards it.)
- Implementation of the constraint-based algorithm in a *fully distributed environment* (the CSP solution is also performed **on the transputers**). This involves the new theoretical problem: distribution of test and inference results as well (due to the limited number of physical neighbors in the communication network, syndrome bits and optionally CSP results are necessary to send to/received from other processing elements);
- Implementation of the constraint-based algorithm in a *hierarchical partially distributed environment* (the CSP solution is performed by **the control processors in the C-Net**). This approach is very promising as it exploits the full range of fault tolerant capabilities of a Parsytec machine. Moreover, the hierarchical view is consistently present in the structural layout of a Parsytec GC: the C-Net itself can be also considered as a separate multiprocessor that needs some kind of (self-)diagnosis as well;
- Employment of *more sophisticated test* facilities, for the ability to use a *less pessimistic test invalidation* scheme instead of PMC. It requires further studies about the internal properties of the transputer and other system components;
- Determination of the *practical diagnostic capabilities* of constraint-based diagnosis. In the theory, it is proved in [2] that **all the deterministic information** from the test results is exploited during implication. Although it does not makes formulation of exact upper limits for the number of faults, it should be desirable to estimate the maximal number of diagnosable faults in this particular system topology;

- Selection of the *optimal backtracking strategy* for the CSP solver. The “graph-based backjumping” used in the current version of the algorithm was chosen only for its best experimental performance; a deeper theoretical analysis of the problem (what was impossible to perform within the very limited time period of development) may result the selection of an even better algorithm;

- Decreasing the *memory usage* of the constraint solver. In the current algorithm the static data storage representing the constraints required about 1 MByte; it was one of the main obstacles of CSP solver implementation on the Parsytec transputers. The space complexity should be reduced to an acceptable level while preserving the speed of the solution algorithms.

Bibliography

[1] A. Pataricza, K. Tilly, E. Selényi, M. Dal Cin: *A Constraint Based Approach to System Level Diagnosis*

[2] E. Selényi: *System Level Fault Diagnosis in Multiprocessor Systems with a General Test-invalidation Model,* Thesis, Technical University of Budapest, 1975.

[3] K. Tilly: *Constraint Based Logic Test Generation* (Ph.D. Thesis under preparation)

[4] J. Altmann: *Structure of the Fault Diagnosis Algorithm in the Esprit Project FTMPS*, Esprit Project Review, Aachen, Oct. 1993.

[5] U. Montanari: *Networks of Constraints: Fundamental Properties and Applications to Picture Processing*, Information Sciences vol. 7, 1974, pp. 95-132.

[6] R. Mohr, T. C. Henderson: *Arc and Path Consistency Revisited*, Artificial Intelligence, vol. 28 (1986), pp. 225-233.

[7] A. Mackworth, E. C. Freuder: *The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems*, Artificial Intelligence, vol. 25 (1985), pp. 65-74.

[8] R. Seidel: *A New Method for Solving Constraint Satisfaction Problems*, IJCAI '81, pp. 338-342.

[9] D. Manchak and P. van Beek: *A Binary CSP Solution Library for C Language* (Available by FTP from ftp.cs.ualberta.ca: /pub/ai/CSP)

[10] Grzegorz Kondrak: *A Theoretical Evaluation of Selected Backtracking Algorithms,* M.Sc. Thesis, University of Atlanta, Edmonton, 1994.

[11] E. Hyvvnen: *Constraint Reasoning Based on Interval Arithmetic*, IJCAI 1989, pp. 1193-1198

[12] M. Dal Cin: *Fault Tolerance: Theory and Practice* (lecture text), IMMD III, FAU Erlangen-Nürnberg

[13] M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, J. Hönig, W. Hohl, E. Michel and A. Pataricza: *Fault Tolerance in Distributed Shared Memory Multiprocessors*, Springer Lecture Notes on Computer Sciences, 1993.

[14] D. K. Pradhan: *Fault-Tolerant Multiprocessor and VLSI-based System Communication Architectures*, Fault-Tolerant Computing: Theory and Techniques, Prentice-Hall, New York, 1985, pp. 467-569.

[15] J. D. Russel and C. R. Kime: *System Fault Diagnosis: Closure and Diagnosability with Repair*, IEEE Trans. Comput., vol. C-24, no. 11., pp. 1078-1088, Nov. 1975

[16] J. D. Russel and C. R. Kime: *System Fault Diagnosis: Masking, Exposure and Diagnosability Without Repair*, IEEE Trans. Comput., vol. C-24, no. 12., pp. 1155-1161, Nov. 1975

[17] C. R. Kime: *System Diagnosis*, Fault-Tolerant Computing: Theory and Techniques, Prentice-Hall, New York, 1985, pp. 577-623.

[18] T. Bartha: *Diagnosis Algorithms of Multiprocessor Systems.* Diploma Thesis, Technical University of Budapest, 1993

[19] F. Preparata, G. Metze, R. Chien: *On the Connection Assignment Problem of Diagnosable Systems*, IEEE Trans. Comput., vol. EC-16, no. 6., pp. 848-854, Dec. 1967

[20] J. G. Kuhl and S. M. Reddy: *Distributed Fault Tolerance for Large Multiprocessor Systems*, Computer Architecture News 8, pp. 23-30, 7th Intl. Symposium on Computer Architectures, 1980.

[21] J. G. Kuhl and S. M. Reddy: *Some Extensions to the Theory of System Level Fault Diagnosis*, IEEE Proceedings 10th Intl. Symposium on Fault-tolerant Computing, pp. 291-296, 1980.

[22] C. S. Holt and J. E. Smith: *Self-Diagnosis in Distributed Systems*, IEEE Trans. Comput., vol. 34, no. 1., pp. 19-31, 1985.

[23] G. Bóna, I. Erényi and F. Vajda: *Többmikroprocesszoros rendszerek*. Müszaki Könyvkiadó, 1986.

[24] INMOS, Ltd.: *The T9000 Transputer Product Overview Manual* (preliminary information), April 1991

[25] Parsytec Computer GmbH: *Parsytec GC Technical Summary, V1.0*, 1991.

[26] Parsytec Computer GmbH: *PARIX V1.1 Programmer's Reference*, 1992.

[27] F. Barsi, F. Grandoni and P. Maestrini: *A theory of diagnosability in digital systems*, IEEE Trans. Comput., vol. C-25, no. 6., pp. 585-593, Jun. 1976

[28] S. Maheshwari and S. Hakimi: *On models for diagnosable systems and probabilistic fault diagnosis*, IEEE Trans. Comput., vol. C-25, no. 3., pp. 228-236, Mar. 1976

[29] G. Meyer and G. Masson: *An efficient fault diagnosis algorithm for symmetric multiple processor architectures*, IEEE Trans. Comput., vol. C-27, no. 11., pp. 1059-1063, Nov. 1978

[30] J. C. Laprie (ed.): *Dependability: Basic Concepts and Terminology*, IFIP WG 10.4: Dependable Computing and Fault Tolerant Systems Vol. 5, Springer-Verlag, Wien, 1992, pp. 11-44.

[31] F. Balbach: *Entwurf und Effizienzuntersuchungen von Fehlerdiagnoseverfahren für massiv parallele Rechnersysteme*, Diploma Thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, 1993.

[32] M. Abramovici, M. A. Breuer and A. D. Friedman: *Digital Systems Testing and Testable Design*, Computer Science Press, 1990.