Modeling of Fault-Tolerant Computing Systems [1]

Gy. Csertán[†], J. Güthoff[‡], A. Pataricza[†,‡], and R. Thebis[‡]

[†]Technical University of Budapest
Department of Measurement and Instrument Engineering
H-1521 Budapest, Műegyetem rkp. 9, Hungary
csertan pataric@mmt.bme.hu

[‡]University Erlangen-Nürnberg
Department of Computer Science, IMMD III
D-91058 Erlangen, Martensstrasse 3, Germany
jsguetho rhthebis@informatik.uni-erlangen.de

Abstract

Typical after design activities, such as reliability and performability evaluation, diagnostic development should be integrated into the design cycle of fault-tolerant computing systems in order to increase its effectiveness. A novel framework of various evaluation tools is presented in this paper, sharing a common input model, the high level behavioral description of the system.

The dataflow computational paradigm is used for this reason supporting both uninterpreted and interpreted modeling in a hierarchical way during the whole design cycle. According to this approach system design is a cyclic process, in which the system engineer stepwise refines and optimizes the system.

1 Introduction

The effective implementation of fault-tolerant digital computing systems largely depends on the integration of performance, reliability, and maintenability evaluation tools into the system design environment. However this integration results in modeling and data representation problems. The developers of these tools have to offer a concurrent engineering approach in order to overcome this problems.

The above mentioned tools should use a common basic model of the system during the whole system design process in order to avoid unnecessary model transformations. Since in different design phases the system must be described at different levels of detail, a modeling approach has to be chosen, which is able to handle different levels of abstraction. Hardware-software co-design, the most promising approach to cope with

[1]This research is part of the Hungarian-German Joint Scientific Research Project #70 with additional support from: SFB 182(DFG), Konrad Zuse Program (DAAD), OTKA-760, T-3394, and F7414 (Hungarian NSF)

design complexity, uses high level models in the early design phases. The final layout can be automatically synthesized from the final, evaluated and validated version of the design. Behavioral models have the advantage over gate level structural models, that their evaluation is less computation extensive.

The dataflow computational paradigm provides the hierarchical high level behavioral description of parallel asynchronous systems [Jon89]. In this modeling approach the system is decomposed into several processing units passing data between each other. Processing units are modeled by dataflow nodes and interconnections by channels, where the nodes are defined by their behavior. Since dataflow networks support both uninterpreted and interpreted modeling, they can be used during the whole system design cycle, e.g. in [BS93] dataflow networks are proposed for early validation of control systems.

Our aim is to show that this approach is a suitable background for modeling and designing fault-tolerant computing systems. For this reason we suggest a framework consisting of several simulation and analysis tools sharing a common model. Pre-defined (and pre-analyzed) system components are held in libraries and the system is modeled by simply connecting library components. Using this approach system design becomes an iterative process. After each step of model refinement the system engineer is able to evaluate the impacts of the last design changes on systems reliability, performability, and diagnosability. Thus finally an optimal design can be aimed at. Fault simulation and testability analysis can be done at the highest level of abstraction (uninterpreted model without timing). Detailed dependability evaluation is done by simulation in VHDL at lower levels of abstraction (timed uninterpreted and interpreted models) in order to determine the timing behavior (error latency) and performability of the system. Finally a reliability analysis can be performed with Petri nets. Since error simulation is an exhaustive method, only errors with the most serious effects or errors of the least reliable components, identified by reliability analysis, are taken into account.

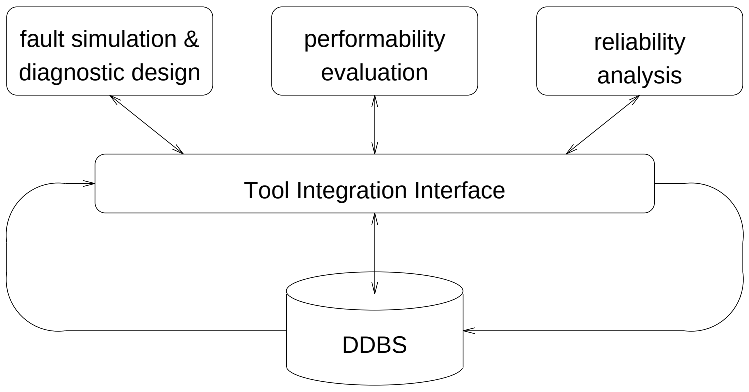

Figure 1: Overview of the Evaluation Framework

Figure 1 shows the idea of the framework. The Design Database stores all the design related information (library elements of components, evaluation results, etc.) The cooperating tools have parallel access through the Tool Integration Interface to the data stored in the database.

The paper is divided into 5 parts. In section 2 an example is given, that will be evaluated in subsequent sections. Section 3 presents testability analysis, in section 4 performability evaluation is dealt with, and finally section 5 describes reliability analysis.

2 Example Description

In this section we describe an example, which will be used to show evaluation aspects of testability, performability and reliability.

A CPU can execute write and read operations to the main memory. The main memory consists of a shadow RAM (RAM_1,RAM_2) including parity. The write operation sends data via the BUS to the component FORK. The FORK adds a parity bit to the data and stores both the data and the parity bit in the shadow RAMs. When executing a read operation a comparator checks the data parts. If the data is equal, it will be sent via the BUS to the CPU, if not, an additional unit takes the parity bits to choose the correct data. The parity fails, if the additional unit yields: both data parts correct or both data parts incorrect. The comparator and the additional unit together is called COMP+. The additional parity check is included to grant a non-stop operation in case of a single bit fault.

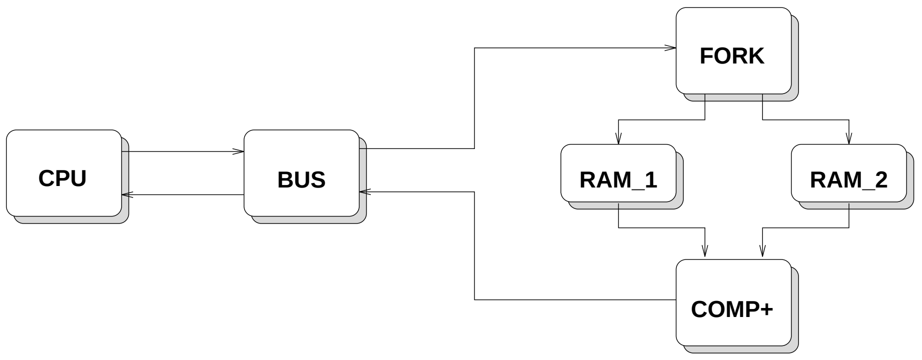

Figure 2: Example of System with Shadow RAM

3 Testability Analysis and Error Simulation

In this part of our work we show that the input model of Sheppard's integrated diagnostics [SS91] and the derived testability measures can be obtained from the dataflow model. "Testability analysis is the process of assessing the inherent ability (called testability) of a system to detect and diagnose faults using the prescribed test equipment and procedures." [Ofs91] Thus testability analysis has to be stimulus independent, and its results give an upper bound on measures gained when testing a system. Some of the most important measures are: percentage of detectable faults, percentage of uniquely diagnosable faults, hidden faults, dominant faults.

3.1 The Error Model for Testability Analysis

Faults are mainly hardware related and are usually modeled at a lower level of abstraction. Therefore at higher levels of abstraction the introduction of an error model is necessary. In the proposed modeling approach components are described by their error propagation behavior. Errors of a component are expressed by its state, which according to the black-box modeling method is identified by the rough classification of the result it delivers:

1. A component, which delivers correct data is said to be ok.
2. A component is said to be incorrect inc, if the data it delivers is incorrect.
3. If there is no response from a component it is said to be dead.

4. Diagnostic uncertainty can be expressed by assigning the state `x`.

Due to differentiating between the erroneous states `inc` and `dead` and due to modeling of uncertainty this error model can describe the system more precisely than traditional error models with only two states (error free/erroneous).

3.2 Error-Propagation in the Proposed Model

Testing of a system is nothing else than trying to propagate information from the component under test to the outputs of the system in order to examine the components state [ABF90]. Thus error propagation along a selected path of components is the background for testability analysis. In case of testability analysis, in contrast with test set development, the test path is predefined ("prescribed test set") and loop free. Our future work aims at extracting the test set from the dataflow model in a very similar way as it is done by classical (low level) test development algorithms, e.g. PODEM, D-algorithm [ABF90].

In the proposed approach the error propagation properties of the system are described by the behavior of the nodes of the dataflow network. This behavior is characterized by describing the:

- external errors of a component
- internal errors of a component
- built in error detection capabilities of a component
- fault propagation from the inputs of a component to the outputs

In Figure 3 the error propagation behavior of the dataflow nodes is given in a C-like notation. The different steps of the function `dataflow_node_behavior()` are:

`compute_external_effects` External errors of a component may occur due to the interaction with other components. In this step the effects of possible erroneous inputs are computed. For example an error free component can get erroneous when receiving an `inc` message. In case of more than one erroneous inputs the error with the most severe effect is selected and the state of the component is changed accordingly. Severity of errors can be different for different components.

`add_internal_errors` Internal errors are independent of the components environment (inputs). They occur randomly according to the current state of the component, and can lead to further changes in the state of the component. In this step effects of possible internal errors are modeled and the state of the component is changed. Due to an internal error an error free component can get erroneous, despite of the received `ok` message.

`detect_errors` Components with built in error detection capabilities are able to deliver test results. Errors of incoming messages (represent errors of other components) are signaled according to the components detection capabilities, depending on the state of the component and the type of errors. For example an error free component can signal incoming `inc` messages by `inc`, or an erroneous component may signal both `ok` and `inc` inputs by an `x`, modeling diagnostic uncertainty according to the PMC test invalidation model.

`propagate_errors` Input messages are propagated through the component. The type (`ok, inc, dead, x`) of output messages depends on the type of incoming messages and on the state of the component. An erroneous component for example may change an `ok` input message to an `inc` output message denoting its erroneous state.

```
dataflow_node_behavior()
  read_inputs;
  compute_external_effects;
  add_internal_errors;
  detect_errors;
  propagate_errors;
  write_outputs;
```

Figure 3: Error Propagation Behavior of a Dataflow Node (C-like Notation)

Evaluation of the model can be done by error simulation. In this method faults are propagated from the selected inputs (control points) of the system through the component under study to the primary or special test outputs (observation points) of the system. Since state of the system is composed from the states of the different components the number of possible system states can be very large: 3^N, where N is the number of components and 3 corresponds to the three possible component states. In order to avoid this complexity explosion the number of considered erroneous components has to be restricted. One way is turning on error injection (adding of internal errors) only at components, which are identified to be unreliable or to have serious effects on the system. A second solution is to stop the simulation when a given number of errors is reached - assuming t-diagnosability of the system.

3.3 Testability Analysis of the Example

The analysis aims at evaluating the testability properties of the comparator-shadow RAM unit as part of the example. In order to improve the testability of a system test paths have to be loop free. For this reason during the test cycles of the network must be cut. Figure 4 shows the unfolded (loop free) version of the example introduced in Figure 2.

In this experimental setup for error propagation the control point is the `BUS`, from which testing starts, while observation points are the `COMP+` component and the `BUS` respectively. `COMP+` itself is able to detect errors, thus it is a test output, while the `BUS`, on which further error detection is possible is a primary output. In the following test on the `BUS` is referred to `TOB`. Since the aim of testing is detecting the errors of the RAMs the external memory test `TOB` can be done by an independent test component or in our case by the `CPU`. For simplicity `dead` states are not modeled and as result of the reliability analysis the `FORK` and `COMP+` are supposed to be error free.

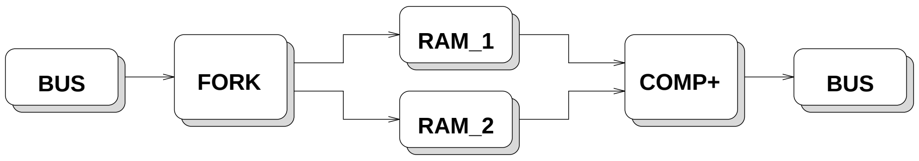

Figure 4: The Unfolded Model of the Comparator-Shadow RAM Part of the Example

The model was evaluated by error simulation. Results of the simulation are given in Figure 5. In the upper half of the figure the test results signaled by `COMP+` are shown while in the bottom part errors observable on the `BUS` are given . In the right column the state of the system is given and in the left column the corresponding test results are shown. When interpreting the results it is assumed that error detection done by `TOB` is perfect, i.e. all the errors reaching the `BUS` are signaled by `TOB`.

DETECTED BY COMP+:	
ok	(bus.ok)(fork.ok)(mem_a.ok)(mem_b.ok)(comp+.ok)
inc	(bus.ok)(fork.ok)(mem_a.ok)(mem_b.inc)(comp+.ok)
inc	(bus.ok)(fork.ok)(mem_a.inc)(mem_b.ok)(comp+.ok)
ok	(bus.ok)(fork.ok)(mem_a.inc)(mem_b.inc)(comp+.ok)
ok	(bus.inc)(fork.ok)(mem_a.ok)(mem_b.ok)(comp+.ok)
ok	(bus.inc)(fork.ok)(mem_a.ok)(mem_b.inc)(comp+.ok)
ok	(bus.inc)(fork.ok)(mem_a.inc)(mem_b.ok)(comp+.ok)
ok	(bus.inc)(fork.ok)(mem_a.inc)(mem_b.inc)(comp+.ok)

OBSERVABLE ON THE BUS:	
ok	(bus.ok)(fork.ok)(mem_a.ok)(mem_b.ok)(comp+.ok)
x	(bus.ok)(fork.ok)(mem_a.ok)(mem_b.inc)(comp+.ok)
x	(bus.ok)(fork.ok)(mem_a.inc)(mem_b.ok)(comp+.ok)
inc	(bus.ok)(fork.ok)(mem_a.inc)(mem_b.inc)(comp+.ok)
inc	(bus.inc)(fork.ok)(mem_a.ok)(mem_b.ok)(comp+.ok)
inc	(bus.inc)(fork.ok)(mem_a.ok)(mem_b.inc)(comp+.ok)
inc	(bus.inc)(fork.ok)(mem_a.inc)(mem_b.ok)(comp+.ok)
inc	(bus.inc)(fork.ok)(mem_a.inc)(mem_b.inc)(comp+.ok)

Figure 5: Results of Error Simulation

It can be seen that whenever only a single error is present on the inputs of `COMP+`, it can be detected. But if errors are present on both inputs, `COMP+` fails to signal the errors. The bottom part of the figure shows that `COMP+` causes uncertainty in error propagation, since in case of one erroneous input it may deliver either correct or incorrect results (denoted by `x`). It can also be seen that examining the errors on `BUS TOB` can not differentiate between the error of `BUS` and that of `RAM_1` or `RAM_2`. It results problems when diagnosing the system.

Inspecting the results we can see that to each system state different from "system is error free" (identified by error free components) a test outcome different from `ok` or `x` can be assigned. It means that each component error can be detected by some of the tests, moreover any combination of the errors can also be detected, thus error detection ratio of the system is 100%. On the other hand we can not combine the test outcomes in such a way that a given combination corresponds only to one and only one system state (theoretically in case of two tests two states would be diagnosable). It results that none of the component errors can be uniquely diagnosed, thus isolation level (ratio of uniquely diagnosable errors) is 0%. It results that all derived measures, such as ratio of hidden errors, dominant errors, etc. are also 0. However, if we were satisfied with localization of an error within the memory, and we suppose a single error model, then memory error would be diagnosable. If one evaluates the results of the two tests independently and compares them, it turns out, that adding an external memory test to the built-in test of

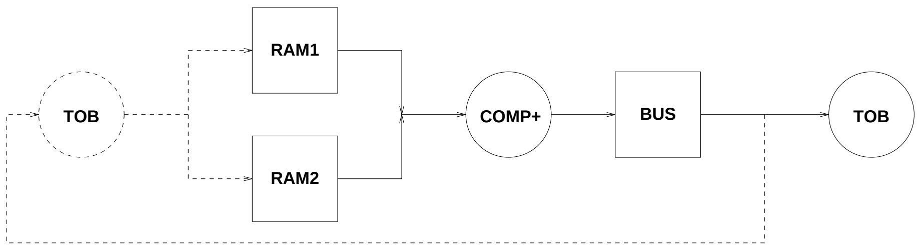

Figure 6: Input for Sheppard's Integrated Diagnostics

`COMP+` does not improve the testability of the system. We can overcome this problem by either turning off `COMP+` during the test, or by improving the testability capabilities of `COMP+`.

Finally Figure 6 defines the input model for Sheppard's integrated diagnostics. It gives the error-test and test-test dependency relationship, which is presented in form of a dependency graph. Note that dotted lines denote the cycle during normal operation. In the figure circles denote the tests, and boxes the fault isolation conclusions, in our case they are the errors of components. Arrows denote the dependency relationship between conclusions and test: for example if `RAM1` fails then test `COMP+` will fail. When there exists a path from test A to test B it denotes, that if test A fails then test B will fail. We can see that an arrow leads from test `COMP+` to `TOB`, which examining the simulation results in Figure 5 shows that if `COMP+` fails then `TOB` also fails. Extracting of this model from our dataflow notation makes it possible to use the theoretical background of Sheppard, and to obtain further testability measures of the system [SS92].

4 Performability Evaluation

In this section we will give some insights how we build a model and how we evaluate the data gathered form the model to do performability evaluation. In general, when modeling a system for analysis we first have to fix what aspects of the system's behavior we are interested in and what kind of data we want to gather from the model in order to estimate the desired true characteristics of the SUI (system under investigation). Furthermore, we must determine what aspects of a complex real-world system actually need to be incorporated into the model.

The technique we use to evaluate a model numerically is simulation and was inspired through a previous work of Aylor et al [Sch92]. They used VHDL [IEE87] as basis for modeling computing systems and introduced the 'building block approach' as a modeling concept for evaluating the system performance and rudimentary for reliability analysis. Unfortunately, the offered blocks used to model a computing system are tailored for evaluating system performance, mainly, and are hard to handle when building simulation models. But, the main idea behind the building block concept and the intention by using VHDL as the description language for modeling computing systems at different levels of abstraction are picked up in this work and extended for performing performability evaluation.

4.1 Analysis Definition

From our point of view, performability evaluation serves as basis to improve the dependability of computing systems with respect to performance. By dependability we mean the reliability and availability of a SUI while safety and security attributes are not considered. When evaluating the performability of a SUI we have to estimate both, the system dependability and the improvement gained through the integration of DIMs (dependability increasing methods). The questions depicted in Table 1 can be seen as a checklist for what must be analyzed when evaluating performability. The table is divided into two parts, the first concerns with dependability in general while the second is concerned with evaluating the improvement.

1. How does the SUI perform its function in the presence of faults?
2. How sensitive is the SUI to specific faults?
3. How fast is a specific fault propagated through the system?
4. How long does it take before a specific fault is becoming active?
5. How high is the dependability profit after the integration of DIMs?
6. How strong is the system performance influenced by DIMs?

Table 1: Questions of Interest

4.2 Simulation Model

The performability of a computing system depends primarily on the quality of its HW and SW components. But, performability depends also on the type of application processed by a computing system because the SW reflects how a system will be used and how the HW resources are brought into action – the SW determines the operational profile of a computing system. For example, an application which works primarily on processor registers is more dependable than an application which extensively uses external memory (MTBF ratio memory:CPU = 1:12500 [Sch94]). Obviously, the application must be regarded as well as the HW of a computing system when evaluating the system performability.

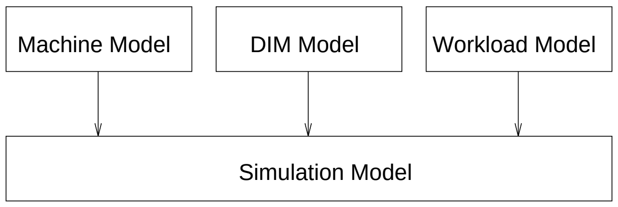

Figure 7: Building a Simulation Model for Performability Analysis

In performance evaluation of network systems the SW and HW components are separated into different models. In [Her92] a separation of the system's model in machine-model and workload-model was suggested, but this modeling concept isn't sufficient for performability analysis. Rather, it is reasonable that the simulation model consists of three parts, machine-model, workload-model, and DIM-model. The machine-model and the workload-model represent the SUI without any DIMs. This permits the evaluation of the "naked" computing system and serves as a reference for evaluating performance loss and dependability profit after the integration of DIMs. The DIM-model contains all components (HW and SW components) which are necessary for improving the dependability of a SUI.

4.3 Evaluation Technique

To evaluate the dependability of the SUI fault simulation is done. Faults are provoked through a fault-injector and the resulting impairments, errors and failures, are observed. The data gained in the analysis of the naked system are used to determine interesting scenarios for doing fault simulation. In opposite to other commonly used high level fault simulation techniques, where time and location for a fault injection is determined by using a pseudo random number generator, in this approach time and location for a fault injection are fixed founded on the knowledge of the operational profile.

Based on the assumptions listed in Table 2 tracing of information flow within the simulation model is obvious. Whenever components perform an action that consumes

• Faults are always getting active through interaction.
• Failures are distinguished in timing and value failures (failure domain viewpoint [Lap92]). The duration of such a failure can be temporary or permanent.
• Only value failures can be propagated through a computing system, while timing failures reside in the component where the timing failure becomes active. An exception are those timing failures which cause a value failure.

Table 2: Assumptions about Fault Activation, Fault Propagation, and Failure Modes

time for processing and whenever service requests are deposited or received, a trace record is written into a trace file. The current error state of both, interaction entity[2] and component, is stored in specific data structures and can be one of `ok`, `faulty known`, `faulty unknown`, or `dead`. These states reflects the actual state of the SUI and must be distinguished from SUI's viewpoint about its current state. I.e. assume that an interaction entity is corrupted due to fault injection but the corruption isn't already detected by the SUI. The actual state of the SUI is `faulty unknown` while from SUI's viewpoint the current state is `ok`.

After the simulation run ends, the trace file is investigated by using a visualization tool. ParaGraph [Hea93], originally a tool for visualizing the performance of parallel programs, was modified to allow the visualization of error traces. Animation, fault propagation, fault coverage, fault latency, and various utilization diagrams show the system engineer how the system behaves in presence of faults. He can immediately observe which components are directly affected and, by stepping through the trace, where the error is propagated and how fast it is propagated. Furthermore, he can investigate if a specific error is detected by the integrated DIMs, how long this error is kept undetected in the system and how high the additional workload is due to error detection/correction.

For questions of interest where visualization isn't appropriate (i.e. how many components are visited by an interaction entity) the system engineer can run several queries. SHQL [Mom94] is a program that reads SQL commands interactively and executes those commands by creating and manipulating Unix files. In our approach it is used to get some strategic information from the trace file – e.g. the result of a query could provide a listing of the most frequently used components. Unfortunately, this public domain version doesn't support the whole SQL standard, so supplementary some AWK [AWK88] scripts are offered for queries not supported by SHQL.

4.4 Example

Recall the example in section 3.3. In order to evaluate the performability of the proposed system timing description must be incorporated into the simulation model. Furthermore, the kind of information carried by the interaction entities is meaningful because evaluating whether two incoming data replica are equal determines essentially the system behavior. Therefore, an interpreted model is needed. For simplicity it is assumed that the workload can be modeled through a Markovian process. Remembering the modeling concept introduced in section 4.2 the structure of the corresponding simulation model is: A description of the underlying fault model will be presented in section 5.1.

In Figure 8 a snapshot of the visualized trace file can be seen. Looking at the fault propagation diagram a system engineer can directly observe the timing behavior of each

[2] An interaction entity is used to facilitate exchange of information between components. Possible interaction entities are tokens, messages etc.

Machine model	DIM model	Workload model
CPU	—	Markovian process
BUS	—	—
RAM	FORK, RAM, COMP+	—

component (horizontal lines) and time and location of interactions (arrows). Different line styles are used to determine the current state of interaction entities and components. I.e. the dotted arrow signals that the interaction entity is in error state, more precisely `faulty unknown`. The faulty entity is forwarded to the COMP+ which in this case is able to tolerate the error. How strongly the system work load is influenced can be read from the utilization summary diagram. The light shading of component BUS indicates that this component has pure transport delay while the other components have pure inertial delays. Inertial delays are used to model usage time of a facility and transport delays model time used for transmitting interaction entities.

The evaluation of the simulation model shows that the proposed system is able to tolerate all memory faults appearing in one RAM where an odd number of bits is corrupted. In this case a non-stop operation of the Shadow RAM can be guaranteed. Other statistically relevant types of faults are at least detectable. Faults which arise in other components can not be tolerated anyway and cause a system failure directly. The performance is reduced by an average of 9.4 percent due to the integration of FORK and COMP+.

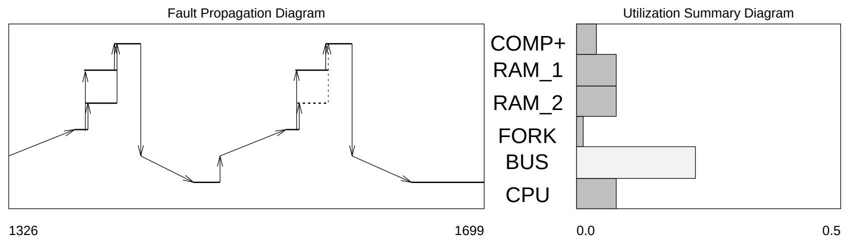

Figure 8: Fault Propagation and Utilization Summary Diagram

5 Reliability Evaluation

Petri Nets (PN) allow the description and analysis of systems with parallel behavior. We use stochastic PN, which are transformable to a Markov chain, to compute required reliability measures like MTTF, availability, coverage etc.. There are two solution types of a Markov chain: the analytical and the simulative. The analytical solution is prefered, because it is an exact one; a steady state or transient analysis is possible.

The analytical solution of a stochastic PN permits to evaluate systems with quite different frequencies of events, therefore it is possible to investigate the effect of very rarely events, like faults, to the behavior of a system. Sometimes quite different frequencies of events cause a stiffness problem [BDMC^{+}94] for most of the Markov chain solution methods. In that case we use the new analytical Multi Level (ML) solution method [HL94], which has no stiffness problem.

In the following example, memory and bus faults are rare events. The effect of different fault types to a memory unit is considered. The system uses a shadow RAM for fault detection and parity bits for recovery; the structure is described in section 2. This system is compared to a system with a single memory (incl. parity). The system is modeled by the PN–type GSPN [MC84].

5.1 Fault Model

The fault model only considers data faults. The data can be corrupted by bus faults or in the component RAM. Two data elements (3 Bits plus a parity bit each) are used to distinguish between the fault types; one bit and multiple bit faults are represented. The following fault types are integrated into the model:

- A bus error sets the affected bits to 1 with probability 0.5 and to 0 with probability 0.5.
- Write operation to the RAM: A interrupted bitline prevents from writing a bit into a cell (stuck-at); connected cells (coupling faults) get the same bit value; a alpha particle inverts a cell after a period of time (bitflip).
- Read operation to the RAM: The bit in the point of intersection of a interrupted bit- and wordline will be read, if there is a read access to a cell in this bitline.

No fault models for `FORK` and `COMP+` is chosen, because the faults of `FORK` and `COMP+` errors are describable by the effects of the other faults and in addition, the probabilities of the effects of the other faults are much higher, than probability of the failure of `FORK` and `COMP+` errors.

5.2 Petri Net Model

In the PN (Figure 9) a reference "word" from memory is observed. The content of the reference variables (original data) is written to the reference word. Data from the reference word (read operation), which is sent back via the bus to CPU, is compared with the original data and permits an evaluation of the system.

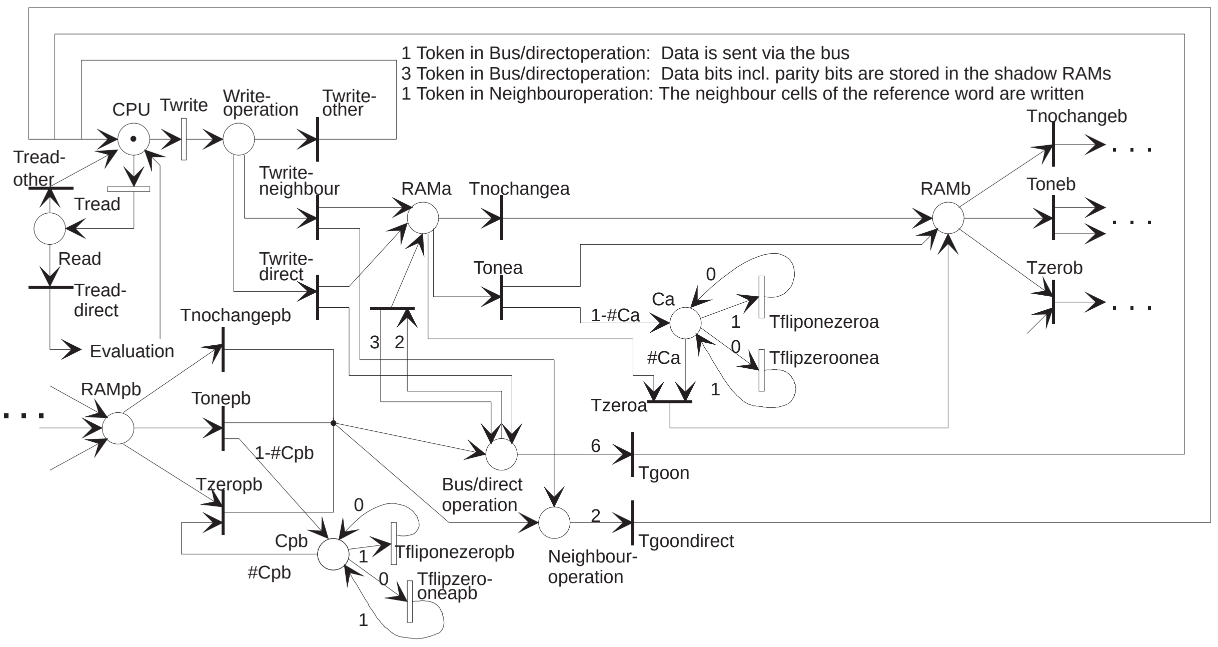

Figure 9: PN model of the Shadow RAM–Parity combination

The CPU executes a write (Twrite) or a read (Tread) operation. There are three write decisions: An access to the reference word (Twritedirect), to the neighbor cells of the reference word (Twriteneighbour) or to the rest of the cells (Twriteother) and two read decisions: An access to the reference word (Treadirect) combined with the evaluation part of the net or to the rest of the cells (Treadother).

Altogether there are eight data cells Cx. x stands for a,b,c,pa,e,f,g,pb. Ca,Cb,Cc are the data bits of the reference word in the first shadow RAM inclusive parity bit Cpa. Ce,Cf,Cg have two meanings: First of all Ce,Cf,Cg are the data bits sent via the bus to the reference word, secondly Ce,Cf,Cg are the data bits of the reference word in the second shadow RAM inclusive parity bit Cpb.

There are four possible actions to change a data cell Cx: The content of the cell is unchanged (Tnochangex), bit value one (Tonex) or zero (Tzerox) is written or a cell is inverted (Tfliponezerox, Tflipzeroonex). One token in Cx means bit value one, no token in Cx bit value zero.

The probabilities of the fault types and of the correct behavior are assigned to the actions, for that purpose the transitions have a condition part to consider the actual state.

5.3 Simulation, Experiment and Evaluation

The PN model has different parameters to describe an experiment. One can determine the rate for read and write operations of the CPU, the probabilities and rates of the fault types of a cell and finally the size of the RAM (number of addresses). The used transitions and places for the evaluation permit to compute the following probabilities for the reference word of the shadow RAM–parity combination:

1. Equalprob, (Unequalprob): The data bits of the both words are identical, (not identical).
2. Cor1prob, (Cor2prob): The data bits of the both words are not identical and the parity comparison yields: the data of the first (second) shadow RAM is correct and the data of the second (first) is incorrect.
3. Errorprob: The data bits of the both words are not identical. The parity comparison is not able to determine the correct word; the comparison tells, that both words are correct or both incorrect.
4. CorData, (IncorData): The word, which is sent via the bus to the CPU (case 1, 2), is correct (incorrect).

In comparison with the last model a single RAM inclusive parity for fault detection is investigated. The following probabilities of the reference word are computed:

1. CC: The read data is correct (comparison with the reference variables) and the parity bit comparison yields: the data is correct.
2. CI: The read data is correct and the parity bit comparison yields: the data is incorrect (inverted parity bit).
3. IC: The read data is incorrect and the parity bit yields: the data is correct (two bit fault).
4. II: The read data is incorrect and the parity bit yields correctly: the data is incorrect.

The input parameters of both models are based on the following assumptions: The computer has 20 MIPS with a mean workload of 10 per cent and every twentieth instruction a read/write operation to the main memory takes place. With 75 per cent a read and with 25 percent a write operation is executed. The experiment parameters are: RAM size (10 words); neighbors of the reference word (4); Writerate (2.5e+4); Readrate (7.5e+4); Buserrorprob (1.0e-6, Exp. 1); Nowriteprob (1.0e-6, Exp. 2,6); Connectprob (0.25e-6, Exp. 3,6); Readfaultprop (0.25e+6, Exp. 4,6); Bitfliprate (1.17e-3, Exp. 5,6).

The probabilities and rates of the faults are assigned to the single cells. From experiment one to five only one fault type is considered. In experiment six all fault types (without bus errors) are integrated. In Table 3 we present the computed results of the single RAM model and the results of the shadow RAM–parity combination:

	Single RAM				Shadow RAM–parity			
Exp.Nr.	CC	CI	IC	II	Unequalprob	Errorprob	Cor1(2)prob	IncorData
1	9.99e-1	0.00e+0	1.50e-6	0.00e+0	0.00e+0	0.00e+00	0.00e+0	1.50e-06
2	9.99e-1	5.00e-7	1.50e-12	1.50e-6	3.00e-6	6.00e-12	1.50e-6	7.61e-13
3	9.99e-1	”	3.38e-12	1.50e-6	3.00e-6	1.25e-11	1.50e-6	1.69e-12
4	9.99e-1	”	2.63e-12	1.50e-6	3.00e-6	1.05e-11	1.50e-6	1.32e-12
5	9.99e-1	”	3.05e-12	1.50e-6	3.01e-6	1.22e-11	1.51e-6	1.53e-12
6	9.99e-1	1.00e-6	7.57e-12	3.00e-6	9.02e-6	7.41e-11	4.51e-6	9.27e-12

Table 3: Results of the single RAM and the shadow RAM–parity combination

The evaluation of the results show:

The use of the shadow RAM–parity combination leads very late to a fail stop of the system, in contrast to a single RAM. For example experiment six shows a fail stop state after the mean time of 4.4 seconds for the single RAM, and after 50 days for shadow RAM–parity combination.

The probability to send incorrect data bits back to the CPU is infinitely small (IncorData 9.27e-12) for the shadow RAM–parity combination. The probability to send back an incorrect word can be neglected, because it is more probable (factor 8) to reach a fail stop state, instead of sending an incorrect word. The single RAM detects only an odd number of faults, the probability of a two bit fault is very small, therefore the probability to send incorrect data bits back to the CPU is infinitely small (IC 7.75e-12).

There are differences in the cases IC of the single RAM (experiment 2 to 5). The ratio of a coupling fault to a stuck-at is 2.25:1, a readfault to a stuck-at 1.75:1 and a bitflip to a stuck-at 2:1.

Bus errors are not detected in both models.

6 Conclusion

In this work we investigated the problem of modeling fault-tolerant computing systems. Typical questions of interest for such systems are concerned with testability, dependability, and reliability and are usually treated completly independent. In our approach results of different evaluation cycles are assessed globally which allows a trade-off between testability, dependability, and reliability. Futhermore, the evaluation time is significantly reduced since results gained in one evaluation cycle are re-used in another.

References

[ABF90] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, New York, 1990.

[AWK88] V. Aho, P.J. Weinberger, and B.W. Kernighan. *The AWK Programming Language.* Addison-Wesley, 1988.

[BDMC$^+$94] Buchholz, Dunkel, Mueller-Clostermann, Sczittnick, and Zaeske. *Quantitative Systemanalyse mit Markovschen Ketten.* Teubner, Stuttgart - Leipzig, 1994.

[BS93] A. Bondavalli and L. Simoncini. Functional Paradigm for Designing Dependable Large-Scale Parallel Computing Systems. In *Proceedings of the International Symposium on Autonomous Decentralized Systems, ISADS '93*, pages 108–114, Kawasaki, Japan, 1993.

[Hea93] M.T. Heath. Paragraph: A tool for visualizing performance of parallel programs. Technical report, Oak Ridge National Laboratory, University of Illinois, 1993.

[Her92] U. Herzog. Network Planning and Performance Engineering. Technical report, University of Erlangen-Nürnberg, Department of Computer Science (IMMD VII), 1992.

[HL94] G. Horton and S. Leutenegger. A multilevel solution algorithm for steady-state markov chains. In *Proceedings of SIGMETRICS 94*, 1994.

[IEE87] *IEEE Standard VHDL Language Reference Manual*, IEEE Standard 1076–1987, 1987.

[Jon89] B. Jonsson. A Fully Abstract Trace Model for Dataflow Networks. In *Proceedings of the 16th ACM symposium on POPL*, pages 155–165, Austin, Texas, 1989.

[Lap92] J. C. Laprie. *Dependability: basic concepts and terminology.* Springer-Verlag, 1992.

[MC84] M. A. Marsan and G. Conte. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 1984.

[Mom94] B. Momjian. Shql. ftp root@candle.uucp, 02 1994.

[Ofs91] S. Ofsthun. An approach to intelligent integrated diagnostic design tools. In *Proceedings of the IEEE Systems Readiness Technology Conference*, Anaheim, California, 1991.

[Sch92] J. M. Schoen, editor. *Performance and Fault Modeling with VHDL*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[Sch94] W. Schlenz. *MTBF-Zeiten für typische Systemkomponenten im gehobeneren PC-Leistungsbereich.* 1994.

[SS91] W. R. Simpson and J. W. Sheppard. System Complexity and Integrated Diagnostics. *IEEE Design & Test of Computers*, 8(3):16–30, September 1991.

[SS92] W. R. Simpson and J. W. Sheppard. System Testability Assessment for Integrated Diagnostics. *IEEE Design & Test of Computers*, 9(1):40–54, March 1992.