Design for Testability with HW-SW Codesign

Gy. Csertán, A. Pataricza and E. Selényi

Technical University of Budapest

Department of Measurement and Instrument Engineering

H-1521 Budapest, Műegyetem rkp. 9, Hungary

e-mail: csertan@mmt.bme.hu

Abstract

Current trends in the development of design automation tools aim at a radical increase in productivity by offering highly automated design tools. As applications include even critical control applications, dependability becomes to an important design issue.

A novel approach supporting concurrent diagnostic engineering using a data flow behavioral description is presented in this paper. The basic idea of this new method is the extension of the descriptions of the functional elements with the models of fault effects and fault propagation at each level of the hardware-software codesign hierarchy, thus allowing design for testability of digital computing systems.

Using the presented approach test generation can be done concurrently with the system design and not only in the back-end design phase as it had been done previously. For test generation purposes the generalized forms of the well known logic gate level test design algorithms can be used.

Keywords: diagnostic design, testability, test generation, PODEM, data flow, HW-SW codesign

1 Introduction

The advent of low-cost implementation technologies of application specific circuits opens new horizons for custom-tailored solutions. The availability of low-cost, but highly complex off-the-shelf programmable components (PLDs) and ASIC technologies allows for the use such a background even for small enterprises, and not only for the market leaders in state-of-the-art technologies, like some five years ago. Recent efforts aim the reduction of cost and time of the design tasks by developing integrated environments for system engineering. These offer various tools for the computer architects and circuit designers based on a homogeneous tool-box and common engineering database for the whole design process. An important characteristic of such tools is that activities earlier performed only after the final engineering design are pushed forward into an early design phase, thus allowing a radical shortening of the design-feedback loop. Practical experiences show a 1:20 reduction in design time, while the resulting hardware overhead due to the automated design is as low as 40%. Moreover the use of automated design technologies radically improves the product's design quality. One such design approach is Hardware-Software Codesign (Fig. 1), that denotes "the joint specification, design, and synthesis of mixed HW-SW systems" ([BBC$^+$93, RB95]).

A main insufficiency of these tools originates in the lack of an integrated support for the follow-up phases of dependability analysis. This becomes crucial in safety related applications, like process control and automation. The avoidance of costly re-design cycles needs the pushing of diagnostic design (test generation, testability analysis), into early phases of system design as well. In [SS94] a method is presented for doing testability analysis as part of integrated diagnostics in early design phases, but the problem of generating and designing of the test set remains still unsolved.

The aim of our work is the development of a tool-box for model-based diagnostic and dependability evaluation in the form of an extension of the existing functional design tools. The basic models and technologies developed are fully coherent with those used in the original tools in order to keep the integrity of the design environment and avoiding unnecessary model transformations.

The basic idea of the methodology is as follows:

1. A system is modeled at the highest level of abstraction of the functional design process usually by data flow models [Sch92, BS93]. Only the flow of data and the processing-related delay times are modeled in the form of token flows without any description of the individual data transformation in the components (Level 1 and level 2 *uninterpreted modeling* in Fig. 1). This phase aims primarily at performance analysis and optimization and it is supported by formal analysis methods, e.g.

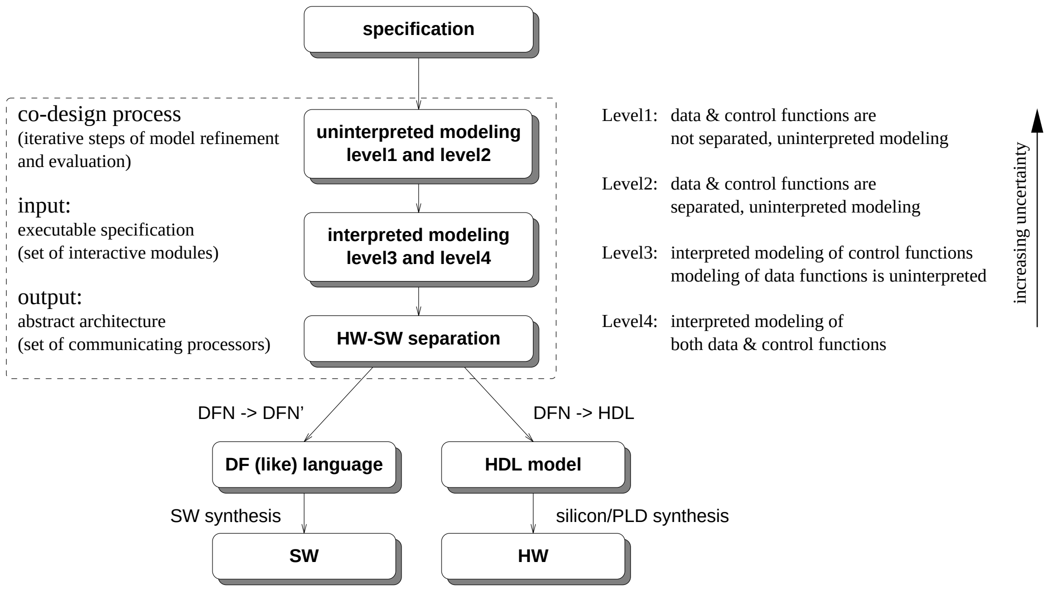

specification
co-design process
(iterative steps of model refinement and evaluation)
input:
executable specification
(set of interactive modules)
output:
abstract architecture
(set of communicating processors)
uninterpreted modeling level1 and level2
interpreted modeling level3 and level4
HW-SW separation
Level1: data & control functions are not separated, uninterpreted modeling
Level2: data & control functions are separated, uninterpreted modeling
Level3: interpreted modeling of control functions modeling of data functions is uninterpreted
Level4: interpreted modeling of both data & control functions
increasing uncertainty
DFN -> DFN'
DFN -> HDL
DF (like) language
HDL model
SW synthesis
silicon/PLD synthesis
SW
HW

Figure 1: HW-SW codesign process

on the basis of automatic translations into timed Petri-nets.

2. More and more structural and functional details are added by stepwise refinement into this initial model thus defining increasingly precisely the system's structure and the data processing functions of its components. (Level 3 *mixed uninterpreted-interpreted modeling* in Fig. 1).
3. Finally, when all component functions become fully defined (Level4 *interpreted modeling* in Fig. 1), hardware-software separation can be done and the automatic or interactive hardware and software synthesis processes can be started.

The presented approach is based on the idea of extending the data flow notation by incorporating faults and fault effects. This extended notation will be used in the modeling phases of HW-SW codesign, thus fault related information can be gained concurrently with the system design by the evaluation of this composite model.

In uninterpreted modeling the tokens representing the data can be marked either as correct or as erroneous. A superset of the fault propagation paths can be estimated by tracing their flow from the fault site in the network. Due to the simplifications all elements are assumed to propagate potentially all faults, (no data dependencies are modeled) only necessary conditions can be estimated, but even this over-pessimistic results can be still used for an effective control strategy in the test search procedures in more detailed models.

Later, after introducing data dependencies at the mixed and interpreted models costly heuristic or structural test generation algorithm must be invoked for the final decision. However the high-level dependability analysis provides not only an inexpensive way for comparative analysis of alternative constructs, but serves as a tool for test strategy design.

In previous works [CGPT94, Cse94, CPS95] and in this work it is shown that the following problems can be solved using the presented approach:

- fault simulation
- test generation, fail-safe test generation
- estimation of optimal diagnostics strategies
- testability analysis for both built in and maintenance tests
- failure modes and effects analysis (FMEA)

The paper is organized as follows: Section 2 introduces the modeling approach, and presents a simple system and its model as an example. In

Section 3 a representative of the family of test pattern generation algorithms is presented, and a test is generated for the example. Finally Section 5 contains concluding remarks and a short overview of the future work.

2 The Modeling Approach

2.1 The fault model

Faults are mainly hardware related and usually modeled at a lower level of abstraction. Therefore it is necessary to introduce an error model at higher levels of abstraction. Since in uninterpreted modeling data dependencies are undefined, it has to express uncertainties due to the neglected data dependencies. In the proposed approach a multi valued fault model is used instead of the stuck-at gate level fault model. Its advantage is the high expressive power for the description: the quite complex functional units can be described more precisely and various other requirements, like safe testing, can be considered. A potential multi-valued fault model can be defined: according to the black-box modeling approach, component faults are identified by the rough, and for the sake of the compactness, simplified classification of the results they deliver:

- `ok` message denotes that the component delivered correct computational result
- `inc` message denotes that the component delivered incorrect data
- `dead` message will be sent, if the component, due to a fatal fault, does not deliver results at all

- `x` message is used to express uncertainty. The correctness of the result depends on the actual data values received by the component and on the actual implementation of the component (for a given data value it would be `ok`, for another it would be `inc`)

2.2 The data flow notation

The data flow notation, proposed in [Jon89], is well-suitable for conceptual modeling of computing systems in the early design phases [BS93], for early validation of computing systems [BBS93], and for performance evaluation [CBBS94].

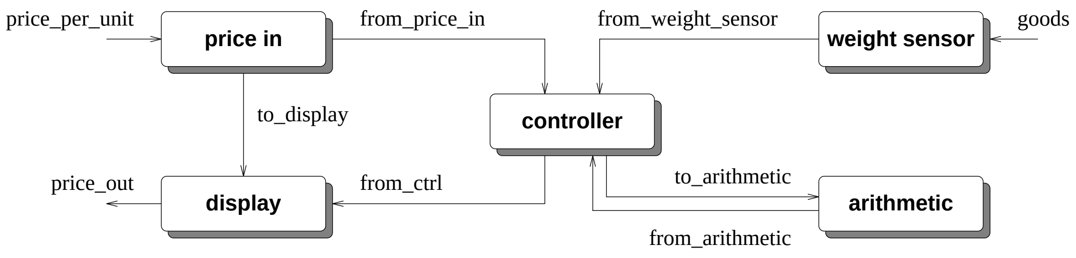
A *data flow network* N is a set of nodes P_N, which execute concurrently and exchange data over point-to-point communication channels C_N. The *data flow node* represents the functional elements of the system. The signal propagation attributes of an element are described by a simple relation between inputs and outputs, eventually depending on the previous state of the node. Note, that as the correlation of the inputs and outputs is described by this relation in a weaker form than by an input-output function, this behavior can be also non-deterministic. The *channels* of the data flow network symbolize the interaction between the functional elements of the system. Internal channels link two nodes. Input (output) channels connect a single node to the outside world representing the primary inputs (outputs) of the system. *Communication events* occur when data items (subsequently called tokens) are inserted into an input channel (input event describing the arrival of some data to the primary inputs) or data items are removed from an output channel (output event denoting the appearance of results on a primary output of the system).

The functional behavior of a node p is defined by a set of firing rules R_p. S_p defines the set of possible states of the node. A node is ready to execute as soon as the data required by one of its firing rules are available and the node is in a proper state. The meaning of firing rule $f \in R_p$, denoted by $f = (s, X_{in}, s', X_{out})$ is that if the node p is in state $s \in S$, each of the input channels $i \in I_p$ holds at least $X_{in}(i)$ data items, then firing rule f is potentially selected for execution. The execution of firing rule f removes $X_{in}(i)$ data items from each input channel $i \in I_p$ and outputs $X_{out}(j)$ data items on each output channel $j \in O_p$, while the node changes its state from s to s'.

2.3 An example

The selected example is very simple due to space limitation and can not introduce the full modeling power of the presented approach (refer to [Cse94]). The system is an intelligent scales, that can calculate the price of goods according to its weight and to the unit price. Modeling is done at the highest level of abstraction (uninterpreted modeling). The fault model is restricted to single internal faults, that can be one of the following:

- `eq-more` identifies the fault when a component delivers a result, which is either equal to or larger then the correct one. Actually in our case it is considered to as a fault free result.



INTELLIGENT SCALES:
P_N ={price in, weight_sensor, controller, display, arithmetic}
C_N={price_per_unit, from_price_in, from_weight_sensor, goods, to_display, to_arithmetic, from_arithmetic, from_ctrl, price_out}

WEIGHT SENSOR:
I={goods}
O={from_weight_sensor}
S={eq-more , less, dead}
R={f1 ... f8}

f1=(eq-more; goods=eq-more; eq-more; eq-more->from_weight)
f2=(eq-more; goods=less; eq-more; less->from_weight)
f3=(eq-more; goods=dead; eq-more; x->from_weight)
f4=(eq-more; goods=x; eq-more; x->from_weight)
f5=(less; goods=eq-more; less; less->from_weight)
f6=(less; goods=less; less; less->from_weight)
f7=(less; goods=dead; less; less->from_weight)
f8=(less; goods=x; less; less->from_weight)

Figure 2: Data flow model of the intelligent scales

- token `less` is sent by a component if it delivers a result, that is less than the correct one.
- `dead` denotes that a component does not deliver results at all.
- `x` expresses the uncertainty when either `ok/more` or `less` could be sent.

We assume, that the system has no built-in fault detection capabilities. From the point of view of the shopkeeper fault `less` is of the greatest severity since in this case the price paid by the customer is less than the value of the goods. The data flow graph of the system and the formal notation of one of the nodes are shown in Fig. 2. (Note that if it is necessary the fault `ok/more` could be split into two faults `ok` and `more`.)

The system consists of 5 parts: **price in** reads in the price per unit of the goods from a keyboard and sends it to the controller and to the display as well. Malfunctions of the component are: not delivering result (e.g. due to a broken wire), or delivering faulty result `less`. The **weight sensor** measures the weight of the goods and sends the results to the controller. The weight sensor always sends result, but it can be either `ok/more` ore `less`. The **controller** in the first step of its functioning receives the weight and the price per unit of the goods and delivers them to the arithmetic unit. In the second step the computed price received from the arithmetic unit is forwarded to the display. The controller can deliver either `ok/more` or `less` results, or it can be even `dead`. The **arithmetic** unit is responsible for computing the price of the goods from the price of the unit and from the weight. When the component is faulty computation results can be incorrect or it is possible that the component does not deliver results at all. Finally the **display** displays the price per unit and the price of the measured goods. The display can have one of the faults `ok/more`, `less`, `dead`.

Inputs of the system are: `price_per_unit`, `goods`, while `price_out` is the output of the system. The initial state of fault-free components is ok_0. A verbal interpretation of some firing rules of the `weight_sensor` node (Fig. 2) is:

- f1- During a fault-free functioning this rule describes the component. Since only fault-free messages `eq-more` are received and the component does not have any internal fault, it remains in fault-free state `eq-more`.
- f2- Describes the fault propagation of the fault-free component: if the input message, received from `goods` is faulty `less`, the result will also

be faulty `less` and it will be delivered into `from_weight`.

f5- Due to an internal fault the sensor measures the goods faulty. The result of the measurement is less then the weight of the goods, and the faulty result is delivered to the controller via `from_weight`.

3 Test Design in HW-SW Codesign

The base of effective fault detection and diagnostics is a well planned test strategy. In this section we will show that test strategy design can be done concurrently with system design by using a data flow model based automatic test pattern generation (ATPG). The presented algorithm is a generalized form of logic gate-level test pattern generation algorithms. The idea of generalization arises when considering the correspondence between the two models:

- Similarity to the gate and module-level stuck-at fault model, where faults are modeled at the output of logic gates. Errors of a functional data flow node are manifested at the outputs in the form of erroneous messages.
- The behavior of a data flow functional element is described by a transfer relation, similarly to the truth or state transition tables of logic gates and modules.
- The model may contain loops that, just like in case of sequential logic have to be cut and an iterative array model can be constructed in both cases [ABF90].

- Since components can have states, the testing of a system has to start from a predefined initial system state. (In practical data flow models examined till yet there was no need for the search of a self-initialization sequence.)

We will exploit this correspondence and present the high-level version of a gate-level ATPG algorithm. As a representative example we selected the well known PODEM algorithm [ABF90, Goe81] that is widely used for test generation for stuck-at faults in logic circuits.

3.1 The PODEM algorithm

In order to generate a test for a given fault the problem of test generation is recursively divided into the subproblems of: implication and checking; line justification; fault propagation. Implication and checking aims at the reduction of the problem space, line justification is responsible for setting the primary inputs (PIs) according to a given line and fault propagation tries to propagate the state of a line to the primary outputs (POs). The PODEM (Path-Oriented Decision Making) algorithm (Fig. 3) is characterized by a direct search process: it directly manipulates the PIs and tries to propagate the fault to the POs. In each step of the algorithm checking and implication is done. To keep track the still open problems a set is maintained during the algorithm: the *D-frontier* contains the gates from the outputs of which the fault has to be propagated towards the POs. The advantage of Podem() over other test pattern generation algorithms is that due to the direct search that:

- no consistency check is needed
- the J-frontier can be eliminated
- backward implication are not necessary

In the proposed approach solution of the subproblems is slightly different from the original one:

- Due to the multi-valued fault model `eq-more, less, dead, x` values are used instead of 0 and 1. It means that instead of values D (1 in the good, 0 in the faulty circuit) and $\overline{\mathrm{D}}$ (0/1), fault-pairs `eq-more/less, eq-more/dead, eq-more/x, less/eq-more, less/dead, less/x, dead/eq-more, dead/less, dead/x` are propagated.
- Instead of the truth table firing rules are used. Possible actions depend on the state of the component. States of the component have to be consistent in subsequent blocks of the iterative array model (predecessor and successor states).
- Checking has to ensure that the constraints imposed by the global testing requirements, i.e. safe testing, are fulfilled.

Test generation starts with initialization of the channels, where the value `ND` (not defined) is assigned to each channel. After the initialization the Podem() procedure is called (Fig. 3). In each step when Podem() is executed some checking occurs, a PI is selected, implication is done, and Podem() is called recursively again to check the results of the implication step. The activities of the Podem() procedure can be outlined as:

Step 3 the stop criterion is checked, e.g. if a fault pair has been propagated to a PO, test generation is successful.

```
1:  PODEM()
2:  begin
3:    if (error at PO) then return SUCCESS
4:    if (test not possible) then return FAILURE
5:    k=Objective()
6:    j=Backtrace(k)
7:    for (v=all possible faults)
8:    begin
9:      Imply(j,v)
10:     if (PODEM()=SUCCESS) then return SUCCESS
11:   end
12:   return FAILURE
13: end
```

Figure 3: The PODEM algorithm

Step 4 if no test can be generated, Podem() has to be stopped. This case is when:

- the target fault can not be activated, since a different value has been propagated to the output of the faulty component.
- no error propagation step can be done, since the D-frontier is empty

Step 5 an objective (a channel) for error propagation is selected. Usually it is a channel from the D-frontier.

Step 6 a PI being in connection with the selected channel are selected.

Step 7–12 All possible faults are probed at the PIs in order to fulfill the objective by implications. If none of the probes are successful Podem() returns failure and another PI (according to Step 6) has to be selected and probed again.

In each step Podem() is executed two other procedures are called: *Objective()* selects a channel to which a fault pair has to be propagated. For this

```
1: Objective()  /* fault is n=f */
2: begin
3:   if (all output of n is ND) then N=n
4:   else select a node N from D-frontier
5:   select one input m of N
6:   return m
7: end
```

Figure 4: Procedure Objective()

reason in:

Step 3,4 a component is selected. It is either the component a test has to be generated for or it is a component from the D-frontier.

Step 5 a still unassigned (it has a value `ND`) input of the node is selected.

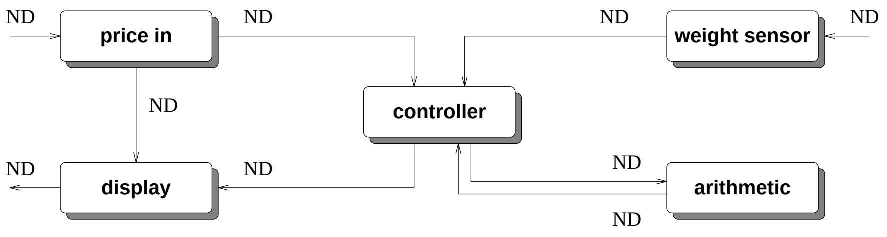
```
1: Backtrace(k)
2: begin
3:   while (k is an output)
4:   begin
5:     select an input j of node n /* k is an output of n */
6:     k=j
7:     end
8:   return k
9: end
```

Figure 5: Procedure Backtrace()

The other procedure *Backtrace()* is responsible for finding the PIs, with which adjustment a fault pair has to be propagated to the selected channel:

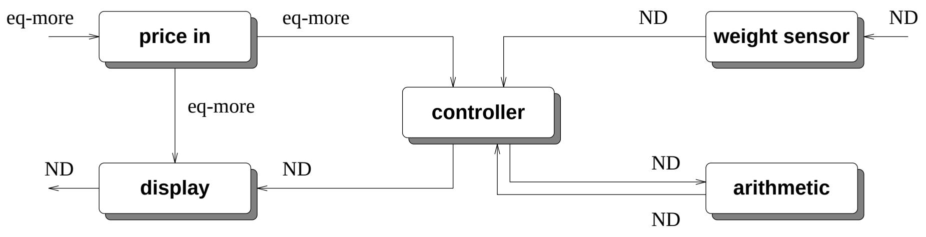
Step 3–7 A search is done toward the PIs of the data flow modeled system. To an output of a component an input is assigned. It will denote the implication path from the PI to the selected objective.

Step 0:



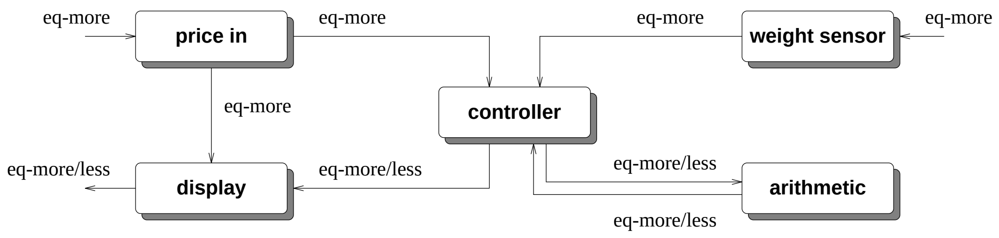
Step 1:

Objective()=from_price_in
Backtrace()=price_per_unit
Imply() -> price_per_unit=eq-more, to_display=eq-more, from_price_in=eq-more
D={controller}



Step 2:

Objective()=from_weight_sensor
Backtrace()=goods
Imply() -> goods=eq-more, from_weight_sensor=eq-more, to_arithmetic=eq-more/less, from_arithmetic=eq-more/less,
from_ctrl=eq-more/less, price_out=eq-more/less
D={display}



Step 3:

SUCCESS

Figure 6: Test generation for `less` fault of the controller

3.2 Test generation for the example

To enlighten the previously defined algorithm, test generation is shown in detail for the `less` fault of the controller component in the simple example. Steps of the test generation are presented in Fig. 6 step-by-step. Note that identifier of channels are omitted!

Steps of test generation can be explained as:

Step 0 Initialization. `ND` is assigned to all channels. Test generation can be started.

Step 1 First call of the Podem() procedure. Since the POs have not been reached yet, Objective() and Backtrace() are called. In this step all the outputs of the controller unit are `ND`, thus the objective is channel `from_price_in`. Backtrace identifies the PI `price_per_unit`. Afterward implication is done, but no error pairs appear, thus the D-frontier remains empty.

Step 2 After the implication of the 1st step, Podem() is called again. This time the objective is channel `from_weight_sensor`. Backtrace now identifies the other PI of the system: `goods`. As a result of implication an error pair appears on the output channel of the display component, that is now element of the D-frontier.

Step 3 Third, last call of Podem(). Checking detects the error pair `eq-more /less` at the PO `price_out`, thus test generation is finished successfully.

The result means, if the controller has a `less` fault, it can be detected by measuring a known weight. (Prize must also be typed correctly.)

4 Conclusion and Future Work

In this work we presented a modeling approach which can be used in the early phases of HW-SW codesign. It supports testability and dependability analysis in such a way that it becomes an integral part of the design process, since in the proposed data flow model both the functional and fault propagation/fault effects information are incorporated. By means of a simple example we have shown that even in this phase of the design test strategy design and testability analysis can be done concurrently with the system design.

Future work incorporates the implementation of an environment in which dependable hardware-software codesign can be done. For this reason the Ptolemy design environment, developed at the University of California at Berkeley, will be used.

References

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

[BBC$^+$93] G. Boriello, K. Buchenrieder, R. Camposano, E. Lee, R. Waxman, and W. Wolf. Hardware/Software Codesign. *IEEE Design and Test of Computers*, pages 83–91, March 1993.

[BBS93] C. Bernardeschi, A. Bodavalli, and L. Simoncini. Dataflow Control Systems: An Example of Safety Validation. In *Proceedings of SAFECOMP'93*, pages 9–20, Poznan, Poland, 1993.

[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.

[CBBS94] Gy. Csertán, C. Bernardeschi, A. Bondavalli, and L. Simoncini. Timing Analysis of Dataflow Networks. In *Proceedings of the 12th IFAC Workshop on Distributed Computer Control Systems, DCCS'94*, pages 153–158, September Toledo, Spain, 1994.

[CGPT94] Gy. Csertán, J. Güthoff, A. Pataricza, and R. Thebis. Modeling of Fault-Tolerant Computing Systems. In *Proceedings of the 8th Symposium on Microcomputers and Applications, uP'94*, pages 95–108, October Budapest, Hungary, 1994.

[CPS95] Gy. Csertán, A. Pataricza, and E. Selényi. Dependability Analysis in HW-SW codesign. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium, Erlangen, Germany*, April 1995.

[Cse94] Gy. Csertán. Dependability Analysis in HW-SW Codesign. Technical report, Institute of Computer Science III, University of

Erlangen-Nürnberg, Martenstr. 3, D-91058 Erlangen, Germany, 1994. In preparation.

[Goe81] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, C-30(3):215–222, March 1981.

[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.

[RB95] J. Rozenblit and K. Buchenrieder, editors. *Codesign.* IEEE Press, 1995.

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

[SS94] W. R. Simpson and J. W. Sheppard. *System Test and Diagnosis.* Kluwer Academic Publishers, 1994.