Dependability Analysis in HW-SW Codesign[1]

Gy. Csertán A. Pataricza E. Selényi

Dept. of Measurement and Instrument Eng.
Technical University of Budapest
H-1521 Budapest, Műegyetem rkp. 9, Hungary
e-mail: csertan, pataric, selenyi@mmt.bme.hu

Abstract

The increasing complexity of todays computing systems necessitates new design methodologies. One of the most promising methods is hardware-software codesign, that supports unified hardware-software modeling at different levels of abstraction, and hardware-software synthesis. As applications include even critical applications, dependability becomes to an important design issue.

A novel approach for the underlying modeling in hardware-software codesign 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 error propagation at each level of the hardware-software codesign hierarchy.

From the extended system model various dependability measures can be extracted. This paper concerns test generation, solved by a generalized form of the well-known logic gate level test generation algorithms and extraction of the input model of integrated diagnostics, allowing testability and diagnosability analysis of the system.

Keywords: diagnostic design, integrated diagnostics, testability, test generation, dataflow, 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-shell programmable components (PLDs) and ASIC technologies allows for the use of such a background even for small enterprises, and not only for the market leaders. Unfortunately the complexity offered by this technologies can not be dominated by traditional design methods any more. Recent efforts aim at solving this problem and reducing the cost and time of the design task by providing new design methodologies and developing integrated environments for system engineering. These offer various tools for the computer architects and circuit designers based on a homogeneous toolbox and common engineering database for the whole design process. An important characteristic of such tools is that activities performed earlier 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 new design approach is hardware-software codesign (Figure 1), that denotes "the joint specification, design, and synthesis of mixed HW-SW systems" [4].

A main insufficiency of this tools (like Ptolemy [8], COSMOS, SpecSyn) originates in the lack of an integrated support 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 [11] a method is presented for testability analysis as part of integrated diagnostics in early design phases, but the problem of generating the input model of this method and designing of the test set remain still unsolved.

The aim of our work is the development of a toolbox for model-based diagnostics 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.

In this paper a novel approach is presented, that uses the dataflow notation as the modeling methodology of HW-SW codesign. Using this approach the behavior of the functional units of digital computing systems can be hierarchically described and aspects of faults, their effects, and error propagation can be handled during the design process. As it is shown in [6] and in this work, the following major problems can be solved concurrently with system design:

- fault simulation
- test generation, including generation of fail-safe

[1]This research is part of the EC Research Project #CP93:9624 FUTEG (FUnctional TEst Generation and diagnostics) with additional support from: Hungarian-German Joint Scientific Research Project #70 and OTKA T-3394 (Hungarian NSF)

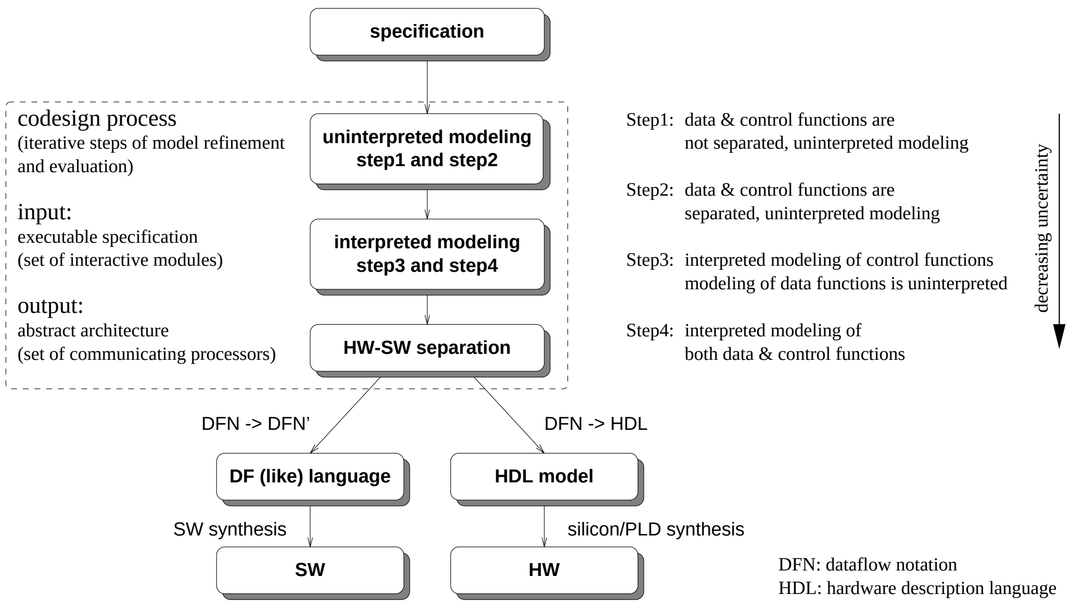

specification
codesign process
(iterative steps of model refinement and evaluation)
input:
executable specification
(set of interactive modules)
output:
abstract architecture
(set of communicating processors)
uninterpreted modeling step1 and step2
interpreted modeling step3 and step4
HW-SW separation
Step1: data & control functions are not separated, uninterpreted modeling
Step2: data & control functions are separated, uninterpreted modeling
Step3: interpreted modeling of control functions modeling of data functions is uninterpreted
Step4: interpreted modeling of both data & control functions
decreasing uncertainty
DFN -> DFN'
DFN -> HDL
DF (like) language
HDL model
SW synthesis
silicon/PLD synthesis
SW
HW
DFN: dataflow notation
HDL: hardware description language

Figure 1: HW-SW codesign process

tests
- estimation of optimal diagnostic strategies
- testability analysis for both built-in and maintenance tests
- failure modes and effects analysis (FMEA), risk analysis

The paper is organized as follows: Section 2 introduces the modeling approach, and presents a simple system and its model as example. In Section 3 a representative of the family of test pattern generation algorithms is presented, and a test is generated for the example. Section 4 deals with the extraction of the integrated diagnostics model illustrated by some results of the analysis of the example. Finally Section 5 contains concluding remarks and a short overview of the future work.

2 The Modeling Approach

Since the application area of HW-SW codesign is very broad, no general solution can be found for all problems. Usually a vertical slice is taken handling a partial problem in depth and solution is given for that particular aspect. We focus our main attention on the design of digital computing and control systems, where dependability is of primary concern. In this case the system is modeled at the highest level of abstraction of the functional design process usually by dataflow models [10, 3]. 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 (Step 1 and Step 2 *uninterpreted modeling* in Figure 1). This phase aims primarily at performance analysis and optimization usually supported by formal analysis methods, like analysis after an automatic translation into timed Petri nets.

More and more structural and functional details are incorporated through stepwise refinement into this initial model thus defining increasingly exactly the system's structure and the data processing functions of its components. (Step 2 *mixed uninterpreted-interpreted modeling* in Figure 1). After the refinement uncertainty within the model decreases and analysis results become more exact.

Finally, when all component functions become fully specified (Step 4 *interpreted modeling* in Figure 1), separate automatic or interactive hardware and software synthesis processes can be started in parallel.

Our approach is based on the idea of modeling the fault effects and their propagation similarly to the flow of data in the functional model. In uninterpreted modeling the tokens representing the data can be marked either as correct or as faulty. A set of error propagation paths can be estimated by tracing their flow from the fault site in the network. As uninterpreted modeling does not handle data dependencies, at the highest level of abstraction diagnostic uncertainty has to be introduced in order to express conditional error propagation. This way the simulation and test generation algorithm delivers a superset of propagated fault effects in the system. In the model all potential consequences of a fault are incorporated. In the subsequent steps of model refinement this global overview of the system effectively supports test generation procedures by radically restricting the search space to the solutions of those from the coarse model.

Using a multi-valued logic a more detailed fault model and a more precise description of the reactions of functional units to erroneous input values can be defined. Therefore a global overview of the system testability and diagnostics can be estimated. As example, the tokens and component fault states can be qualitatively grouped (colored) according to the severity of the fault effects into categories like:

- catastrophic (causes a damage in a component)
- fatal (blocking the further operation, e.g. an undetected wrong opcode input of a CPU-like element)
- incorrect (may invoke only error propagation, like wrong input data to be processed by the CPU)

In such a way not only invalidation relations and potential test paths can be estimated, but a fail-safe test process can be designed as well by incorporating the inhibition of the propagation of catastrophic errors into the goals of the test generation algorithm.

It should be pointed out, that the use of other guiding attributes in this user-defined colorings of the tokens and propagation rules offers full freedom for the analysis of different user requirements.

By adding fault occurrence, fault latency and detection probabilities the model can serve as a starting for a more detailed dependability analysis.

Moreover a more detailed description allows to handle uncertainty: at a higher level the unknown behavior of a component is identified by only one color. Later this color can be split into many shades, representing the behavior of the component in more details, without modifying the structure of the component.

Later, when introducing data dependencies at the mixed and interpreted models costly heuristic or structural test generation algorithms 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 and can control the used heuristics. Remember, that the reduction of uncertainty during successive model refinement monotonically restricts the solution space.

2.1 The Dataflow Notation

The dataflow notation is well-suitable for conceptual modeling of computing systems in the early design phases [3], for early validation of computing systems [2], for performance evaluation, if extended with the notion of time [5], and for being the modeling base of HW-SW codesign [4]. In this work the asynchronous dataflow notation, introduced in [7], is used.

The proposed notation meets the requirements for the specification language in HW-SW codesign:

- Since dataflow can express both large grain and fine grain parallelism, homogeneous modeling in all phases of the design process is possible.
- Concurrency can be easily expressed in terms of dataflow.
- Hierarchical modeling is supported.
- Communication within dataflow networks is straightforward.
- Synchronization of communicating units is done by waiting for data from another unit.
- Performance evaluation of dataflow models is solved, the theoretical background is Petri nets.
- Scalability of the dataflow notation allows the simple description of massively parallel computing.

A *dataflow network* N is a set of nodes P_N, which execute concurrently and exchange data over point-to-point communication channels C_N. The *dataflow node* represents the functional elements of the system and describes their signal propagation behavior by a simple relation between input and output, eventually depending on the previous state of the node. The use of relations instead of input-output functions allows the modeling of non-deterministic behavior. For instance in case of diagnostics this provides a proper mean to express diagnostic uncertainty. The *channels* of the dataflow 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 the set of firing rules R_p over the input domain and over S_p, 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'.

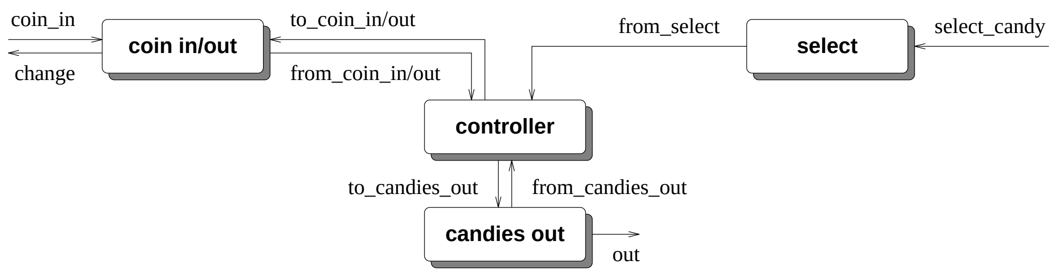
2.2 An example

Due to space limitation the selected example is kept very simple and can not even introduce the full modeling power of the presented approach. The system is an automaton that delivers different candies.

According to the proposed approach, the modeling is uninterpreted and for more accurate description of the complex functional units a multi-valued fault model is used. This fault model has to express uncertainties originating in the neglected data dependencies. According to the black-box modeling approach component faults are identified by the rough, and for the sake of the compactness of the example, simplified classification of the results they deliver:

- `ok` colored token denotes that the component delivered correct computational result.
- `inc` token denotes that the component delivered incorrect result.
- `dead` token is sent, if the component, due to a fatal fault, does not deliver results at all.
- `x` message is used to express uncertainty. `x` is sent if the result, depending on the input and on the implementation of the component, can be either correct or incorrect.

An adder component, that receives fault free inputs is considered to enlighten the meaning of the different



CANDY AUTOMATON:
P_N ={coin_in/out, select, controller, candies_out}
C_N ={coin_in, change, to_coin_in/out, from_coin_in/out, from_select select_candy, to_candies_out, from_candies_out, out}

CANDIES OUT:
I={to_candies_out}
O={from_candies_out, out}
S={ok$_0$, inc, dead}
R={f1 ... f12}

f1=(ok$_0$; to_candies_out=ok; ok$_0$; ok->from_candies_out, ok->out)
f2=(ok$_0$; to_candies_out=inc; ok$_0$; ok->from_candies_out, inc->out)
f3=(ok$_0$; to_candies_out=dead; ok$_0$; ok->from_candies_out, dead->out)
f4=(ok$_0$; to_candies_out=x; ok$_0$; ok->from_candies_out, x->out)
f5=(inc; to_candies_out=ok; inc; ok->from_candies_out, inc->out)
f6=(inc; to_candies_out=inc; inc; ok->from_candies_out, x->out)
f7=(inc; to_candies_out=dead; inc; dead->from_candies_out, dead->out)
f8=(inc; to_candies_out=x; inc; ok->from_candies_out, x->out)
f9=(dead; to_candies_out=ok; dead; dead->from_candies_out, dead->out)
f10=(dead; to_candies_out=inc; dead; dead->from_candies_out, dead->out)
f11=(dead; to_candies_out=dead; dead; dead->from_candies_out, dead->out)
f12=(dead; to_candies_out=x; dead; dead->from_candies_out, dead->out)

Figure 2: Data flow model of the candy automata

tokens. If the adder is fault free, an `ok` token is sent. If a faulty adder adds every time 2 to the result of the addition, an `inc` token is sent. If the 0 bit of the adder is stuck-at-0 an `x` token is sent, since depending on the values of the input the result can be either correct (8+2 is 10 in both the fault free and in the faulty case) or incorrect (8+1 is 9 in fault free and 8 in the faulty case). If the adder is implemented by an independent hardware component and the power of the component is broken a `dead` token will be sent. What can be expressed by the use of `dead` tokens? The answer gives an example for token grouping according to severity: in a responsive real-time application it is very important to have results within a defined time limit. If a component does not deliver any result, it can be considered as a severe fault, while delivering incorrect result is less severe. On the other hand, in a safety critical application such a component can be detected by a watchdog, but an incorrect result can lead to system crash.

The dataflow graph of the system and the formal definition of one of its components are shown in Figure 2. The components are assumed not having built-in fault detection capabilities, and the evaluation of the model is restricted to the case of single internal faults. The system consists of four parts:

coin_in/out sends as first step of its operation the calculated sum of coins inserted by the user to the controller. The change determined by the controller is returned to the user in the second step. Malfunctions of the component are `inc` (wrong recognition of the value of a coin) and `dead` (a coin is stuck).

select sends the identifier of the candy selected by the client to the controller. We assume, that this component can not be `dead`, but it may deliver incorrect results (keyboard fault).

controller issues an order to the candies_out component to deliver a candy according to the client's selection and the sum of coins. After delivering the candy it calculates the change. The controller delivers either correct or incorrect results (fault in the numerical computation).

candies_out delivers the candies according to the command from the controller. The end of the process is reported back to the controller. Errors of the component are `inc` (wrong type of candy is delivered) or `dead` (candy is out of stock).

Inputs of the system are: `coin_in`, `select_candy`, while `change` and `out` are its outputs. The initial state of fault-free components is ok_0. A verbal interpretation of some firing rules of the candies_out node (Figure 2) is:

f1- In a fault-free system this rule describes the candies_out node. As only error free messages are received on its input (`to_candies_out=ok`) it always remains in its error free state `ok`$_0$ and sends error free messages to its outputs (`ok->from_candies_out` and `ok->out`)

f2- Describes the error propagation of the fault-free component in the state `ok`$_0$. An erroneous input (`to_candies_out=inc`) will result in faulty delivering of candies (`out->inc`) while the other output is assumed to remain unaffected by the error (`from_candies_out->ok`). As an external fault does not have a permanent effect on `candies_out`, it remains in its error free state `ok`$_0$.

f5- Due to an assumed internal fault, the component is permanently in the erroneous state `inc`.

Correct input messages (`to_candies_out=ok`) will be resulting in incorrect output `out->inc` value (wrong type of candy is delivered), but a correct signal will be sent to the controller `from_candies_out->ok` (candy is delivered successfully).

3 Test Strategy Design

The basis of effective fault detection and diagnostics is a well planned test strategy. This test strategy describes the execution order of the tests of a given subset of the system's test set. Selection of the subset is done according to some criteria, like test time minimization, maximum fault coverage, safe testing, etc. The test set of the system usually is created automatically from the system description and fault model. It contains the test vectors describing the inputs of the system being necessary to detect component faults at the outputs of the system. Inputs of the system can be either the primary inputs (PI), or special, test inputs (TI). Outputs once again are either primary outputs (PO), or test outputs (TO). Earlier test pattern generation was executed at the logic-gate level. Since in case of complex systems the very large number of logic gates prohibits the use of traditional test generation method, new approaches are necessary. In this section it will be shown that test strategy design can be done concurrently with system design even at a higher level of abstraction by using a dataflow model based automatic test pattern generation (ATPG). ATPG at the logic-gate level is a very well elaborated field of the computer science offering well-proven solutions for practical problems. As our approach uses a generalized form of gate-level ATPG, at first a comparison between logic-gate level description and the proposed dataflow description is given:

- In the gate and module-level stuck-at fault model, faults are modeled at the output of logic gates. Faults of a functional dataflow node are manifested similarly at the outputs in the form of faulty messages.
- The behavior of a dataflow 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 [1].
- Since dataflow components can have internal states as well, the testing of a system requires a predefined initial system state. (In practical dataflow models examined till yet there was no need for the search of a self-initialization sequence.)

Due to this correspondence, methods and solutions of logic-gate level test pattern generation can be generalized for the dataflow model and high-level ATPG algorithms can be generated. As a representative example the D-algorithm [1, 9] is selected, that is well known and widely used for test generation for stuck-at faults in logic circuits.

3.1 The high-level D-algorithm

In order to generate a test for a given fault the problem of test generation is recursively divided into the subproblems of:

- error propagation
- line justification
- implication and checking

Error propagation tries to propagate the error of a line to the POs, line justification is responsible for setting the PIs according to the fault of a given node, and implication and checking aims at the reduction of the problem space. The D-algorithm performs implication and checking after each line justification and error propagation step. Error propagation has priority over line justification. To keep track the still open problems two sets are maintained during the algorithm: the *J-frontier* containing the gates of which inputs have to be justified and the *D-frontier* containing the gates from the inputs of which the error has to be propagated towards the POs.

Due to modification during adaptation, our solution of the subproblems is slightly different from the original one:

- Due to the multi-valued fault model not only 0 and 1 values are used. (E.g. in the candy automaton example `ok, inc, dead, x` values are used.) In correspondence with it instead of value pairs D (1 in the good, 0 in the faulty circuit) and $\overline{\mathrm{D}}$ (0/1), various value pairs are propagated. (In the example `ok/inc, ok/dead, ok/x, inc/ok, inc/dead,`
`inc/x, dead/ok, dead/inc, 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). For this reason nodes and channels are instantiated, i.e. objects with the same instance number belong to the same block.
- Since components may have multiple outputs the J- and D-frontiers contain channels instead of gates.
- Checking has to ensure that the additional constraints imposed by the global testing requirements are fulfilled. E.g. in safe testing the propagation of a `dead` token is prohibited.
- A frame program of the high-level D-algorithm (Figure 3) is necessary because components may have multiple outputs, thus error propagation can be started in more than on direction. The set input mappings (IM), created in Step 4, contains all those input mappings for which a fault pair is activated on one or more of outputs of the component. Test generation is successful if at least for one input mapping the D-algorithm executes successfully, Step 5–8.

The high-level D-algorithm (Figure 4) recursively calls itself and at each call performs the possible implications, checks the decisions made by the previous

```
1:  test generation algorithm(N, err)
2:  begin
3:    set node N  state to err
4:    create the set of input mappings IM
5:    repeat
6:      select the next element of IM
7:      D-alg()
8:    until SUCCESS or IM is empty
9:  end
```

Figure 3: The test generation algorithm

```
1:   D_alg()
2:   begin
3:     if (imply&check()=FAILURE) then  return FAILURE
4:     if (error not at PO) then
5:       if (D-frontier empty) then  return  FAILURE
6:       repeat
7:         select C from D-frontier
8:         if (Propagate(C)=SUCCESS) then  return SUCCESS
9:       until all elements of D-frontier have been tried
10:      return  FAILURE
11:    if (J-frontier empty) then   return  SUCCESS
12:    select C from J-frontier
13:    if (Justify(C)=SUCCESS) then   return  SUCCESS
14:    return  FAILURE
15:  end
```

Figure 4: Procedure D-alg()

call and solves an error propagation or a line justification subproblem (Figure 5).

The Imply&check() procedure selects all the components of which inputs or outputs have been changed since the last call (Step 3). Then it tries to make implications and consistency check for the selected components. If check fails for any of this components no test can be generated. When encountering such a situation this solution will be immediately rejected as if it was a contradiction and backtracking is invoked.

Suppose that channel C is an output channel of component N. The procedure Justify(C) creates the subset of firing rules of N, such that firing rules belonging to F produces the required output on C. Then D-alg() is called to continue test generation. If this fails for all elements of F, channel C can not be justified.

The function of Propagate(C) is similar to that of Justify. In this case C is an input channel of the node, and firing rule pairs from FP have to propagate the fault pair present on C to one or more outputs. If the continued test generation fails for all of the selected firing rule pairs, the fault on C can not be propagated.

3.2 Complexity issues

The algorithms, especially the error propagation, line justification, and implication procedures, are slightly more complex then the original one due to the higher complexity of functional dataflow elements. Moreover due to the multi-valued fault model the decision tree of such an ATPG algorithm is larger, e.g. for a given decision the number of alternatives is higher. This can lead to an increased number of backtracking.

Since the dataflow semantics used for the modeling maintains the compositionality property [7] these complexity issues can be managed by hierarchical test generation. Suppose that the test results of a subnetwork for a given test vector are known (a partial test

```
1:   Imply&check()
2:   begin
3:     create the set N
4:     repeat
5:       select the next element of N
6:       set inputs, outputs and state of the node
           if it can be done uniquely
7:       if (input-output, state or criteria inconsistency exists)
           then  return  FAILURE
8:     until all elements of N have been processed
9:     return  SUCCESS
10:  end
```

```
1:   Justify(C)
2:   begin
3:     create the set F
4:     repeat
5:       select the next element of F and
           set inputs, outputs and component state
6:       if (D-alg()=SUCCESS) then   return  SUCCESS
7:     until all elements of FP have been tried
8:     return FAILURE
9:   end
```

```
1:   Propagate(C)
2:   begin
3:     create the set FP
4:     repeat
5:       select the next element of FP and
           set inputs, outputs and component state
6:       if (D-alg()=SUCCESS) then   return  SUCCESS
7:     until all elements of FP have been tried
8:     return FAILURE
9:   end
```

Figure 5: Procedures for solving subproblems

has been generated). If this is placed into a larger dataflow network, test generation can be started with error propagation from the output of the subnet to the POs and line justification can be started from the inputs of the subnet towards the PIs of the system without going inside of the subnet.

3.3 Test generation for the example

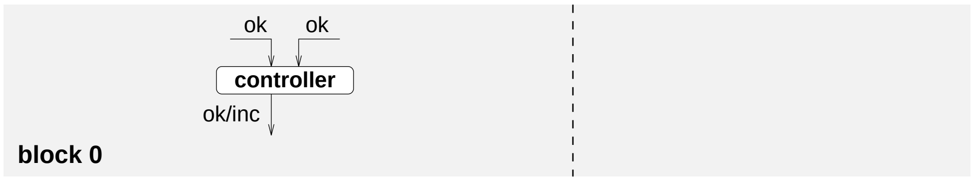
To enlighten the previously presented algorithm, test generation is presented in details (Figure 6) for the `inc` fault of the controller component in the simple example. The test generation procedure is shown in Figure 6 step-by-step.

Vertical partitioning of Figure 6 denotes the different steps of test generation. In horizontal partitioning the leftmost part contains the identifier of the steps. In the middle the unfolded dataflow model of the system is shown (only channel and nodes with defined state are shown) and in the rightmost column states of the nodes and the channels, the J- and the D-frontiers are enlisted. The different steps in the process are:

Step 1: State of the controller is $\texttt{ok}_0/\texttt{inc}$. The input mapping $(\texttt{from\_coin\_in}/\texttt{out}_0 = \texttt{ok}$, $\texttt{from\_select}_0 = \texttt{ok})$ is selected and outputs are set according to it: a value pair appears on the output channel of the controller $(\texttt{ok}/\texttt{inc}- > \texttt{to\_candies\_out}_0)$. The D-algorithm can be started.

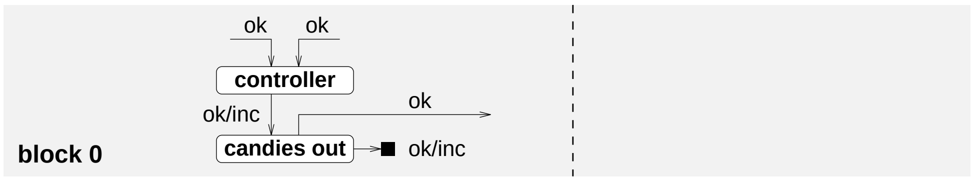
Step 2: Error propagation step: the value pair from $\texttt{to\_candies\_out}_0$ is propagated through the component candies_out. A value pair appears on the PO $\texttt{out}_0$, but propagation can not be stopped, there is still data on channel $\texttt{from\_candies\_out}_0$.

Step 1:



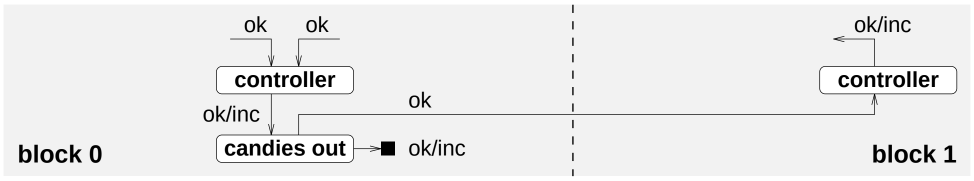
ctrl$_0$=ok$_0$/inc
from_select$_0$=ok
from_coin_in/out$_0$=ok
to_candies_out$_0$=ok/inc
J={from_coin_in/out$_0$, from_select$_0$}
D={to_candies_out$_0$}

Step 2:



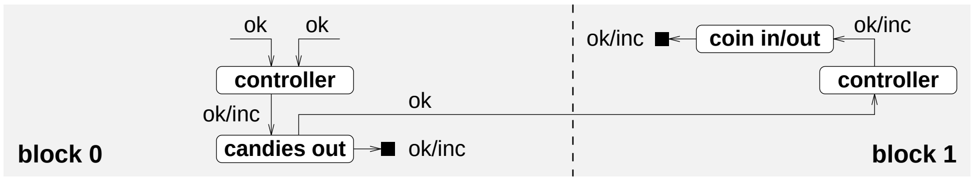
candy$_0$=ok$_0$
out$_0$=ok/inc
from_candies_out$_0$=ok
J={from_coin_in/out$_0$, from_select$_0$}
D={from_candies_out$_0$}

Step 3:



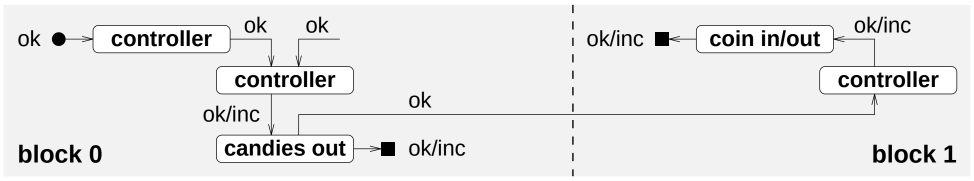
ctrl$_1$=ok$_1$/inc
to_coin_in/out$_1$=ok/inc
J={from_coin_in/out$_0$, from_select$_0$}
D={to_coin_in/out$_1$}

Step 4:



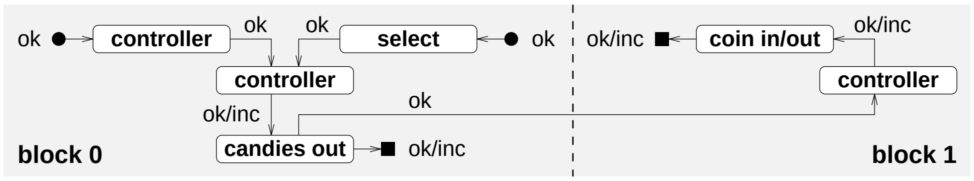
coin$_1$=ok$_1$
change$_1$=ok/inc
J={from_coin_in/out$_0$, from_select$_0$}
D={}

Step 5:



coin$_0$=ok$_0$
coin_in$_0$=ok
J={from_select$_0$}
D={}

Step 6:



select$_0$=ok$_0$
select_candy$_0$=ok
J={}
D={}

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

input		system state				output	
coin_in	select_candy	coin_in/out	select	controller	candies_out	out	change
ok	ok	ok	ok	ok	ok	ok	ok
ok	ok	inc	ok	ok	ok	inc	x
ok	ok	ok	inc	ok	ok	inc	ok
ok	ok	ok	ok	inc	ok	inc	inc
ok	ok	ok	ok	ok	inc	inc	ok
inc	ok	ok	ok	ok	ok	inc	ok
inc	ok	inc	ok	ok	ok	inc	x
inc	ok	ok	inc	ok	ok	inc	ok
inc	ok	ok	ok	inc	ok	x	inc
inc	ok	ok	ok	ok	inc	x	ok
ok	inc	ok	ok	ok	ok	inc	ok
ok	inc	inc	ok	ok	ok	inc	x
ok	inc	ok	inc	ok	ok	inc	ok
ok	inc	ok	ok	inc	ok	x	inc
ok	inc	ok	ok	ok	inc	x	ok

Table 1: Results of fault simulation

Step 3: Error propagation from `from_candies_out`$_0$ through the controller. This will be the second instance of the controller in the test. A value pair is propagated to `from_coin_in/out`$_1$. (`from_coin_in/out`$_1$ denotes that the channel is in the second block of the iterative array model.)

Step 4: Error propagation from `from_coin_in/out`$_1$ trough component coin_in/out. The value pair `ok/inc` reaches the PO `change`$_1$ and the D-frontier is empty. Error propagation is finished, line justification can be started.

Step 5: Line `from_coin_in/out`$_0$ is justified by setting PI `coin_in`$_0$ to `ok`.

Step 6: Justification of `from_select`$_0$ by setting PI `select_candy`$_0$ to `ok`. The J-Frontier is empty, the system is in consistent state (components in the first block are in their `ok`$_0$ state), test generation is finished successfully.

The generated test vector maps `ok` on both `coin_in` and `select_candy`. As a result in the fault free system `ok` appears on both outputs of the system: `out` and `change`. If the components `coin`, `candy`, and `select` are fault free and the controller has the fault `inc`, after applying the test vector `inc` appears on both outputs of the system.

4 Testability Analysis

For testability analysis the integrated diagnostics approach of Sheppard et. al. [11] is used. This approach is based on the conclusion-test and test-test dependency relations. Conclusion is the isolation of a fault, in our case fault of a component. Test is any information source relevant to the diagnostic problem. Dependency relationships among tests and conclusions are described in form of a directed dependency graph. In the dependency graph tests and conclusions are represented by nodes (tests are denoted graphically by circles, conclusions by boxes) and dependencies are directed edges. If a test T_2 depends on T_1 (if T_1 fails T_2 will also fail), then a path exists from T_1 to T_2. Similarly if a test T_3 depends on the conclusion C_1 (if C_1 fails T_3 will also fail), then a path exists from C_1 to T_3. (A test fails, if it does not deliver the intended result). The adjacency matrix of the dependency graph, the so-called dependency matrix, has two parts, one describing test-test dependencies (upper part) and a second one describing conclusion-test dependencies (lower part). If conclusion C_3 depends on test T_1 then the [3,1] element of the lower part is set. Based on the dependency matrix different testability measures can be computed [11]:

- isolation level (ratio of diagnosable faults)
- nondetection (fault coverage)
- test leverage (robustness of the test set)
- overtesting (ratio of uniquely diagnosable faults to number of tests is relatively high)
- undertesting (ratio of uniquely diagnosable faults to number of tests is relatively low)
- test uniqueness (a test can detect/diagnose only one fault)
- test redundancy (multiple tests can detect and/or diagnose the same fault)
- false alarms (the cumulative effects of multiple faults produces an identical syndrome as a different fault)

This method of testability analysis is used in our modeling approach since all dependency relationships can be extracted from the dataflow model by means of fault simulation. For this purpose in [6] a parallel fault simulation is proposed. In the previously presented example 5 possible conclusions can be considered: C_1 is coin_in/out=`inc`, C_2 is select=`inc`, C_3 is controller=`inc`, C_4 is candies_out=`inc`, and finally the

	T_{1a}	T_{1b}	T_{2a}	T_{2b}	T_{3a}	T_{3b}
T_{1a}	F		F		F	
T_{1b}	F	F	f	f	f	f
T_{2a}			F		f	
T_{2b}	F	f	f	F	f	f
T_{3a}			f		F	
T_{3b}	F	f	F	f	F	F
C_1	F	f	F	f	F	f
C_2	F		F		F	
C_3	F	F	f	F	f	F
C_4	F		f		f	
C_0			F		F	

Table 2: Dependency matrix for the example

conclusion no fault is denoted by C_0. (For simplicity `dead` faults are omitted.) In the analysis of the example 3 tests are considered: T_1 (the one generated in the previous section, a test for controller=`inc`), T_2 (for the fault controller=`inc`), and T_3 (for candies_out=inc). They map the inputs `coin_in` to `ok, inc, ok` and `select_candy` to `ok, ok, inc`. The outputs `out` and `change` are referred to T_{1a}, T_{2a}, T_{3a} and as T_{1b}, T_{2b}, T_{3b}.

Simulation results are shown in Table 1. The corresponding dependency graph is in Figure 7 and the dependency matrix is in Table 2 respectively. In the dependency matrix F denotes "strong" dependency, i.e. when a conclusion fails the test will also fail, while f denotes "weak" dependency, i.e. when a conclusion fails failure of the test will depend on the actual value in the test vector, corresponds to an `x` output value. In the graph solid lines present "strong" dependency, and dashed lines present "weak" dependency.

How is the dependency graph extracted from the simulation results? For example, each time the inputs are `ok, ok` and coin_in/out fails, T_{1a} will also fail as denoted by the `inc` result on `out`. In the dependency graph it is represented by the solid line from C_1 to T_{1a}, while in the dependency matrix by the element [7,1]=F. In the same case test result T_{1b} is data dependent (output `change` is `x`), thus the dependency is represented by the dashed line from C_1 to T_{1b} and by the element [7,2]=f.

In the analysis the term *pessimistic* case is used, when only "strong" dependencies are considered. The term *optimistic* is used when also "weak" dependencies are taken into account. The pessimistic case is the worst case of analysis from the point of view of dependability measures. On the other hand in optimistic case the upper bound of dependability measures can be estimated.

Since the example is simple not all of the previously mentioned testability measures are really meaningful. The most important results of the analysis are:

- It can be seen from the dependency matrix, that none of the faults produce an identical syndrome as the fault-free case. Thus fault coverage is 100%, all faults can be detected in both the pessimistic and in the optimistic case. (Syndrome is the result of tests when the given fault occurs.)
- Isolation level equals the ratio of uniquely isolatable groups to all fault conclusion and denotes the ratio of diagnosable faults. In the pessimistic case failure signatures can be divided into three groups (group1: C_1, C_3; group2: C_2; group3: C_4) according to the observable fault effects enlisted in the dependency matrix. Thus the value of isolation level is $3/4 = 0.75$. In the optimistic case the number of groups is 4, thus isolation level is 1.0 indicating, that all of the faults can be diagnosed although tests have been generated only for the controller and for candies_out.
- T_1 and T_2 are redundant, since in all cases they result in identical syndrome. Thus one of them can be left out during the system test.

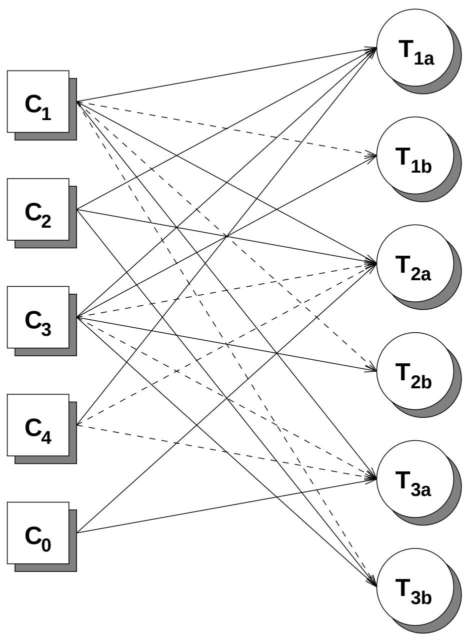


Figure 7: Dependency graph for the example

5 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 as an integral part of the design process, since in the proposed dataflow model both the functional and error 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 [8] design environment, developed at the University of California at Berkeley, was used. As part of the future work we want to identify and examine the constraints imposed by the various testing criteria on HW-SW separation.

Acknowledgments

The authors want to express their gratitude to Prof. M. Dal Cin and Prof. L. Simoncini, whom hosted them for a period at the University of Erlangen, Germany, and at the University of Pisa, Italy respectively, where this work was started. Also their useful comments to this work are gratefully acknowledged.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] A. Kalavade and E. A. Lee. A Hardware-Software Codesign Methodology ofr DSP Applications. *IEEE Design & Test*, pages 16–28, September 1993.

[9] J. P. Roth, W. G. Bouricius, and P. R. Schneider. Programmed Algorithms to Compute Test to Detect and Distinguish Between Failures in Logic Circuits. *IEEE Transactions on Electronic Computers*, EC-16(10):567–579, October 1967.

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

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