PhD Thesis:

A Framework for Early Testability Assessment

by

György Csertán
Department of Measurement and Information Systems
Technical University of Budapest
H-1521 Budapest, Műegyetem rkp. 9, HUNGARY

June 2, 2000

Advisors:

Associate Prof. András Pataricza
Department of Measurement and Information Systems
Technical University of Budapest

Prof. Endre Selényi
Department of Measurement and Information Systems
Technical University of Budapest

To my parents for their love and support.

Preface

The five years, during which this work was carried out, was a very fruitful period of my life. I want to express my gratitude to all the people who helped me in some way through this period.

András Pataricza and Endre Selényi, my advisors, introduced me to the world of fault-tolerant computing. Special thanks to András Pataricza who undertook the supervision of my Ph.D. studies. The many hours we spent with discussion were very helpful to overcome my problems and to create this work. He was the person whose contacts lead me to Pisa and Erlangen to the famous schools of European fault-tolerant computing. In many cases he arranged the financial background of my visits and travels to conferences, making it possible to present my work on international forums.

The Department of Measurement and Instrument Engineering, lead by Gábor Péceli, was the main place of my Ph.D. studies. The atmosphere of the department and the helpfulness of the staff made it a pleasant working place.

In 1993 I spent a three-month scholarship at the University of Pisa, where the use of dataflow models was initiated. There we used the dataflow models for temporal analysis of control systems, but later this model was the starting point of my modeling approach. For their inspiration I want to thank to Luca Simoncini, Andrea Bondavalli, and Cinzia Bernardeschi.

In 1994 and 1995 I had the opportunity to spend several month first as a Ph.D. student then as a guest researcher at the University of Erlangen, where many of the implementation work was done and the MEMSY model was constructed and elaborated. I am thankful to Mario Dal Cin the head of IMMD3 for hosting me at his department for this period and to Wolfgang Hohl and Christine Cetin for helping me through the German bureaucracy. I am grateful to Ralph Thebis, Stefan Dalibor, Susann Allmaier, and Richard Grillenbeck for their friendship and for not leaving me feel alone and lost in a foreign country.

Finally I want to thank to my family and friends for being always supportive despite the fact that many times they were the sufferers of this five years work.

Contents

1 Introduction **1**

1.1 Problem Statement . . . 2
1.2 Previous Work . . . 2
1.2.1 CAD tools for system design . . . 3
1.2.2 CAD frameworks for dependable design . . . 5
1.3 Contribution of the Work . . . 10
1.4 Outline of the Thesis . . . 13

2 Modeling Approach **15**

2.1 Dataflow Networks . . . 17
2.1.1 Informal Presentation of the Model . . . 17
2.1.2 Formalism of Dataflow Networks . . . 18
2.2 Model Extension . . . 21
2.2.1 Fault Modeling . . . 22
2.2.2 Uncertainty Modeling . . . 23
2.2.3 Aspects of Fault-Tolerance in the Model . . . 25
2.3 Evaluation of Dataflow Models . . . 27
2.4 An Example . . . 29
2.4.1 Basic Operation . . . 30
2.4.2 Extended Operation . . . 31
2.5 Contribution . . . 33

3 Model Refinement **35**

3.1 Approaches of Model Refinement . . . 35
3.2 Dataflow Refinement . . . 36
3.2.1 Domain Refinement . . . 37
3.2.2 Structure Refinement . . . 38
3.3 Consistency Checking After Refinement . . . 40
3.3.1 From DFN to NDFST . . . 41
3.3.2 Bisimulation of NDFSTs . . . 43
3.3.3 Checking Algorithm . . . 45
3.4 Application of Refinement in Modeling . . . 46

3.4.1 Model Changes and Refinement . . . 46
3.4.2 Multi-Level Modeling and Refinement . . . 47
3.4.3 Uncertainty Modeling and Refinement . . . 49
3.5 Refinement of the Example . . . 50
3.5.1 Refinement of Candies_out . . . 50
3.5.2 Refinement checking . . . 51
3.6 Contribution . . . 54

4 Test Generation **57**
4.1 Test Generation Approaches . . . 57
4.1.1 Gate-level Test Generation . . . 57
4.1.2 High-level Test Generation . . . 58
4.2 Dataflow Test Generation . . . 59
4.3 PODEM for Dataflow Models . . . 61
4.3.1 PODEM for Sequential Circuits . . . 61
4.3.2 Adaptation of PODEM . . . 62
4.3.3 Description of the Algorithm . . . 64
4.3.4 Possible Simplification . . . 67
4.4 Complexity Issues . . . 67
4.5 Test Generation for the Example . . . 68
4.6 Contribution . . . 70

5 Testability Analysis **71**
5.1 On Integrated Diagnostics . . . 72
5.2 Integrated Diagnostics and the Dataflow Approach . . . 74
5.2.1 Extracting the Input Model of Integrated Diagnostics . . . 74
5.2.2 Dealing with Uncertainty . . . 74
5.3 Testability Measures after Refinement . . . 75
5.4 Testability Analysis of the Example . . . 78
5.5 Contribution . . . 82

6 An Application Study **83**
6.1 MEMSY . . . 83
6.1.1 Modeling . . . 84
6.1.2 Model Statistics . . . 87
6.2 Model of the CPU . . . 87
6.2.1 Basic Operation of the CPU . . . 89
6.2.2 Extended Operation of the CPU . . . 92
6.2.3 Uncertainty Effects . . . 93
6.3 Evaluation . . . 93
6.3.1 Test Generation . . . 93
6.3.2 Testability Analysis . . . 95

6.3.3 Time Complexity of the Evaluation . . . 99
6.4 Contribution . . . 100

7 Conclusion and Future Work 101

A Abbreviations and Symbols A-1
A.1 Abbreviations . . . A-1
A.2 Symbols . . . A-3
A.2.1 Dataflow Notation . . . A-3
A.2.2 Nondeterministic Finite-State Transducer Notation . . . A-4
A.2.3 Set and String Notation . . . A-4
A.2.4 Testability Notation . . . A-4

B Definitions and Derivations B-1
B.1 Definitions . . . B-1
B.2 Derivations . . . B-2

C Formal Description of the Candy Automaton C-1
C.1 candy automaton . . . C-1
C.2 coin_in/out . . . C-2
C.3 select . . . C-3
C.4 controller . . . C-3
C.5 candies_out . . . C-4
C.6 candies_out (refined) . . . C-5
C.7 hwl . . . C-5
C.8 mechanics . . . C-6

D Formal Description of MEMSY D-1
D.1 MEMSY-I . . . D-1
D.2 CPU . . . D-2

E Publications E-1

List of Figures

1.1 Design Flow of HW-SW Codesign 11
1.2 Model Evaluation in HW-SW Codesign 12

2.1 An Example Dataflow Network 19
2.2 DFG of the Candy Automaton 30

3.1 Example of Structure Refinement 39
3.2 Algorithm of Refinement Checking 45
3.3 DFG of the Refined Candy Automaton 50
3.4 Reachability Graph of $NDFST_1$ 54
3.5 Reachability Graph of $NDFST_2$ 55

4.1 Iterative array model of a dataflow network 62
4.2 The PODEM algorithm 64
4.3 The Objective() Procedure 66
4.4 The Backtrace() Procedure 66
4.5 PODEM for the Candy Automaton 69

5.1 Testability Analysis Based on System Structure 71
5.2 Testability Analysis Based on the Test Set of the System 72
5.3 Dependency Graph for the Example 81

6.1 A MEMSY node with 1 CPU and 8 CMMUs 86
6.2 Dataflow Model of the CPU 89
6.3 FSM Model of the CPU (Basic Operation) 90
6.4 Test for the Data Fault of Terminal #1 96
6.5 Results of Testability Analysis 97
6.6 Results of Test Set Optimization 98
6.7 Execution Time of a Single Run 99

List of Tables

1.1 Classification of CAD tools in computer design 3
1.2 Tools for high-level software design 4
1.3 Tools for high-level hardware design 5
1.4 Tools for HW-SW codesign 5
1.5 CAD Frameworks 6

2.1 Advantages of the dataflow modeling paradigm 16
2.2 Set of States of the Components 32

3.1 Set of States of the Components After Refinement 51

4.1 Similarities between gate-level and dataflow ATPG 60

5.1 Test Vectors for the Candy Automaton 79
5.2 Results of Fault Simulation 80
5.3 Dependency Matrix for the Example 80

6.1 PMC-like Error Propagation Function of the Components 85
6.2 Statistics of MEMSY-I 87
6.3 Statistics of the Components 88
6.4 States of the CPU (Basic Operation) 89
6.5 Components for Which Test Generation Was Successful 94
6.6 Results of Testability Analysis 97
6.7 Results of Test Set Optimization 98
6.8 Execution Time of a Single Run 99

Chapter 1

Introduction

The availability of low-cost, but highly complex off-the-shelf programmable components (PLDs), microprocessors, and ASIC technologies allows to implement very complex applications in digital systems even for small enterprises, and not only for the market leaders in state-of-the-art technologies, like some five years ago. Since many of the small enterprises can not dominate the implementation technologies of these complex applications, recent efforts aim at the reduction of cost and time of the design process by developing automated, integrated environments for system engineering.

The common characteristic of these automated design methods is, that they rely on a mathematical model of the system, which is kept up-to-date during the design; therefore, automatic evaluation of system properties is possible in each phase of the design. Further feature 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, that leads to a significant reduction in design time, for a price of a tolerable overhead. Last but not least, the use of automated design technologies radically improves the product's design quality as well as the quality of service provided by the system.

As even more applications, like process control, transport and automation systems, include safety related aspects, dependability becomes an important design issue. Dependability is the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers. The main insufficiency of design automation systems nowadays originates in the lack of an integrated support for the follow-up phases of dependability analysis: reliability analysis, safety analysis, diagnostic design.

In order to avoid costly re-design cycles, dependability analysis has to be pushed into early phases of system design as well. As part of dependability analysis, diagnostic design provides the system's diagnostic plan, the aim of which is to locate the faults within the system by means of executing the test set of the system. By localizing the faults, that can then be repaired by the maintenance personal, a properly elaborated diagnostics can largely increase the availability of the system. Integrated diagnostics aims to maximize the effectiveness of diagnostics by integrating the individual tasks of

testability, testing, maintenance aiding. Therefore, integrated diagnostics effectively can be used in system design environments that aim early diagnostic design.

1.1 Problem Statement

The aim of this work is to overcome the above mentioned problems by providing a framework, that can be used even in early phases of the design. It should be organically integrated into the design flow, and has to give support for the solution of modeling, model checking, fault simulation, test generation, and testability analysis related problems that arise during the design of fault-tolerant computing systems. The main requirements for such an approach can be summarized as:

- There is a need for a high-level, hierarchical, model-based approach in diagnostic design which results from the design process itself: current VLSI approaches specify the design in terms of building blocks and libraries of modules and standard-cells are used.
- Modeling of uncertainty has to be solved in the approach, since in early phases of a design process, modeling is done at a high-level of abstraction. Here the lack of information about lower level implementation details leads to significant under-definition of the components.
- Due to refinement at the lower levels of abstraction, the models become more and more precise and uncertainty decreases. Mechanisms are required to control information addition in order to keep and check the consistency of the model.
- The modeling approach has to support traditional, functional evaluations, e.g. validation, performance analysis, temporal analysis.
- Diagnostic design has to be solved within the approach by providing test generation, testability analysis, and test set optimization.
- The efficiency of the evaluation tools is not of primary concern at the suggested level of modeling, as the models will be of a moderate size because of the high-level of abstraction and aimed at is only a prototype of the testability analysis framework.

1.2 Previous Work

Since our aim is to integrate system design and dependable design into a common framework, the presentation of previous work is divided into two parts. The first part gives an overview of the current design environments with special emphasis on system design. The second part gives a summary of well known frameworks for dependable design and dependability analysis.

properties	low-level logic synthesis	high-level architecture design		
		HW	SW	HW-SW
commercially available	+	+	+	-
modeling background	non-formal	semi-formal	semi-formal	non-formal
maturity level	mature	usable	usable	prototypes
standardized language	EDIF, VHDL	-	UML, SDL	-

Table 1.1: Classification of CAD tools in computer design

1.2.1 CAD tools for system design

A large number of CAD tools have been implemented during the last ten years in order to support the design of computing systems. A rough classification of these CAD tools can be found in Table 1.1.

Since CAD tools have been used for at least 5 years for low-level logic synthesis, they are far beyond their childhood and mature commercially available examples exist. One of the most widely used tools is AutoLogic [`http://www.mentor.com`]. The common characteristic of these tools is that they rely on non-formal models. They accept the circuit description in some standardized description language, e.g. EDIF [`http://www.edif.org`], VHDL [IEE93] and generate ASIC layouts, FPGA or PLD mappings. They can cope with very large designs, size usually above hundreds of thousands of logic-gate equivalents, but they do not provide a way to design an application of such size or even not to evaluate parameters of the design. Therefore, they can not be used for high-level design.

Tools for high-level architecture design fall into three categories: software (SW) design, hardware (HW) design, and HW-SW codesign. The various approaches and tools do not have such a long history as logic synthesis CAD programs. Tools for SW and HW architecture design are built in the last 2-3 years, they are commercially available, but they are not fully mature yet. On the other hand HW-SW codesign is a very young discipline where merely prototypes of academic institutions exist. One exception is the tool Ptolemy, which was bought by Hewlett Packard in Summer 1997, and its alpha version is sold currently.

High-level SW design aims at automatic program generation for a given programming language from a high-level description. The description language is different for the various tools and modeling approach, but recently more and more tool providers start to use the Unified Modeling Language (UML), that has been standardized in January 1997 by IEEE and IBM. (For the description of the 1.1 Version from July 1997 please refer to [`http://www.rational.com`].) The target language largely depends on the task the tools are used for. For general applications usually C++ (or other object oriented language) is used, while for embedded control applications C or assembly is preferred. The semi-formal nature of these tools originates in the fact, that rely on a formal model,

properties	Rational Rose	ObjectGEODE	Telelogic Tau
input language	UML	OMT, SDL	SDL, TTCN
target language	C++	C, C++	C++
provider	Rational Corp.	Verilog	Telelogic
http	www.rational.com	www.verilog.fr	www.telelogic.se
evaluations	model checking code gen.	model checking test cases simulation code gen.	model checking test cases simulation code gen.

Table 1.2: Tools for high-level software design

but in the evaluation phase of the design no formal methods are used, most commonly only some kind of simulation is provided. A few tools for SW design are shown in Table 1.2. They, and the others not mentioned here can only deal with design of SW, no emphasis is put on the diversity of evaluation possibilities and dependable design is fully neglected.

High-level HW design aims at automatic generation of the hardware in a given hardware description language from a more abstract, higher than logic-gate level (usually architecture level) description. Modeling languages range from state charts to activity charts, synchron- or asynchron dataflow notation, etc. The target language of the tools are standardized low-level hardware description language, usually VHDL or VerilogHDL. The semi-formal property of these tools is due to the missing formal algorithms during model evaluation and model refinement. Only a few of the tools provides formal validation of the design, most of them supports only model checking. A few tools for HW design are shown in Table 1.3. A common characteristic of all these tools is that they do not support the various evaluations necessary during the design, they provide only a controlled quality of the implementation process. An exception is NP-Tools, within which formal validation of the model is possible but support for design is rather poor (e.g. code generation is very limited). In general, dependable design is with these tools is impossible.

Recently a lot of effort is put into the combination of high-level HW and SW design approaches, in order to aim at an even higher level of system design. The approach is called HW-SW codesign. It aims at the joint specification, design, and synthesis of mixed hardware-software systems and as such it starts from a high-level formal description of the system and generates, usually, C or C++ code for the SW modules of the system and VHDL description of the HW modules. HW-SW codesign environments date back to 1995 when their first example Ptolemy came out. They are still in development phase, only prototypes of academic institutions exists, that are already used by well known industrial companies to evaluate case studies. The three mostly known tools

properties	Statemate MAGNUM	NP-Tools
input language	State Charts	prop. logic
target language	Verilog, VHDL	Verilog
provider	I-logix	Logikkonsult
http	www.ilogix.com	www.lk.se
evaluations	model checking simulation code gen.	simulation validation code gen.

Table 1.3: Tools for high-level hardware design

properties	Ptolemy	Cosmos	Polis
input language	DF	C, VHDL	CFSM
target language	C, m56000, VHDL	C, VHDL	C, VHDL
provider	Berkeley	Imag	Berkeley
http	ptolemy.eecs.berkeley.edu	www.imag.fr	www-cad.eecs.berkeley.edu
evaluations	simulation code gen.	simulation code gen.	simulation code gen.

Table 1.4: Tools for HW-SW codesign

are presented in Table 1.4. Many of them can not boast about a formal mathematical background that makes the use of formal methods impossible. Developers of dependable applications can hardly exploit the benefits of these tools, since they lack a support for fault modeling.

1.2.2 CAD frameworks for dependable design

A number of CAD frameworks have been implemented during the last ten years in order to support the design of dependable computing systems. Many of the tools satisfies one or more of our previously stated criteria, but no one satisfies them all. In the following a short overview is given of the most important dependability evaluation methods and tools (Table 1.5 shows a comparison of them). They are stable and mature representatives of a class of tools for the given application areas. The three previously shown HW-SW codesign tools are also given in the table in order to be able to compare their properties to the necessary criteria, since our aim is to embed dependability analysis into HW-SW codesign.

Ptolemy[ELWW95] (Berkeley, USA): Ptolemy is a flexible foundation upon which to build prototyping environments. Several such environments have been built, including dataflow-oriented graphical programming for signal processing, discrete-

properties	Ptolemy	Cosmos	Polis	Depend	PANDA
description	(codesign)	codesign	codesign	reliability	reliability
modeling language	DF	VHDL, C	CFSM	PN, MC	PN
hierarchy	+	+	+	-	-
refinement	-	-	(+)	-	-
uncertainty	-	-	-	-	-
general evaluation	+	-	+	-	-
dep. evaluation	-	-	-	+	+
ATPG	-	-	-	-	-
testability analysis	-	-	-	-	-

properties	UltraSAN	Surf-2	BudaTEST	Turbo Tester	Int.Diag.
description	reliability	reliability	ATPG	ATPG	testability
modeling language	SAN	MC, GSPN	VHDL	AG	TCDG
hierarchy	+	-	-	+	-
refinement	-	-	-	-	-
uncertainty	-	+	-	-	+
general evaluation	+	-	-	-	-
dep. evaluation	+	+	-	-	-
ATPG	-	-	+	+	-
testability analysis	-	-	-	(+)	+

Table 1.5: CAD Frameworks

event modeling of communication networks, register-transfer-level circuit design, synthesis environment for embedded software, and design assistants for hardware-software codesign. The Ptolemy system is fundamentally extensible. Users can create new component models, new design process managers, and even entirely new programming environments.

Unfortunately codesign support is not fully matured yet, automated design feedback and refinement are missing. The synchronous dataflow models are best suitable for DSP applications but their usage is very restricted in the field of general digital computing systems. From the palette of built-in evaluation support dependability evaluation is missing.

COSMOS (Imag, France): COSMOS is a codesign methodology and tools aimed at the design and synthesis of complex mixed hardware-software systems. The main steps needed in order to transform a system-level specification into a mixed hardware-software one are:

- system level design and synthesis

- communication synthesis
- architecture mapping

COSMOS is still in its childhood: from the mixed VHDL (for the HW components) and C (for SW components) model only code generation is supported, no other evaluation is possible, but the concept of COSMOS is promising.

POLIS[BCJ+97] (Berkeley, USA): The POLIS system is centered around a single Finite State Machine-like representation. A Codesign Finite State Machine (CFSM), like a classical Finite State Machine, transforms a set of inputs into a set of outputs with only a finite amount of internal state. The difference between the two models is that the synchronous communication model of classical concurrent FSMs is replaced in the CFSM model by a finite, non-zero, unbounded reaction time. This model of computation can also be described as Globally Asynchronous, Locally Synchronous. CFSMs are also synthesizable and verifiable models, because many existing theories and tools for the FSM model can be easily adapted for CFSM. The model allows the following evaluations:

- high level language translation
- formal verification
- system co-simulation (uses Ptolemy as a simulation engine)
- design partitioning
- hardware synthesis
- interfacing implementation

POLIS is a typical codesign environment, that supports many evaluation methods that are necessary during digital system design. In POLIS refinement is not supported, although FSMs would allow it, and the CFSM model is originated from FSM. Description of non-determinism is not supported even for the earliest models, that makes the evaluation of high-level models questionable.

Depend (NASA, USA): DEPEND is a simulation-based environment especially geared for the design and evaluation of mission/life critical systems. Using DEPEND, a designer can analyze an entire system under various realistic stress conditions to determine the types of faults to which it is especially vulnerable. The designer can also study the dynamic interactions between the components in the system and identify the major impediments to dependability. As a modeling language DEPEND uses Petri Nets and stochastic processes (Markov chains).

- availability
- mean time between failures, mean time between repairs
- coverage

- fault isolation times, repair times
- error latencies

DEPEND focuses only on system-level models, and its field of evaluation is restricted to classical reliability measures. Diagnostic design and testability analysis is not supported at all.

PANDA (University of Erlangen, Germany) PANDA is a software tool for the analysis of timed Petri nets. It is focused on the area of modeling of fault-tolerance strategies for distributed systems with the object of improving their efficiency. In PANDA both transient and steady-state analysis are available. The main capabilities of PANDA are:

- Inclusion of optimization methods. Since modeling tools such as Petri nets are often used at the design phase, the ability to automatically perform optimization according to selected target functions and boundary conditions is an attractive feature of the software package.
- Parallel computing. The mathematical analysis of Petri-nets frequently requires the solution of extremely large systems of equations. The ability to use modern parallel architectures gives access to the substantially larger memories and computing performance of these machines.
- Fault models. PANDA includes additional fault-modeling tools for designing success diagrams and fault trees and converting them automatically into Petri nets.

PANDA suffers all the restrictions of the usual Petri net based dependability evaluation tools: there is no support for hierarchical modeling, testability analysis, test generation and uncertainty modeling is restricted.

UltraSAN[SIQW95] (UIUC, USA): UltraSAN is a software package for model-based evaluation of systems represented as stochastic activity networks (SANs). The graphical user interface allows for easy specification of the desired models and facilitates the generation of graphs and tables for the obtained results.

The model specification process in UltraSAN is carried out in a hierarchical fashion. On top of the composed model, reward structures can be used to define performance, dependability, and performability measures. To solve the specified model, UltraSAN provides analytic solvers as well as discrete-event simulators.

Although UltraSAN supports hierarchical modeling, the problem of model refinement is not solved. The palette of general evaluations is restricted to performance evaluation, while diagnostic design and testability analysis are missing from dependability evaluation.

Surf-2[Bo93] (LAAS-CNRS, France): Surf-2 is a software tool-box for evaluation system dependability. System behavior may be modeled by either Markov chains or Generalized Stochastic Petri Nets (GSPN). In the latter case, structural verifications are carried out before generating the reachability graph. The derivation of a continuous-time Markov chain from a GSPN is performed from the markings that make the timed transitions firable. The measures of dependability are obtained by processing the Markov chain. The tool provides for the evaluation of different measures of dependability including pointwise measures, asymptotic measures, mean sojourn times and, by superposing a reward structure on the behavior model, reward measures such as expected performance and cost. The administration of the models provides a support for processing a model up to the computation of the numerical results. It allows the management of model libraries based on the notions of an assigned model (assignment of numerical or symbolical values to the parameters of the model) and a model folder (the specification of computation environment and common processing of several models).

Surf-2 is a classical GSPN based modeling tool, that supports dependability evaluation, limited system verification (based on reachability), and performance evaluation. Unfortunately is lacks of hierarchical description in modeling, test generation, testability analysis, support for uncertainty modeling.

BudaTEST[SPTP96] (TUB, Hungary): BudaTEST is a software package for functional-level automatic test pattern generation (ATPG). The system is described by an abstract VHDL model, usually the output of a high level synthesis tool. The input architecture is converted into a constraint network, to which additional constraints are added to represent the test generation problem. Thus ATPG is solved as a constraint satisfaction problem (CSP) by means of a systematic search in the state space of the variables. General constraint solving methods (prior filtering etc.) are employed, but the search is most accelerated by a special heuristic control, characteristic to the TPG problem, which considers a number of cost functions, mainly those related to data and fault propagation costs. A special coloring algorithm is applied in order to explore propagation paths before type-interpreted decisions are made.

Since BudaTEST is a specialized ATPG tool, it concentrates only on test pattern generation, and does not deal with testability analysis or other aspects of dependable design.

Turbo Tester (TUT, Estonia): Turbo Tester is a PC-based set of tools for digital design and test. All of the test tools in TT are implementing algorithms based on the original alternative graph (AG) theory. Instead of directly using the gate-level description for solving test generation and simulation tasks, a higher structural path level has been chosen. Turbo Tester could be used in a number of laboratory

works, as:

- test pattern generation for digital circuits
- test quality and fault coverage analysis
- dynamic test analysis
- design for testability
- built-in self-test analysis
- fault diagnosis in digital circuits

Although Turbo Tester uses a hierarchical model, the refinement rules between different levels of hierarchy are not defined. The set of tools gives a good coverage of testability related problems, but general as well as dependability related evaluations are not possible. The main drawback of the method and tools is, that they do not support uncertainty modeling, even not at the highest level of the hierarchy.

Integrated diagnostics (Arinc, USA): Integrated diagnostics, especially the method developed at Arinc, are a very good means for diagnostic design of digital computing systems. The approach is based on the flow of information among the components of the system. It is centered around the so called dependency relation, that describes test and conclusion dependency, i.e. the circumstances that are necessary the test to fail. Various measures can be extracted from the test conclusion dependency graph (TCDG):

- testability measures
- optimized test set
- test scheduling
- coverage of uncertain tests

The method is specialized to diagnostic design. No approach is given for the production of dependency graph, nor for test generation. Neither dependability nor general evaluation is possible. The model supports uncertainty computation, but lacks hierarchy and refinement.

Although the above tools together cover the whole application field, that was mentioned in the problem statement, none of them provides a sufficiently broad coverage. Therefore, there is still a strong need for a framework that issues the many inter-related aspects of dependable design.

1.3 Contribution of the Work

This work focuses on the field of design of digital computing and control systems, where dependability of the computer is of primary concern. Emphasis is put on the first, early

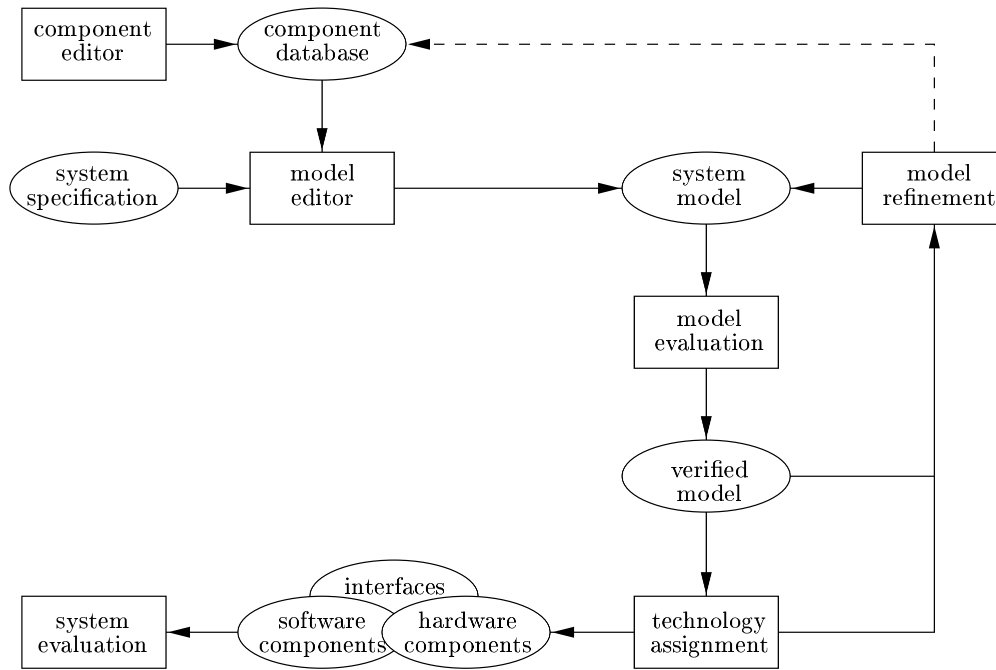


Figure 1.1: Design Flow of HW-SW Codesign

step of system design that consists of the iterative process of model construction, model evaluation and model refinement. Since the addressed systems are a highly complex mixture of hardware and software components, HW-SW codesign is selected to be the design approach into which diagnostic design is integrated.

HW-SW codesign is defined as the joint specification, design, and synthesis of mixed hardware-software systems [BBC+93]. HW-SW codesign is one of the most promising new design methodologies for embedded systems consisting of mixed hardware and software components. Its main advantage lies in the support for design automation and design verification through the whole design cycle. Its typical design flow is depicted in Figure 1.1.

At the beginning of HW-SW codesign a mathematical model of the system is constructed from the system specification and a component database by using a model editor. Once the system model is defined an evaluation-refinement loop is executed as long as a sufficiently detailed, verified model of the system is reached. The aim of this refinement loop is to proceed in many steps from a system-level model down to a low-level where technology assignment can be done. The "in-loop" evaluation has to make sure, that the current model meets the requirements stated in the specification. In Figure 1.2 the evaluation step is shown in more detail from the point of view of our work[1]. Once the system model is verified at the lowest-level, technology assignment is done, that replaces the hardware-software separation step of earlier used design methods. After technology assignment the software- and hardware components and the interfaces of the system are

[1]Shaded areas identify the topics to which this work contributes.

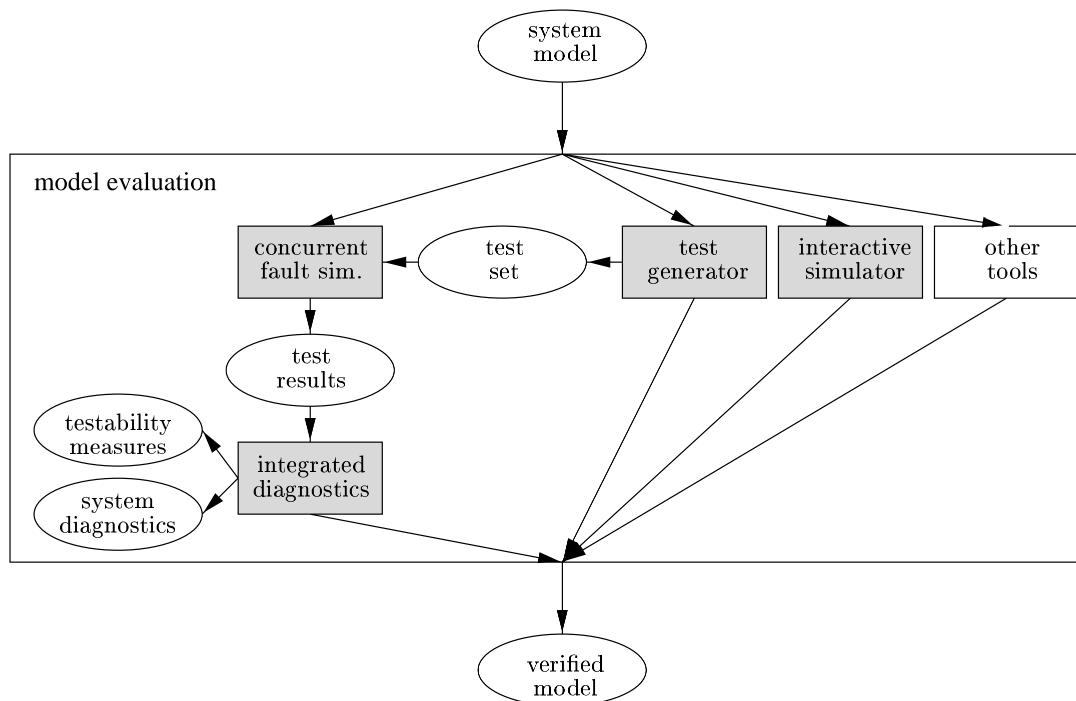


Figure 1.2: Model Evaluation in HW-SW Codesign

defined. In this final phase system evaluation and automated synthesis is started.

In our version of an integrated design environment, in the first step of system design the system is modeled by extended non-deterministic dataflow networks. At this level of abstraction in structural design only the flow of data is modeled in the form of token flow without any description of the data transformation performed by the components. This is called uninterpreted modeling. At this level, primarily performance analysis and optimization are aimed at and test generation and testability analysis can be started.

Through stepwise refinement more and more structural and functional details are incorporated into the initial model. As the system's structure and the data processing functions of the components are increasingly exactly defined the inherent uncertainty of the model decreases, leading to more exact analysis results. This more detailed system is represented by interpreted modeling, where the token flow of uninterpreted modeling are replaced by data flow, and data transformations are also considered. Test generation and testability analysis are done in this phase. Test generation at this level delivers the system-level diagnostic test set of the system.

Finally, after technology assignment and automated HW and SW synthesis, a back-annotation step has to feed back information into the functional system model in order to describe the final design. The back-annotated information contains the assignment of functional units to HW and SW components. As a result, multiple behavioral faults, i.e. faults originating in the same single physical fault due to hardware reuse but leading to failure of two different functional units, can be described by the model. Test generation

and testability analysis are still possible, and the result of test generation is the low-level tests of the components, e.g. self-test of system components.

Our fault modeling approach is based on the idea of modeling the fault effects and their propagation similarly to the flow of data in the functional model. Tokens representing the data can be colored either as correct or as faulty. A set of error propagation paths can be estimated by tracing their flow from the fault site to the outputs of the network. Diagnostic uncertainty is introduced in order to express conditional error propagation. This way the simpler simulation and test generation algorithms deliver a superset of propagated fault effects in the system, but in the model all potential consequences of a fault are incorporated.

A more detailed fault model and a more precise description of the reactions of functional units to erroneous input values can be defined by using multi-valued logic. The use of other guiding attributes in a user-defined coloring of the tokens and propagation rules offers full freedom for the analysis of different user requirements.

In subsequent steps of model refinement, e.g. at register-transfer level, this global overview of the system effectively supports test generation procedures by restricting the search space of the significantly more detailed model to the solutions of those from the coarse model [FUT95]. Of course the whole process of model refinement is only useful if the results from a previous design phase can really be used to guide the evaluation of the current phase model, leading to the mentioned restricted search space. Therefore, the rules for model refinement are defined.

A correspondence is shown between gate-level models and the dataflow modeling approach. It gives the necessary background to try to adapt methods that are already proved useful at logic circuits. For automatic test pattern generation the PODEM algorithm is selected as an example. Adaptation prepares the algorithm to cope with multi-valued fault model and modeling uncertainty.

For testability analysis a fault-dictionary based method is used. In our approach the dictionary is constructed by concurrent fault simulation for the test set. It is shown, that from the fault dictionary the test-conclusion dependency graph, the input model of integrated diagnostics, can be extracted. Using integrated diagnostics the testability measures of the system can be evaluated and diagnostic design is possible.

1.4 Outline of the Thesis

This work is usually written in third person singular. Whenever I wanted to emphasize my contribution first person plural is used. The four theses are written in first person singular. The work consists of seven chapters and five appendices. Since the various tasks of the work can easily be separated they represent the main parts of the work and are presented in Chapters 2, 3, 4, and 5. These four chapters start with a short introduction of the topic, with the overview of previously done work, then our approach and its contribution is presented. Therefore, overview of the various topics as a separate

chapter is omitted.

Chapter 1 gives an introduction of the thesis. It presents the motivation of the work, its contribution to the field of design automation of dependable systems, and the outline of the thesis.

Chapter 2 presents the used dataflow notation and the systematic process of model construction in the modeling approach. It gives an overview of the features of the approach and a detailed example of a candy automaton.

Chapter 3 shows the technique of model refinement for dataflow networks and defines the rules of model refinement in order to keep the consistency of the model during information addition. An algorithm is presented with which refinement of a model can be checked automatically.

Chapter 4 discusses the methods for test generation and gives an algorithm for test generation of the modeling approach. The algorithm is an adaptation of PODEM. The functioning of the algorithm is presented in the example.

Chapter 5 introduces integrated diagnostics and shows how this approach can be used for testability analysis in the modeling approach. It presents the proof of convergence of testability measures regarding the original model when refinement rules are not broken. Testability analysis is shown on the example.

Chapter 6 gives an extensive application study. The whole process of modeling, test generation, and testability analysis is presented for the well known MEMSY system. During analysis the testability of three different MEMSY configurations is compared.

Chapter 7 presents a summary of this work and gives suggestions for topics of future research and possible applications.

Appendix A gives the list of abbreviations and presents the symbols that are used thorough this work.

Appendix B gives the summary of definitions and derivations that are used in this work.

Appendix C contains the formal notation of the candy automation example that, because of its complexity, is omitted from previous chapters.

Appendix D presents the rather longish formal notation of the MEMSY model, and describes one of its components, the CPU in detail.

Appendix E presents the lists of publications of the author that are appeared or accepted up to the finishing of this work.

Chapter 2

Modeling Approach

The increasing complexity and diversity of applications demands advanced design methods for the development of both hardware and software [RB95, Uba92]. These methods utilize the currently available new technologies such as system-level specification and simulation [Sch92], soft-prototyping, formal design and verification [BBS93], high-level synthesis [CW91, Cam79]. The common background of these technologies is, that they rely on a proper modeling approach.

The choice of a modeling approach fundamentally depends upon the exact task that the model is to be used for. During all phases of testing and diagnostic design, the following most important tasks have to be executed: test generation, fault simulation, testability analysis. Certainly it is desirable that besides the above tasks the approach supports the execution of traditional design tasks like performance evaluation, formal validation, etc. This way the designer has an homogeneous, integrated, model-based design environment.

Since the aim of diagnostics is the identification of the faulty component, component-based models are normally used. Component-based models consist of an explicit description of the systems structure (i.e. the interconnections between components) together with the description of each component. The model of the system is referred to as the *system model*, while the models of components are called *component models*.

The success of the approach is effected by the method that is selected to describe the functions of components into which the system is divided. The most commonly used methods are: hardware or functional description languages [IEE93, Bro89], Petri nets [Pet62, Mur89, Gen86, Jen90], different types of system graphs [Uba92], dataflow networks [And89, Den85, JA91, Kah74, LM87], control flow-charts [Den85], binary decision diagrams [Ake78],or statecharts [Har87, Har97]. Each of them leads to using different mathematics, different fault models and therefore different evaluation methods.

In the early phases of design one is usually interested only in the inputs and outputs of the components and not in their internal representation. Therefore, component models are functional or behavioral models that regard the component as a black-box. However, in most of the approaches instead of black-box models glass-box models are used

properties:	support for:
simple graphical representation, modularity, compactness, hierarchical description	easy to survey model
direct support for black-box as well as glass-box modeling	model construction is similar to the way the designer thinks in early design phases
stepwise refinement	modeling in multiple phases of system design
distributed nature, expressing both fine- and coarse grain parallelism	description of asynchronous, concurrent, parallel actions
data-driven approach	event-driven representation of real-time computation
referential transparency, atomicity, information protection (no sharing)	inherent support for modeling of fault-tolerant applications
well defined mathematical formalism	formal methods in the evaluation
functional behaviour	system function is a composition of individual functions
direct translation into process algebras, timed Petri nets	validation and timing analysis
direct representation of information propagation	error propagation path are easily tracable, simply check of timeliness and communication delays

Table 2.1: Advantages of the dataflow modeling paradigm

[LCSC94]. In glass-box models details of the internal system representation are also described. This includes a description of the internal state space of the component and/or the internal distribution of the system into sub-components. The sub-components may again be described in different levels of detail (in a black-box view or in a glass-box view), supporting model refinement [Bro95, dBdRR90, Jon89a].

In [CGPT94, CPS96, CPS95, Cse96, CP96] reasoning can be found about dataflow networks, to be the modeling method during diagnostic design. It was shown, that the dataflow notation is well-suitable for conceptual modeling of computing systems in the early design phases [BS93, Sch92], for early validation of computing systems [BBS93] and for performance evaluation [CBBS94, BBC+97].

The main advantages of dataflow models as a description method for fault-tolerant digital computing systems are summarized in Table 2.1. In the left column the properties of the dataflow approach are given, while the right column gives the advantages these properties provide. In the early phase of system design 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.

Usually control related decisions (like implementation of the controller part as a scheduler, control table or FPGA) are done in later steps of the design; therefore, the neglection of data dependencies does not impose any problems during the modeling. This means that the suggested dataflow description and error model is sufficient to describe the system and meaningful architectural-level decisions can be done, without using a more elaborated and complex formalism, e.g. statechart or VDM models . Since these formalisms use dataflow graphs for the description of the structure, in subsequent steps of the design, if it becomes necessary, the dataflow model can be transformed without problems into the more elaborated models in order to describe control aspects of the system with greater detail.

2.1 Dataflow Networks

The dataflow programming paradigm is used to describe asynchronous parallel computations, in which data is passed between the nodes of the computation through channels realized by ideal FIFO queues [And89, Den85, KBB86, KS92, Lee91, LM87]. The behavior of the nodes can be deterministic as firstly presented by [Kah74] or non-deterministic as showed by [Jon89b]. The latter formalism extended by the notion of time and priorities [Can93] is the starting point of our modeling approach.

2.1.1 Informal Presentation of the Model

A dataflow network (DFN) is a set of nodes that execute concurrently and exchange data items over unidirectional, point-to-point communication channels. (From now on data items are called tokens.) The dataflow nodes represent the components of the system, its states and describe their data propagation behavior by a simple input-output relation. The channels of the dataflow network symbolize the interaction between the components 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 tokens are inserted into an input channel (input event) or data items are removed from an output channel (output event). Input events describe the arrival of data to the primary inputs, while output events describe the appearance of result on the primary outputs of the system. Tokens represent the data (i.e. messages) passed between the components. Tokens are characterized by their color, that describes the properties of the modelled data (e.g. fault-free, faulty). The graphical representation of a dataflow network is a dataflow graph (DFG), nodes of which are drawn as boxes and channels of which are drawn as directed arcs.

The behavior of a node is defined by the set of firing rules. A node starts a computation (executes a firing rule) as soon as the tokens required by one of its firing rules are available in the input channels and it is in a proper state. After finishing the computation tokens are produced onto the output channels and the node enters a new state. To

the firings execution time can be assigned, that denotes the time of the computation. Since operation of a node is sequential, new computation can only be started after the preceding one is finished. Priority can be assigned to the firings, to solve the problem of computations that could be started concurrently.

2.1.2 Formalism of Dataflow Networks

Dataflow nodes are defined in form of 6-tuples that also consist the definition of the possible token types. Defining the token types locally for the node rather than globally for the whole network allows to describe the transformations of the network and that of the node more concisely and more precisely. It results that coherent mathematical methods can be used to describe model refinement and refinement checking (see later).

Definition 2.1 *A* dataflow node *n is a tuple $(I_n, O_n, S_n, s_n^0, R_n, M_n)$ where:*

> I_n *- set of input channels*
> O_n *- set of output channels*
> S_n *- set of states*
> s_n^0 *- initial state, $s_0 \in S_n$*
> M_n *- set of tokens*
> R_n *- set of firings, $r_n \in R_n$ is a tuple $(s_n, X_{in}, s'_n, X_{out}, \pi)$*
>> s_n, s'_n *- states before and after the execution of the firing, $s_n, s'_n \in S_n$*
>> X_{in} *- input mapping, $X_{in} : I_n \mapsto M_n$*
>> X_{out} *- output mapping, $X_{out} : O_n \mapsto M_n$*
>> π *- priority of the firing, $\pi \in \mathbb{N}$*

$\emptyset$, the empty set is included in M_n. It is necessary to be able to describe computations that do not consume input tokens and/or do not produce output tokens on some of the channels.

In the definition of mappings X_{in} and X_{out} the assumption is used that at a time at most one token is moved by the firings from/to a single channel. It makes the definitions and proofs easier to survey, while it does not restrict the modeling power of the notation. With a little effort they can be generalized for the case when more then one token is moved at a time.

The meaning of *firing rule* $r_n = (s_n, X_{in}, s'_n, X_{out}, 0)$ is that if node n is in state s_n and $\forall i_n \in I_n$ contains at least the tokens $X_{in}(i_n)$, then r_n is potentially selected for execution. Potentially means, that it is not sure that r_n will be executed, since multiple firings can be selected for execution simultaneously, and only one of them will be executed. The execution of r_n removes $X_{in}(i_n)$ tokens from $\forall i_n \in I_n$ and outputs $X_{out}(j_n)$ tokens onto $\forall j_n \in O_n$. After execution the node changes its state from s_n to s'_n.

The selection of transition to fire is done according to *priority*, i.e. from the set of potentially executable firings the one with the highest priority is selected. If more than

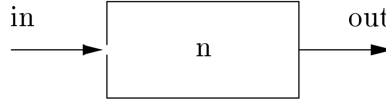


Figure 2.1: An Example Dataflow Network

one firings have the same priority, then selection is done randomly. This causes a non-deterministic behavior of the dataflow node and constitues the foundation of uncertainty modeling.

Definition 2.2 *A* dataflow channel *is an unbounded FIFO queue. It is mapped to exactly one node as an input channel and to exactly one node as an output channel. The state of channel c is denoted by the sequence of tokens it contains:* $S_c = \times^\infty M_c$*, where* $\times$ *denotes the Cartesian product, and* M_c *denotes the set of tokens the channel can contain. The initial state of the channel is described by the null-sequence Ø.*

If channel c is connected to nodes n_1 as input and to n_2 as output then for the token sets M_{n_1}, M_{n_2}, M_c has to hold that $M_{n_1} \subseteq M_c$, $M_{n_2} \subseteq M_c$. Usually a single, homogeneous token set is used through the whole dataflow network. Unbounded channels are necessary to ensure fully asynchronous operation of the nodes. A contrast of unbounded channels are the 1-bounded channels that yield fully synchronous operation, i.e. a node can only start the execution of a firing if the output channels are empty. They are empty, if the successor nodes finished their operation that removed the tokens from the channels.

Definition 2.3 *A* dataflow network *DFN is a tuple* (N, C, S) *where:*

N - set of nodes
C - set of channels (I-input, O-output, and IN-internal channels)
S - set of states; Cartesian product of node and channel states

Note that the definition of DFN does not contain information about the connection of channels and nodes explicitly; it is given in the definition of nodes.

An example

In Figure 2.1 a dummy example is given, in order to introduce the notation that is used in the whole work. The formal description of the network is the following:

network	DFN=({n}, {in,out}, {(s,0,0), (s,ok,0), (s,0,ok), (s,ok,ok)})
nodes	n=({in}, {out}, {s}, s, {ok,∅}, {r1})
firings	`r1=<ok;in=ok;ok;out=ok;0>`

The token set consists of ∅ and a token called ok. The network is made of a single node n and channels in and out. The node has a single state s, while the state of the

channels is either $\emptyset$ or ok, i.e. channels are either empty or they contain an ok token. For the sake of simplicity the FIFO is supposed to be 1-bounded. It is easily observable, how the state of the network is composed from the node and channel states by Cartesian product. Firings are usually printed in `typewriter` style. `r1` in this case removes an ok token from channel in and puts an ok token into channel out. The node remains in state s.

Features of the formalism

If one considers our dataflow formalism, the description of the dataflow nodes correspond to a production system with very simple rules. Both the condition and the action part of the rules consist only a single state and a few tokens. It eliminates the usage of rule matching, that will contribute to a simpler and more effective implementation of the evaluation algorithms. This simplicity is also useful in the formal handling of the models. In the real life many formalisms failed because of their complexity made the use of formal methods extremly tedious if not impossible (e.g. colored Petri nets). Therefore in the formal design of computer systems, the less complex formalisms are preferred (e.g. plain Petri nets, BBDs).

Such a dataflow based production system is used in the market leader CAD tools in early phases of design, that is called structural design. In structural design the data dependencies and data transformation behavior of components can be neglected. That is why in the first step uninterpreted dataflow modeling is suggested and the rule based approach of AI could not be used effectively.

On the other hand, the simplicity of the formalism does not decrease its modelig power, since relations in a most general form describe both the state transition and the output (state transition relation, output relation). However, user friendliness of the notation can be queried. If the formalism is used as an underlying formalism of modeling in a CAD tool, a more comfortable way has to be provided to the user for model editing. In this more comfortable editor the user can work with relations, functions, domain, and codomain. This dataflow notation can be called extended dataflow networks. Thus the user can define an 8 bit full-adder as a function $a + b = c$, where the domain is the set of numbers that can be represented on 8 bits: $a, b = [0, 255]$, while the codomain of the function is the set of numbers that can be represented on 9 bits: $c = [0, 511]$. The 65536 rules that are necessary to describe the full-adder can be generated automatically.

In AI like complex rule based systems whole procedures can be used to describe the condition- and action parts of the rules. As a result the rules will be complex, and as such the completeness and consistency of the rule base can only hardly be checked, which is a natural demand in dependable design, but imposes one of the greatest problems of AI.

In contrast with it, at architectural level, which is above functional level, well structured dependable computing systems can easily be described by dataflow networks, and

the firing rules of the nodes will be simple. Therefore, the completness and correctness of the rules can be checked locally for the individual dataflow nodes with little effort.

2.2 Model Extension

The dataflow formalism itself is suitable for modeling both fault-free and faulty systems. However when modeling faulty systems, one has to incorporate into the model the basic aspects of fault, fault effect, and fault propagation. Usually it is done by the systematic extension of the fault-free model. This systematic extension process is based on the fault model.

At logic-gate level the relationship is strong between the physical defect and the fault. These fault models, e.g. the stuck-at model, are called physical fault models. At higher levels of abstraction, where the tie is more tenuous, logical fault models are used. For logical fault modeling the two most common methods are model perturbation [Sch92, FUT95] and fault nodes in graph models [RS80, Uba94].

In model perturbation a systematic way is developed to perturb the device model. For example possible control faults perturb the control points that switch between operation sequences. It is done usually by using if-then-else or case like structures. The list of such perturbations becomes the fault list. A test is found for each fault in the list.

In graph models the nodes represent the components of the system. For each component the precomputed faulty behaviour is stored in the nodes description. A fault of a component is manifested in form of faulty output. For each component fault the faulty output has to be propagated by a depth-first, forward search from the inputs of the graph towards its outputs.

In this work the second method of logical fault modeling is used. It is more suitable to the dataflow modeling approach, since it is a graph based method. A fault can occur in a processing element, that is denoted by a dataflow node of the graph, and it is propagated along the communication path, that is denoted by arcs of the graph.

The problem of generating the extended model must not be overlooked at high levels of abstraction, where the relation is loose between the highly abstract component description and the physical faults. Therefore; automatic generation of the extended description is hardly if at all possible. Besides manual model composition, two alternatives are left to the designer to generate the extended model:

1. From the specification of the system or from a design decision the specification of the component can be derived. For example the specification states that the system must made up of fail-safe components. It imposes constraints on the implementation of the component: the extended model differs from the original one, such that in the former one, the components are supposed to be fail-safe, e.g. some self-testing facility has to be included into the component during implementation.

2. A similar component was implemented previously and its design is stored in a design library. If the designer decides that the same design should be used, the high-level description can be abstracted from the low-level one by reversing the rules of refinement (see in the next chapter).

2.2.1 Fault Modeling

Our method 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, in the simplest case the tokens that represent the data can be marked either as correct or as faulty. In finer modeling this two markings can be split into several other markings, denoting different kinds of faults. This process of labeling the tokens according to some criteria is called coloring.

A more detailed fault model and a more precise description of the reactions of components to erroneous input values can be defined by using multi-valued coloring. The number of possible faults in the multi-valued fault model is limited only by the number of simultaneously allowed different token colors, which is theoretically unlimited but in practical applications a too large number of token colors often results in unacceptably long evaluation times. As an example of a multi-valued fault model, the tokens can be qualitatively colored according to the severity of the fault effects:

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

As another example consider some floating point operations. The results can either be:

- accurate (within a small range around of the correct value)
- acceptable (within a maximal range around the correct value)
- underflow (below the lower limit of the acceptable range)
- overflow (above the upper limit of the acceptable range)

In the suggested method the faults of the components are manifested in sending tokens that are marked as erroneous. Note, that it strongly resembles the well known gate-level stuck-at fault model of logical circuits, where the internal faults of logical

gates are modeled by erroneous output values. Component faults are described by the erroneous states of the node; therefore, the set of states of the node has to be extended with the erroneous states. In case of faults the behavior of a component is denoted by additional firing rules of the node; hence, the set of firing rules must also be extended.

Using such a fault model the set of error propagation paths can be estimated by tracing the flow of tokens from the fault site in the system (erroneous node) through the components (other error-free or erroneous nodes) to the primary outputs.

2.2.2 Uncertainty Modeling

Uncertainty describes the confidence attached to a given state or behavior of the model and can be used to represent the essentially subjective and incomplete knowledge about the modeled system. Therefore, dealing with uncertainty is a crucial point of the modeling method.

In digital systems one usually has to differentiate between functional non-determinism and non-determinism due to underspecification. The first term relates to the planned non-deterministic behavior of the system, e.g. execution order of concurrently executable processes can not be decided in advance, or the point of time of fault occurance can not be predicted. The second term describes the effect of simplification used during modeling, e.g. at higher levels of abstraction fault modeling is usually more abstract to facilitate simulation or uninterpreted modeling can not handle data dependencies because of the neglected data values.

In state-transition systems uncertainty appears in form of ambiguity during the selection of the next action to be done: In a given state under the same input conditions multiple actions could be executed concurrently leading to different successor states and delivering different output, but for some reason only one of the actions can occur. Different modeling approaches differ in uncertainty modeling, how they describe the selection mechanism among concurrent transitions.

In many engineering systems uncertainty modeling is solved by using stochastic processes that is a probabilistic method. To the possible transitions probabilities are assigned, which express the likelihood that the given transition occurs. Selection then is done randomly according to the probabilistic variables. The whole model is transformed to stochastic processes, most often to Markov-chains, where transient or staedy-state analysis can be done. The analyis results are state probabilities and transition frequencies: the probability that a system is in the given state, a number the transition is executed in unit time. This method relies on the knowledge of transition probabilities and needs quite a fair amount of computational resources for analysis. Examples for such approaches are Petri nets, stochastic activity networks, finite-state machines. From the tools that are mentioned in Chapter 1 Panda, SURF-2, and Depend solve uncertainty modeling this way.

In this work we intentionally avoid using probabilistic models for three reasons:

1. At high levels of abstraction it is hardly possible to extract the state or value probabilities.

2. If probabilities can still be extracted they are so inaccurate, that the evaluation results become useless.

3. Large probabilistic models are difficult to handle.

In our approach uncertainty is modeled by utilizing the non-deterministic behavior of the proposed dataflow networks. Instead of using a probabilistic approach we suggest the following method. Functional non-determinism and the effects of faults are expressed by the firing non-determinism of dataflow nodes, i.e. concurrently executable firing rules describe the fault-free operation and the transition from fault-free to faulty state. The effects of model simplification are described by the value non-determinism of dataflow nodes, i.e. the value of the token sent by the firing is uncertain. For this aim we further extend the set of tokens, node states, and firing rules. This differenciation between firing- and value non-determinism is arbitrary and it serves merely for convenience purposes. Actually, value non-determinism can always be converted into firing non-determinsm.

An example of firing non-determinism is the uninterpreted description of control structures, e.g. an if-then-else construct. In the basic case branching is based on a data value, that is transformed into a single token in the uninterpreted case. As a result branching of the control sequence is done randomly, i.e. the construct is described by two (or more) firing rules that has the same input mapping, pre-state, and priority, but has different post-state according to the two branches.

An example of value non-determinism is the modeling of a faulty component when no information is available whether the component delivers error-free or erroneous results. In this case firing rules of the node can be formulated in such a way that the node sends uncertain, so called x tokens. This solution is similar to test invalidation at system-level diagnostics. Here in the simplest model [PMC78] the statements of the fault-free tester sub-system about the tested sub-system are regarded as correct, whereas the statements of a faulty tester are regarded as uncertain. Of course other test invalidation models [RK78, Sel85] can be adapted too.

For the handling of these firing rules, we suggest to use list based processing in the various model evaluation algorithms. The idea behind list based processing is that the selection among concurrent transitions can be represented like a tree. When a selection is done all other cases are inserted into the list a possible branches. When the selected branch lead to a leaf, processing has to continue on the left-open branches and go on, until all leaves are reached.

A natural question is whether the complexity of the algorithms could be descreased by using a deterministic dataflow formalism. The usage of the non-deterministic notation in the modeling phase is reasonable because of the direct support for uncertainty modeling and a more compact model. Non-deterministic dataflow models can be un-

folded to deterministic ones and vice versa, similarly like finit-state machines. The unfolded model will contain more nodes, channels, and firing rules than the original one. The simpler firing rules allow to implement less complex algorithms, but the increased number of model elements put the effectivity of the evaluation into question. To decide for which models which method is more effective lies outside the scope of this work. Based on the few examined models we decided for the first method.

In contrast to stochastic processes list based processing is a combinatorial approach and this way, at the highest level of abstraction diagnostic uncertainty can be introduced and conditional error propagation can be expressed. It results that the simulation and test generation algorithms deliver a superset of propagated fault effects, and in the model all potential consequences of a fault are incorporated. What we are mainly interested is in, are the fault propagation paths from the inputs to the otputs and withing the system. They tell us whether an input combination is a test for a given fault, or whether a given fault leads to system crash or not. Uncertainty in the modeling phase causes uncertainty of the results, e.g. in some cases the above questions can not be answered uniquely.

Since this rough classification of the results is sufficient at the given level of abstraction and at the solution of this type of problems, it is clear that a probabilistic approach would not give more useful results. Of course one can still use the probabilistic approach. In this case equal probabilities can be assigned to concurrent transitions and analysis results have to be interpreted as follows. A probability of zero denotes non-existence, one denotes existence. Anything else between zero and one denotes uncertainty and can be thrown away. This leads to the same results as the suggested combinatorial approach but for a significantly higher cost.

2.2.3 Aspects of Fault-Tolerance in the Model

The overall behavior of the system (e.g. response for a given test vector, or the system-wide effect of a fault) is determined by the local behavior of its components. Therefore, aspects of fault-tolerance have to be expressed by the description of components. The firing rules of dataflow nodes give the designer full freedom to model error appearance, error disappearance, permanent fault, transient fault, repair, error detection, error correction, fault hiding, fault masking, and error propagation.

To show how these aspects can be described, a single node with one input `in`, one output `out`, and two states `ok` and `fty` is considered. State `ok` denotes the error-free component and state `fty` the erroneous one. There are two types of tokens: `ok` and `fty`. Token `ok` denotes an error-free message and token `fty` an erroneous one. Below are some examples of firing rules for different cases:

error-free operation `<ok;in=ok;ok;out=ok;0>`

The error-free component receives and sends error-free tokens.

erroneous operation `<fty;in=ok;fty;out=fty;0>`

The erroneous component sends erroneous token despite of the received error-free token.

internal fault `<ok;;fty;;0>`

Due to an internal fault the component changes its state from error-free to erroneous.

external fault `<ok;in=fty;fty;out=fty;0>`

As an effect of an external fault, i.e. received erroneous token, the component changes its state from error-free to erroneous.

repair `<fty;in=ok;ok;out=ok;0>`

As a result of repair the error disappears, the component changes its state from erroneous to error-free. Note the external intervene into the system.

error correction `<ok;in=fty;ok;out=ok;0>`

The error-free component is able to detect and correct the received erroneous token and to send an error-free one.

error masking `<fty;in=fty;fty;out=ok;0>`

The erroneous component receives an erroneous token and sends an error-free one, thus the faults causing the errors are masked.

error propagation `<ok;in=fty;ok;out=fty;0>`

The error-free component receives an erroneous token and sends an erroneous one, thus it propagates the error.

Internal fault and repair denotes two special cases of the supposed operation. Usually they effect only the internal state of the component, but do not force it to produce output immediately. Producing output is delayed until some external event happens e.g. arrival of a token at the inputs. Output is produced only as a result of this external event. Output is then a function of the two events. This operation corresponds to the known error latency phenomenon.

To describe this kind of operation two alternatives can be considered. In the first alternative the two firings, e.g. one describing internal fault and one describing fault repair, are always enabled, their condition part is always true; therefore, an infinite fault-repair loop would be possible. To prevent the development of such loops a fault-repair scenario has to be used, that describes the execution sequence of such firings. This alternative is used in the above description of internal fault, but without defining the fault-repair scenario.

In the second alternative the events that effect only the internal state of the component and the subsequent external events are combined into single firings. This leads to a significant increase in the number of firing rules, but prevents infinite fault-repair loops. However, since external control of the internal fault and repair activities is missing, they

can happen fully non-deterministically. This alternative is used in the above description of repair, while the first alternative is used in the examples throughout this work, in order to decrease the number of firings.

2.3 Evaluation of Dataflow Models

The evaluation of possible properties of dataflow networks has a very serious impact on the usefulness of the modeling approach. The more properties can be evaluated by using the same model the higher is the probability that the modeling approach can be used in an integrated design environment. In Chapter 1 the need for evaluation of traditional as well as fault related properties is listed. In this section the possible evaluation methods are listed for the suggested dataflow formalism.

If one wants to implement such a design environment he has multiple choices how evaluations have to be implemented. Currently the evaluation methods for dataflow networks are either direct or indirect methods. In direct methods the properties are evaluated directly on the dataflow model. In this method formal algorithms have to be implemented with a lot of work and the correctness of the implemented algorithms has to be proved.

A much more easier way is to transform the dataflow model into another model, on which the questioned evaluation can easily be executed by using already implemented programs. This method is preferred in each case when the feasibility of a method have to be checked and one does not want to put very much effort into implementation work. In this method the work of algorithm implementation and correctness checking falls off. Instead of it a transformation has to be used and the isomorphism between the two models has to be proved. Later on, if the framework turns out to be useful, the tedious work of adaptation can be done.

This suggestion of using model transformation for the evaluation seems to be in contradiction with the statement in the Introduction that model transformations have to be avoided. The difference is that these transformations can be automated, are proved to be correct, and can be hided from the user. On the contrary manual model transformations need interaction of the user, who has to be an expert of both modeling methods and correctness of the transformation can not be checked.

In the suggested modeling approach the following evaluations can be executed for the extended dataflow model (a + sign denotes evaluations or transformations that have been implemented successfully at our department):

interactive simulation+ In the model construction phase interactive simulation or a so called token game can be used to interactively check the behavior of the system. From this point of view the dataflow model can also be called an executable specification.

The user has always full control over the execution of the dataflow network, i.e.

from the set of potentially executable firings he can select the one to be executed, he can see and modify the state of nodes and the content channels.

validation Formal validation usually consists of evaluation of reachability of given states, checking for dead-locks in the system, finding of home state, deciding boundedness of communication channels.

When using dataflow models validation can be based on checking the properties of the DFN directly [Lee91, Maj94] or indirectly after a dataflow to process algebra [BBS93] or a dataflow to Petri net [ABC+96] translation. In this method first the reachability graph, the space of reachable states, and the state-transition matrix is constructed then evaluation is done by analysis of the graph and the matrix.

temporal analysis+ Temporal analysis aims at the evaluation of temporal properties of the system, e.g. probability of a given state, execution frequency of a given computation, average length of a complex computation, response time of the system.

In dataflow models temporal analysis becomes possible if average execution time is assigned to the firing rules of the dataflow nodes in form of probabilistic variables. Evaluation can be done by Petri net analysis after a dataflow to Petri net transformation [Cse93, CBBS94]. This evaluation usually consists of transforming the Petri net into Markov-chains and analyzing the Markov chains. The state transition probabilities of the Markov-chain correspond to the probabilistic variables that describe the average execution time of the firings.

fault simulation+ Fault simulation is usually executed in order to check the behavior of the system in presence of faults. The initial state of the system corresponds to one of the faulty states and the response of the system to the various input values is examined. These parameters of faulty behavior can be stored in fault dictionaries for further use, e.g. evaluation of the test set (see this work).

Fault simulation of dataflow models is very similar to the interactive simulation; the flow of tokens is evaluated from the inputs of the network to the outputs by executing the enabled firings of the network. It can be implemented in form of discrete event simulation [CGPT94].

test design+ Test design usually aims at generation, evaluation, and optimization of the test set of a system. Test generation derives the set of tests from the description of the system; testability analysis aims at the evaluation of the quality of the test set; finally optimization is done in order to minimize the overhead of testing.

From the point of view of test generation dataflow models strongly resembles gate-level models, therefore gate-level automatic test pattern generation algorithms can be adapted. From the test set and system model the input model of integrated diagnostics can be extracted. It makes evaluation of testability measures and

diagnostic design (selection of tests, ordering of tests) possible (see this work and [TCP95]).

fault effect analysis Failure mode and effect analysis (FMEA) is used to discover the possible effects of a component faults, e.g. disorder in the service of the system, failure of a given function, external faults of components. Based on the knowledge of faults and fault effects it is possible to construct the fault-tree as well as the event-tree of the system.

In dataflow modeled systems fault effects can be extracted from the fault dictionary. Using this knowledge and by tracing down the faulty tokens during fault simulation FMEA of the system can be executed. If the grouping of tokens corresponds the criticality classes of faults and the risk of failures is known and given in form of probability variables, criticality and risk analysis is also possible [CGPT94].

(reliability analysis) Reliability analysis aims at providing the "classical" fault tolerance measures of the system, like reliability, availability, mean time to failure (MTTF), mean time to repair (MTTR), mean time between fault (MTBF). The evaluation is based on the knowledge of the probability that a component will fail in a given time and the probability that the repair will be finished in a given time.

By adding fault occurrence, fault latency and detection probabilities to the nodes in form of probability variables, the dataflow model can serve as a starting point for a more detailed reliability analysis. This analysis is based on stochastic processes and Markov-chains (future work). The state transition probabilities of the Markov-chain correspond to the fault occurrence as well as to the fault detection probabilities of the nodes.

The paradenthesis around reliability analysis denotes that in order to executed this kind of evaluation the dataflow formalism has to be extended by allowing to assign probabilistic variables to the firing rules of the nodes.

2.4 An Example

In this section a simple example is given, in order to present the modeling approach. In subsequent chapters the same example will be used to explain the evaluation methods, e.g. the test generation algorithm. The example is an automaton that sells different type of candies (a vending-machine). In the following the methodical process of model construction is presented. The DFG model of the automaton is shown in Figure 2.2, while the formal description of the network is given in Appendix C. A much more complex example is given in [Cse97] and in Chapter 6. That is the model of MEMSY [CHG+94], an expandable fault-tolerant computer system. The model contains about 50 dataflow nodes in contrast to the 4 nodes of the candy automaton and the number of firing rules of a node often exceeds 200.

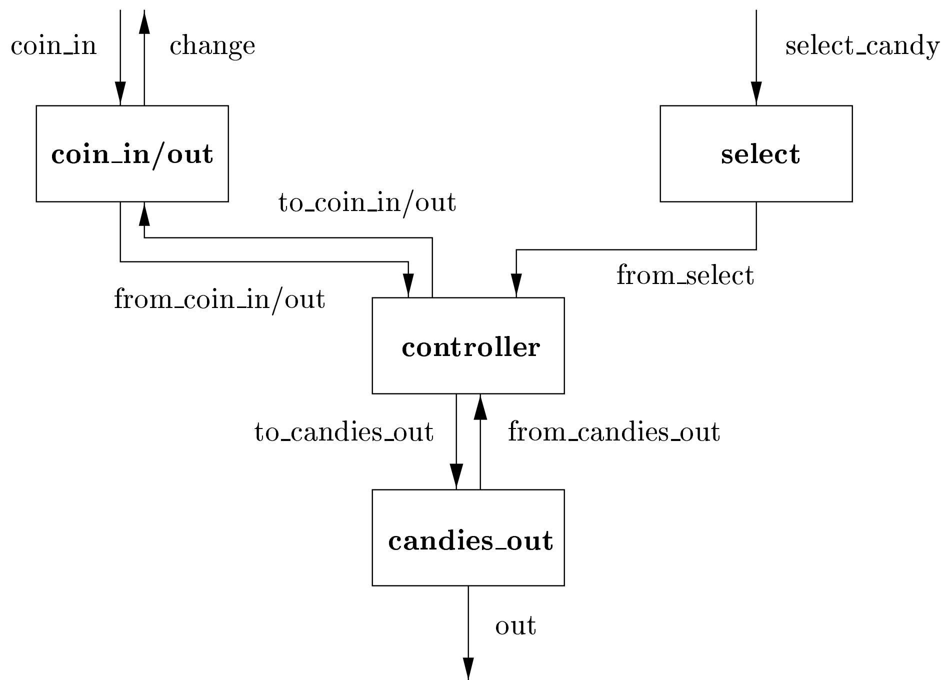


Figure 2.2: DFG of the Candy Automaton

2.4.1 Basic Operation

The candy automaton is designed to take money from the client, to compute the amount of change as a function of the selected candy and the amount of money, and to deliver the change and the candy. The system consists of four components:

coin_in/out In the first step of its operation the component takes the coins the client puts in, computes the sum of their values and sends the result to the controller. In the second step it receives the value of change from the controller and delivers the change to the client. The component is supposed to contain a coin recognizer and some mechanical parts to sort the coins, and an adder to compute the sum.

select It sends the identifier of the candy, that is selected by the client, to the controller. The component is built of a numeric keypad.

controller In the first step of its operation the component receives the value of money from the coin_in/out component, and the identifier of the selected candy from the select component. When the amount of money covers the price of the candy, the controller commands the candies_out component to deliver the candy. In the second step, when candies_out signaled that the candy is delivered, the controller computes the amount of change and tells its value the coin_in/out component. The controller is implemented by a microcontroller.

candies_out It delivers the candies according to the command of the controller. The end of the process is reported back to the controller. This component is typically

a mechanical equipment.

In the basic operation all of the components have a single state `ok` that denotes the error-free component and is never left. In the system only error-free messages are sent, they are denoted by `ok` tokens. The firing rules describe the error-free operation of the components.

2.4.2 Extended Operation

In extended operation, the structure of the model remains unaltered. The set of tokens is extended according to the fault model, and the extension of the set of component states corresponds the supposed errors of the components, while the set of firings represents the changes of the token set as well as the state space.

Fault model (with uncertainty)

The fault model of the candy automaton is a slightly modified version of the fault model presented in the previous section. Faults are identified by the rough, and for the sake of the compactness of the example, simplified classification of the results delivered by the components. The different token colors have the following meaning:

`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 does not deliver results at all, e.g. due to a fatal fault.

`x` token is used to express uncertainty. `x` token is sent if the result can be either correct or incorrect, depending on the input, the state, and the implementation of the component.

State space

The possible states of the components (see Table 2.2) are explained in the following:

coin_in/out The erroneous state of the component is denoted by `rec`. It describes the error that the component does not recognize correctly the type of the coin.

select The erroneous state of select is denoted by `cont`. It describes the possible result of the contact fault of the numeric keypad.

controller The internal faults of the microcontroller leads to the erroneous state `int` in which the component sends incorrect data values.

component:	error
coin_in/out	ok rec
select	ok cont
controller	ok int
candies_out	ok ctrl stk

Table 2.2: Set of States of the Components

candies_out Candies_out has two erroneous states. A control fault may cause that a wrong type of candy will be delivered, this is denoted by state `ctrl`. A candy may stuck within the mechanical part of the component. It leads to state `stk` where the component can not deliver candies any more.

The size of the set of possible component faults is intentionally kept as low as it is shown in the table, in order to preserve the simplicity of the example.

Firing rules

The extended set of firing rules has to describe both the error-free and the erroneous operation. Due to space limitation only the firing rules of the candies_out component are explained here in detail. They are given separately for the three states. In state `ok` the following rules describe the behavior:

$$\texttt{r1} = < \texttt{ok}; \texttt{to_candies_out} = \texttt{ok}; \texttt{ok}; \texttt{from_candies_out} = \texttt{ok}, \texttt{out} = \texttt{ok}; 0 >$$
$$\texttt{r2} = < \texttt{ok}; \texttt{to_candies_out} = \texttt{inc}; \texttt{ok}; \texttt{from_candies_out} = \texttt{ok}, \texttt{out} = \texttt{inc}; 0 >$$
$$\texttt{r3} = < \texttt{ok}; \texttt{to_candies_out} = \texttt{x}; \texttt{ok}; \texttt{from_candies_out} = \texttt{ok}, \texttt{out} = \texttt{x}; 0 >$$

Rule `r1` denotes the basic operation. Rule `r2` states that, if the controller sends incorrect information, the wrong type of candy will be delivered to consumer. Despite the erroneous information the component report the finishing of delivery correctly. According to rule `r3`, the type of delivered candy will be uncertain, in case the information that is received from the controller is uncertain. In this case too, finishing of delivery is reported error-free.

In state `ctrl` the following firing rules describe the erroneous behavior of candies_out:

$$\texttt{r4} = < \texttt{ctrl}; \texttt{to_candies_out} = \texttt{ok}; \texttt{ctrl}; \texttt{from_candies_out} = \texttt{ok}, \texttt{out} = \texttt{inc}; 0 >$$
$$\texttt{r5} = < \texttt{ctrl}; \texttt{to_candies_out} = \texttt{inc}; \texttt{ctrl}; \texttt{from_candies_out} = \texttt{ok}, \texttt{out} = \texttt{x}; 0 >$$
$$\texttt{r6} = < \texttt{ctrl}; \texttt{to_candies_out} = \texttt{x}; \texttt{ctrl}; \texttt{from_candies_out} = \texttt{ok}, \texttt{out} = \texttt{x}; 0 >$$

Rule r4 denotes, that although the received information is error-free, if the control part commands the mechanic erroneously, the type of the delivered candy will be incorrect. In contrast to that, rule r5 states, that if the received information is erroneous, the type of candy can be either correct or false. It is a case of multiple errors within the system, and is a source of uncertain error propagation. Rule r6 is a combination of rules r4 and r5. Success of delivery is independent of the type of delivered candy. This is described by the from_candies_out=ok output mapping of the three firings.

In state stk the following firing rules are necessary to describe the erroneous behavior of the component:

$$\texttt{r7} =< \texttt{stk}; \texttt{to_candies_out} = \texttt{ok}; \texttt{stk}; \texttt{from_candies_out} = \texttt{dead}, \texttt{out} = \texttt{dead}; 0 >$$
$$\texttt{r8} =< \texttt{stk}; \texttt{to_candies_out} = \texttt{inc}; \texttt{stk}; \texttt{from_candies_out} = \texttt{dead}, \texttt{out} = \texttt{dead}; 0 >$$
$$\texttt{r9} =< \texttt{stk}; \texttt{to_candies_out} = \texttt{x}; \texttt{stk}; \texttt{from_candies_out} = \texttt{dead}, \texttt{out} = \texttt{dead}; 0 >$$

These rules express the idea, that if the mechanical part of the component get stuck for some reason, no candy delivery is possible anymore. Therefore, in this case the success of delivery is not reported either.

2.5 Contribution

In this chapter I presented a formal method to solve fault modeling as an integrated part of the architectural design of computing systems. The method is based on an asynchron non-deterministic dataflow notation and its advantage is that is can be used even in early phases of the design. The main steps of the method correspond to the design flow of HW-SW codesign, which is one of the most promising new design method.

- I gave an approach to describe the faults and the fault propagation of the functional units of the system by means of a dataflow formalism. The dataflow nodes that desribe the fault-free behaviour of the components have to be extended by the description of the erroneous behaviour: 1) the state space has to be extended by the erroneous states 2) the token set has to be extended by tokens that denote the qualitatively grouped erroneous data items 3) the set of firing rules has to be extended by firing rules that handle the erroneous states and data items. This way I am able to model the basic fault-related effects of the components: fault appearance, repair, fault propagation, fault detection, fault signaling, error correction, fault hiding.

- I gave a solution for the problem of representing uncertainty. Firing non-determinism of the dataflow networks describes the random execution order concurrent ecents of the system. Value non-determinism describes erroneous data items that can not be grouped uniquely. For this reason one has to extend the set of tokens by tokens that describe uncertain data items and the set of firing rules by firing

rules that handle these additional tokes. Therefore, I am able to describe the uncertainty caused by the lack of information in early design phases.

Chapter 3

Model Refinement

It is well accepted now that the development of complex systems is most adequately carried out by going through a sequence of development phases, that are often referred to as levels of abstraction. In these successive phases a system or system component is described in more and more detail until a sufficiently detailed description or even an efficient implementation of the system is obtained. The individual steps of such a process can be captured by appropriate notions of refinement. In a refinement step, parts or aspects of a system description are made more complete or more detailed.

In order to check whether the refinement step brought any inconsistency into the model, the notion of refinement has to be formalized. Although the stepwise refinement of an inconsistent specification will still lead to an inconsistent implementation, without formalized rules the refinement of a consistent specification could lead to inconsistent implementation. In [Bro95, Jon89a] some approaches of model refinement are discussed while [dBdRR90] gives a broad overview of the topic.

Model refinement is a well known concept in the formal design of computing systems. Unfortunately its importance, that is emphasized in the literature, has not been discovered yet by many developers implementing CAD tools. Therefore, a lot of these tools nowadays support hierarchical modeling, but do not give support for consistency checking between the different models at different levels of abstrction. In this chapter the refinement rules of dataflow models are elaborated. By this novel step our dataflow approach becomes usable in hierarchical modeling.

3.1 Approaches of Model Refinement

In most cases of modeling distributed systems, refinement is formulated for interactive system components, i.e. active components communicating with each other by sending messages via interconnecting channels. In classification of refinement usually two views of a system can be distinguished:

black-box view In the black-box view of a system component, only its interaction with its environment is described. The black-box view consists of a syntactic

interface (input and output channels, and message types) and a semantic interface (relationship between input and output messages).

glass-box view In the glass-box view details of the internal system structure are described. This includes a description of the internal state space of the system and/or the internal distribution of the system into subsystems. The subsystems may again be described in different levels of detail.

Clearly our dataflow description defines a glass-box view of the system. The aspects of glass-box view include the internal representation of the components in one of two forms: state transition system or distributed system. In the former case the component is described by the internal states and firing rules as given in Definition 2.1. In the latter case the node is defined by a dataflow sub-network. If the internal representation is given by a state transition system, the easiest way to create the description is to use state machines that have a well elaborated formal background [CL89]. In [Cse96] the mapping between DFN and non-deterministic finite-state transducers (NDFST) is shown.

The three basic types of refinement of systems with glass-box view are the following:

communication history refinement Communication history refinement changes the syntactic interface of the component, that is the number of input and output channels as well as their message types.

state space refinement The state-oriented description uses a state transition function/relation to describe the behavior of the component. State space refinement changes the set of states of the component.

distribution refinement Distribution refinement refines the component into a subnetwork, i.e. instead of the state transition relation the internal representation of the component is given by a sub-network. The state of the component is then composed from the states of the sub-components.

As it can be seen from the above, refinement is usually done for components, since refinement of a component is always a refinement of the system, due to the fact that the composition operators (parallel, sequential, feed-back composition) are monotonic with respect to refinement. The proof of this compositionality must always be done for the actual modeling language.

3.2 Dataflow Refinement

In this section we formalize the refinement rules for dataflow networks. This is an adaptation of the previously mentioned three basic refinement types of glass-box systems. Compositionality of the used dataflow formalism is proved in [Jon89b]. It results, that refinement of a node is always a refinement of the network.

When switching from one abstraction level to another, the designer can change the dataflow model in many ways. We refer to such a change as a refinement, if and only if the structure of the network remains unaltered [1], and only the behavior of the model is modified. In the dataflow notation it will mean the modification of the dataflow nodes.

Changing the behavior of a dataflow node will always result in changing the firing rules of the nodes. These changes can originate from:

1. changing the token set M_n,
2. changing the set of states S_n,
3. substituting the node with a DF sub-network.

The first two cases are covered by *domain refinement*, while the third case is covered by *structure refinement*. The following subsections present the rules of refinement for these two cases. In general, a refinement of a dataflow network is the composition, i.e. consecutive application, of these two cases.

3.2.1 Domain Refinement

In domain refinement (DR) either the token set or the state space of a dataflow node is changed. In domain refinement the set of firing rules is adapted to the changes of the token set and to the changes of the state space as well.

Suppose that $n = (I_n, O_n, S_n, s_n^0, R_n, M_n)$ and $\hat{n} = (\hat{I}_n, \hat{O}_n, \hat{S}_n, \hat{s}_n^0, \hat{R}_n, \hat{M}_n)$ are the dataflow nodes of two different DFNs. Moreover, a transformation $\mathcal{R}_d$ is given, that:

$$\mathcal{R}_d : n \mapsto \hat{n}$$

Definition 3.1 *The transformation $\mathcal{R}_d$ is a* domain refinement *for node n, iff $\hat{M}_n$ is a refinement of M_n; $\hat{I}_n = I_n$; $\hat{O}_n = O_n$; $\hat{S}_n$ is a refinement of S_n; $\hat{s}_n^0 \in \mathcal{R}(s_n^0)$; and $\forall \hat{r}_n^i \in \mathcal{R}_d(r_n)$ holds that $\hat{s}_{pre}^i \in \mathcal{R}_d(s_{pre})$, $\hat{s}_{post}^i \in \mathcal{R}_d(s_{post})$, $\hat{im}^i \in \mathcal{R}_d(im)$, $\hat{om}^i \in \mathcal{R}_d(om)$, and $\hat{\pi}^i = \pi$, where $\hat{r}_n^i = (\hat{s}_{pre}^i, \hat{im}^i, \hat{s}_{post}^i, \hat{om}^i, \hat{\pi})$ and $r_n = (s_{pre}, im, s_{post}, om, \pi)$.*

Note that for better readability the notation $r_n = (s_{pre}, im, s_{post}, om, \pi)$ is used instead of $r_n = (s_n, X_{in}, s_n', X_{out}, \pi)$. Using the input and outputs mappings $im \in IM$ and $om \in OM$ instead of the sequences corresponds to our assumption that only one token is received/sent at a time on a channel. $IM = \times_{ni} M_n$ and $OM = \times_{no} M_n$, where $ni = \|I_n\|$, $no = \|O_n\|$. ($\times$ denotes the Cartesian product and $\|A\|$ the cardinality of A.) Since $\hat{M}_n$ is a refinement of M_n and the number of input and output channels is unchanged, $I\hat{M}_n$ is also a refinement of IM_n. The same is true for $O\hat{M}_n$ and OM_n.

In the following two examples are given for domain refinement. The first shows token set refinement, while the second presents state set refinement. The dataflow models of

[1] Only some very restricted changes are allowed in the structure of the DFG, they will be discussed later in this chapter.

the examples do not have relevant semantic content, they are thought solely to present the syntactical effect of the transformation.

Example 1: domain refinement

Let $n = (\{in\}, \{out\}, \{on, off\}, on, \{r1, r2\}, \{a, b\})$ and $\hat{n} = (\{in\}, \{out\}, \{on, off\}, on, \{\hat{r}1, \hat{r}2, \hat{r}3, \hat{r}4\}, \{aa, ab, ba, bb\})$ two dataflow nodes and $\mathcal{R}$ a transformation, that transforms n into $\hat{n}$. The firing rules are the following:

$$\mathtt{r1} =< \mathtt{on}; \mathtt{in} = \mathtt{a}; \mathtt{off}; \mathtt{out} = \mathtt{a}; 0 >$$
$$\mathtt{r2} =< \mathtt{off}; \mathtt{in} = \mathtt{b}; \mathtt{on}; \mathtt{out} = \mathtt{b}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{1} =< \mathtt{on}; \mathtt{in} = \mathtt{aa}; \mathtt{off}; \mathtt{out} = \mathtt{aa}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{2} =< \mathtt{on}; \mathtt{in} = \mathtt{ab}; \mathtt{off}; \mathtt{out} = \mathtt{ab}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{3} =< \mathtt{off}; \mathtt{in} = \mathtt{ba}; \mathtt{on}; \mathtt{out} = \mathtt{ba}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{4} =< \mathtt{off}; \mathtt{in} = \mathtt{bb}; \mathtt{on}; \mathtt{out} = \mathtt{bb}; 0 >$$

The transformation $\mathcal{R}$ of the example is a domain refinement, based on Definition 3.1. If, for example, $\hat{r}4$ is modified to $\hat{\mathtt{r}}\mathtt{4} =< \mathtt{off}; \mathtt{in} = \mathtt{ab}; \mathtt{on}; \mathtt{out} = \mathtt{bb}; 0 >$, $\mathcal{R}$ is not a domain refinement any more, since $\hat{r}4 \in \mathcal{R}(r2)$ but $(ab) \notin \mathcal{R}((b))$.

Example 2: domain refinement

Let $n = (\{in\}, \{out\}, \{good, fty\}, good, \{r1, r2, r3\}, \{a, b\})$ and $\hat{n} = (\{in\}, \{out\}, \{good, cold, hot\}, good, \{\hat{r}1, \hat{r}2, \hat{r}3, \hat{r}4, \hat{r}5\}, \{a, b\})$ two dataflow nodes and $\mathcal{R}$ a transformation, that transforms n into $\hat{n}$. The firing rules are the following:

$$\mathtt{r1} =< \mathtt{good}; \mathtt{in} = \mathtt{a}; \mathtt{good}; \mathtt{out} = \mathtt{a}; 0 >$$
$$\mathtt{r2} =< \mathtt{good}; \mathtt{in} = \mathtt{b}; \mathtt{fty}; \mathtt{out} = \mathtt{b}; 0 >$$
$$\mathtt{r3} =< \mathtt{fty}; \mathtt{in} = \mathtt{a}; \mathtt{fty}; \mathtt{out} = \mathtt{c}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{1} =< \mathtt{good}; \mathtt{in} = \mathtt{a}; \mathtt{good}; \mathtt{out} = \mathtt{a}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{2} =< \mathtt{good}; \mathtt{in} = \mathtt{b}; \mathtt{cold}; \mathtt{out} = \mathtt{b}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{3} =< \mathtt{good}; \mathtt{in} = \mathtt{b}; \mathtt{hot}; \mathtt{out} = \mathtt{b}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{4} =< \mathtt{cold}; \mathtt{in} = \mathtt{a}; \mathtt{cold}; \mathtt{out} = \mathtt{c}; 0 >$$
$$\hat{\mathtt{r}}\mathtt{5} =< \mathtt{hot}; \mathtt{in} = \mathtt{a}; \mathtt{hot}; \mathtt{out} = \mathtt{c}; 0 >$$

The transformation $\mathcal{R}$ of the example is a domain refinement, based on Definition 3.1. If, for example, $\hat{n}$ is changed to $\hat{n} = (\{in\}, \{out\}, \{good, cold, hot\}, hot, \{\hat{r}1, \hat{r}2, \hat{r}3, \hat{r}4, \hat{r}5\})$, $\mathcal{R}$ is not a domain refinement any more, since $hot \notin \mathcal{R}(good)$.

3.2.2 Structure Refinement

In structure refinement (SR) a single dataflow node is split (refined) into a sub-network of many nodes, while the token set and the state space remains unaltered. Changes of the network structure does not effect the other nodes as well as the original channels, only the "inner structure" of the refined node is changed. To define structure refinement the definition of firing sequence is necessary:

Definition 3.2 *The* firing sequence fs *of a dataflow network DFN=(N,C,S) denotes the consecutive execution of firing rules of the nodes of the network. It leads from one state of the network to another, and is denoted by* $fs = s \overset{r_i}{\leadsto} s_x \dots s_y \overset{r_j}{\leadsto} s'$, *where* $s, s' \in S$ *are the starting and the final states,* $s_x \dots s_y \in S$ *are intermediate states, and* $r_i \dots r_j \in R = \bigcup_{n \in N} R_n$ *are elements of the firing sequence.*

Note, that between s and s' usually many firing sequences are possible. If s' is not given, the firing sequence is called infinite, otherwise it is called finite. The set of firing sequences of the network is denoted by FS_{DFN}. The input mapping im of a firing sequence is a Cartesian product $im \in IM = \times_{c \in I} M_c$. It denotes the tokens that are removed by the firing sequence from the input channels of the network. Accordingly $om \in OM = \times_{c \in O} M_c$ denotes the tokens that are produced on the output channels of the network. As in case of firings, firing sequences too move at most one token from/to a single channel.

Suppose that $n = (I_n, O_n, S_n, s_n^0, R_n, M_n)$ is a dataflow node, and $DFN = (N, C, S)$ is a dataflow network. Moreover a transformation $\mathcal{R}_s$ is given, as:

$$\mathcal{R}_s : n \mapsto DFN$$

Definition 3.3 *The transformation* $\mathcal{R}_s$ *is a* structure refinement *for node n, iff* $M_n = M_m$, $\forall m \in N$; $(I_n \cup O_n) \subset C$ *such that* $I_n = I$ *and* $O_n = O$; $\forall s_i \in S_n$, $\mathcal{R}(s_i) = \hat{S}_i \subset \times_{m \in N} S_m$ *such that* $\hat{S}_k \cap \hat{S}_l = \emptyset \, \forall k, l$; *and* $\forall r_i \in R_n$, $\mathcal{R}(r_i) = \hat{FS}_i \subset FS_{DFN}$ *such that* $\hat{FS}_k \cap \hat{FS}_l = \emptyset \, \forall k, l$ *and* $\forall fs_i \in \hat{FS}_i$ $s(fs_i) \in \mathcal{R}(s_{pre}(r_i))$, $s'(fs_i) \in \mathcal{R}(s_{post}(r_i))$, $im(r_i) = im(fs_i)$, *and* $om(r_i) = om(fs_i)$.

The definition states that in structure refinement a dataflow node is transformed into a dataflow network. The transformation transforms the input and output channels of the node into input and output channels of the network. The set of channels of the networks contains additionally the internal channels. The token set of the nodes of the DFN is equal to that of the initial node. By definition, the token set of the internal channels is also the same. States of the original node are transformed to the global

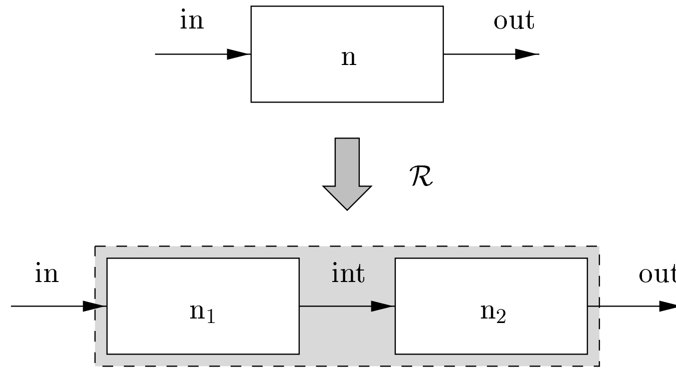


Figure 3.1: Example of Structure Refinement

states of the nodes of the network (Cartesian product of the node states) in such a way, that to one state of the node a subset of global states is assigned, but to one global state only one state of the original node is assigned. The firing rules of the node are transformed to firing sequences of the network in such a way, that the tokens removed by the firing sequences from the input channels of the network and the tokens removed by the firings from input channels of the node are the same, while the pre- and post states of the firing sequences correspond to the pre- and post states of the firing rules.

In the following an examples is given for structure refinement. Again the dataflow model is without any relevant semantic content, it is thought solely to present the syntactical effect of the transformation.

Example: structure refinement (see Figure 3.1)

Let $n = (\{in\}, \{out\}, \{good, fty\}, good, \{r_n 1, r_n 2\}, \{a, b\})$ be a node of a dataflow network, $DFN = (\{n_1, n_2\}, \{in, out, int\}, \{(g, g, X), (g, f, X), (f, g, X), (f, f, X\ )\})$ a dataflow network, and $\mathcal{R}$ a transformation, that transforms n into DFN. In the state of the network g stands for *good*, f for fty, and X denotes the state of the channels. The firing rules of the nodes are the following:

$$\mathtt{r_n1} =< \mathtt{good}; \mathtt{in} = \mathtt{a}; \mathtt{good}; \mathtt{out} = \mathtt{a}; 0 >$$
$$\mathtt{r_n2} =< \mathtt{good}; \mathtt{in} = \mathtt{b}; \mathtt{fty}; \mathtt{out} = \mathtt{b}; 0 >$$
$$\mathtt{r_{n_1}1} =< \mathtt{good}; \mathtt{in} = \mathtt{a}; \mathtt{good}; \mathtt{int} = \mathtt{a}; 0 >$$
$$\mathtt{r_{n_1}2} =< \mathtt{good}; \mathtt{in} = \mathtt{a}; \mathtt{fty}; \mathtt{int} = \mathtt{b}; 0 >$$
$$\mathtt{r_{n_2}1} =< \mathtt{good}; \mathtt{int} = \mathtt{a}; \mathtt{good}; \mathtt{out} = \mathtt{a}; 0 >$$
$$\mathtt{r_{n_2}2} =< \mathtt{good}; \mathtt{int} = \mathtt{b}; \mathtt{good}; \mathtt{out} = \mathtt{b}; 0 >$$

The transformation $\mathcal{R}$ of the example is a structure refinement, based on Definition 3.3. If, for example, $r_{n_2}2$ is changed to $\mathtt{r_{n_2}2} =< \mathtt{good}; \mathtt{int} = \mathtt{b}; \mathtt{good}; \mathtt{out} = \mathtt{a}; 0 >$, $\mathcal{R}$ is not a structure refinement any more, since the firing sequence of DFN that corresponds $r_n 2$ would map $a \mapsto out$ instead of $b \mapsto out$.

3.3 Consistency Checking After Refinement

Clearly, the user is not supposed to well dominate the formalism of dataflow notation, and even not the rules of refinement. Therefore, an automated process has to be provided, that takes as input the two models of different abstraction levels and checks whether the changes indicate a refinement of the system. This problem is decidable in contrast to the more general problem of equality of two such models, that is shown to be undecidable [Mur89].

For consistency checking we exploit the correspondence between non-deterministic finite-state transducers (NDFST) and dataflow nodes. For NDFST's there are mature and validated algorithms to check compatibility, is a well known and already solved

problem. Informally our consistency checking algorithm consists of the following steps: comparison of the structures of the networks, transformation of the nodes of the networks into NDFSTs, and bisimulation of the corresponding NDFSTs. This algorithm of checking is in accordance with our statement in Chapter 2, that problems that do not lay in the mainstream of testability analysis should temporarily be solved by existing, well functioning tools.

A non-deterministic finite-state transducer, also called Mealy-automaton, is a finite-state machine, which has a finite state space and upon receiving elements of an input alphabet changes its state and sends elements of an output alphabet. The mappings between input and output as well as between input and successor states are described by a relation instead of a function of the deterministic finite-state machine. The difference between finite-state automaton and finite-state transducer is that the automaton does not produce output.

In general the refinement of a dataflow node is a refinement of the corresponding finite-state transducer. Refinement rules are defined for finite-state transducers too, but the refinement rules of dataflow networks that are used during modeling of technical systems, consist only a small subset of them. Therefore, it is meaningful to deal with dataflow refinement separately. Moreover, composition of finite-state transducers yields a finite-state transducer, and it is unable to express the asynchronousness of operation and infinity of communication channels.

3.3.1 From DFN to NDFST

A non-deterministic finite-state transducer can describe a dataflow node that corresponds to Definition 2.1 or a dataflow sub-network that is a result of the refinement of a dataflow node. The transformation of dataflow nodes and sub-networks to NDFSTs is presented in this section.

In the following definition relations over sets are rather presented as functions over the power sets of the set. For example relation R from X to Y is denoted by: $R : X \mapsto Y^*$, where Y^* is the power set of Y.

Definition 3.4 *A* non-deterministic finite-state transducer *NDFST is a tuple* $(\Sigma, , , S, s_0, \delta, \omega)$ *where:*

- Σ - *input alphabet*
- , - *output alphabet*
- S - *set of states*
- s_0 - *initial states,* $s_0 \in S$
- δ - *state transition function,* $\delta : S \times \Sigma \mapsto S^*$
- ω - *output function,* $\omega : S \times \Sigma \mapsto , ^*$

The definition of strings over the token set is necessary since the order in which the tokens are mapped to the inputs of the node is important, e.g. a on channel 0 and b on

channel 1 is not the same as b on channel 0 and a on channel 1.

Definition 3.5 *A* string *(or word) of length n over a set A is a sequence of symbols $a_1, a_2, \ldots, a_n$, where $n \in I\!N$ and $\forall i = 1, 2, \ldots, n$, $a_i \in A$. The set of such strings is denoted by A^n.*

Definition 3.6 *The* length *of a string is denoted by $|A^n| = n$.*

The set of all strings that can be constructed from a given set A is denoted by $A^\star$. Of course, like the set of natural numbers, $A^\star$ is infinite.

Definition 3.7 *Let A^m and B^n be two sets of strings. B^n is a refinement of A^m, iff $m = n$, and B is a refinement of A.*

Definition 3.8 *Let A and B be two sets and R a relation for which $\forall a_i \in A$ $R(a_i) \subset B$ such that $R(a_i) \cap R(a_j) = \emptyset$ $\forall i, j$. R is then a refinement relation, and B is called a refinement of A.*

The transformation from dataflow nodes to NDFSTs in fact is a simple substitution in which the input and output mappings of the node are described by the input and output alphabet while the set of firing rules is substituted by the state transition function and the output function.

Definition 3.9 *The* DF node to NDFST transformation *consists of the following substitutions:*

$$\Sigma = M_n^i, \; i = \|I_n\|$$
$$, \; = M_n^o, \; o = \|O_n\|$$
$$S = S_n, \; s_0 = s_n^0$$
$$\delta = \{\ldots, ((r_i(s), r_i(X_{in})), r_i(s')), \ldots\}, \forall r_i \in R_n$$
$$\omega = \{\ldots, ((r_i(s), r_i(X_{in})), r_i(X_{out})), \ldots\}, \forall r_i \in R_n$$

In case of structure refinement the transformation of a dataflow sub-network is necessary. In general a dataflow sub-network, that is composed from nodes that correspond to Definition 2.1 works on finite or infinite token sequences instead of single tokens. In contrast, sub-networks arising during structure refinement of a node are special DFNs that work on single tokens, according to Definition 3.1. Note, that the following transformation is restricted to these special sub-networks, and can not be used for general DFNs.

Definition 3.10 *The* DFN to NDFST transformation *consists of the following substitutions:*

$\Sigma = M_n^i,\ i = \|I\|$ *and* $n \in N$
$\Gamma = M_n^o,\ o = \|O\|$ *and* $n \in N$
$S = S_{DFN},\ s_0 = s_{DFN}^0$
$\delta = \{\ldots, ((s(fs), im(fs)), s'(fs)), \ldots\}, \forall fs \in FS$ *of DFN*
$\omega = \{\ldots, ((s(fs), im(fs)), om(fs)), \ldots\}, \forall fs \in FS$ *of DFN*

3.3.2 Bisimulation of NDFSTs

Similarly to the firing sequence of dataflow networks, the multiple step computation, also called action sequence, has to be defined for NDFSTs. In the used terminology it is denoted by the extended output function.

Definition 3.11 *The* extended output function $\bar{\omega}$ *of a given NDFST is a function* $\bar{\omega} : S \times \Sigma^\star \mapsto \Gamma^\star$.

The notion of bisimulation is based on the extended output function. The underlying idea is, that the two NDFSTs are able to "simulate" each others behavior. Usually it is thought to check the equivalent behavior of different NDFSTs that work on the same input and output alphabet, and their states can be proved to be equivalent by relabeling the elements of the state set.

In our implementation bisimulation is thought to prove the equivalence of an original and a refined NDFST. Therefore the definition is slightly different, and the definition of extended output function refinement is necessary.

Definition 3.12 *Let* $\bar{\omega}_1 : S_1 \times \Sigma_1^\star \mapsto \Gamma_1^\star$ *and* $\bar{\omega}_2 : S_2 \times \Sigma_2^\star \mapsto \Gamma_2^\star$ *be two extended output functions.* $\bar{\omega}_2$ *is a refinement of* $\bar{\omega}_1$ *iff* S_2 *is a refinement of* S_1, Σ_2 *is a refinement of* Σ_1, *and* Γ_2 *is a refinement of* Γ_1.

Informally, bisimulation between two NDFSTs exists, iff the state space of $NDFST_2$ is a refinement of $NDFST_1$ and the possible multi-step behavior from each corresponding state pair s_1 and s_2 are "equivalent", i.e. the produced outputs satisfies the rules of refinement. Note that this notion of bisimulation agrees with the definitions of glass-box modeling.

Definition 3.13 *A relation* $\mathcal{R}$ *between* $NDFST_1$ *and* $NDFST_2$ *is called a* bisimulation, *iff* $\bar{\omega}_2$ *is a refinement of* $\bar{\omega}_1$.

Finally, the following theorem has to be proven in order to be able to use bisimulation for checking the refinement of dataflow networks:

Theorem 3.1 *Let* $NDFST_1$ *and* $NDFST_2$ *be two non-deterministic finite-state transducers that are the results of using Definition 3.9 on the dataflow nodes* n_1 *and* n_2, *or the results of using Definition 3.10 on the dataflow node* n_1 *and the dataflow sub-network* DFN. *A bisimulation* $\mathcal{R}$ *between* $NDFST_1$ *and* $NDFST_2$ *exists iff* n_2 *is a refinement of* n_1 *or* DFN *is a refinement of* n.

Proof The proof of the theorem is done in two steps. First the necessary part is proved, then the sufficient part is proved by using indirection.

1. Suppose, that a bisimulation $\mathcal{R}$ exists between $NDFST_1$ and $NDFST_2$. According to Definition 3.13 $\bar{\omega}_2$ is a refinement of $\bar{\omega}_1$. Therefore, by applying Definition 3.12 S_2 is a refinement of S_1, Σ_2 is a refinement of Σ_1, and , $_2$ is a refinement of , $_1$. Now the thread of the proof has to be split according to whether $NDFST_2$ is transformed from a dataflow node n_2 or from a dataflow sub-network DFN.

   In the first case, applying Definition 3.9 yields, that $M_2^{i_2}$ is a refinement of $M_1^{i_1}$, $M_2^{o_2}$ is a refinement of $M_1^{o_1}$, and S_2 is a refinement of S_1. According to Definition 3.7 $i_1 = i_2$, $o_1 = o_2$, and M_2 is a refinement of M_1. Finally Definition 3.1 states, that this is a domain refinement and n_2 is a refinement of n_1.

   In the second case, the result of applying Definition 3.10 is, that M^i is a refinement of $M_1^{i_1}$, M^o is a refinement of $M_1^{o_1}$, and S is a refinement of S_1. Again, according to Definition 3.7 $i = i_1$, $o = o_1$, and M is a refinement of M_1, in special case $M = M_1$. Using Definition 3.3 results, that DFN is a structure refinement of n_1 in the special case. In the general case Definition 3.1 can be applied subsequently. It states, that a domain refinement preceeded the structure refinement.

2. In proving the sufficient part of the theorem, the steps of the proof are different, depending on whether $NDFST_2$ is transformed from the node n_2 or from the sub-network DFN.

   In the first case suppose, that n_2 is not a refinement of n_1, but $\mathcal{R}$ a bisimulation exists between $NDFST_2$ and $NDFST_1$. According to Definition 3.1, M_2 is not a refinement of M_1, and/or $i_2 \neq i_1$, and/or $o_2 \neq o_1$, and/or S_2 is not a refinement of S_1. Using Definition 3.7 results, that $M_2^{i_2}$ is not a refinement of $M_1^{i_1}$ and $M_2^{o_2}$ is not a refinement of $M_1^{o_1}$. Therefore, by applying Definition 3.9, Σ_2 is not a refinement of Σ_1, and/or , $_2$ is not a refinement of , $_1$, and/or S_2 is not a refinement of S_1. Accordingly, Definition 3.12 states, that $\bar{\omega}_2$ can not be a refinement of $\bar{\omega}_1$. Finally it leads to the fact, Definition 3.13, that no bisimulation exists between $NDFST_1$ and $NDFST_2$, that contradicts to our assumption, that $\mathcal{R}$ is a bisimulation.

   In the second case suppose, that DFN is not a refinement of n_1, but $\mathcal{R}$ a bisimulation exists between $NDFST_2$ and $NDFST_1$. It means, Definition 3.3 states it, that $M \neq M_1$, and/or $i \neq i_1$, and/or $o \neq o_1$, and/or S is not a refinement of S_1. Using Definition 3.7 results, that M^i is not a refinement of $M_1^{i_1}$ and/or M^o is not a refinement of $M_1^{o_1}$. Applying Definition 3.10 we get, that Σ_2 is not a refinement of Σ_1 and/or , $_2$ is not a refinement of , $_1$, and/or S_2 is not a refinement of S_1. Definition 3.12 states, that in this case $\bar{\omega}_2$ is not a refinement of $\bar{\omega}_1$. Applying Definition 3.13 results, that no bisimulation exists between $NDFST_1$ and $NDFST_2$. It is in contradiction with the statement, that $\mathcal{R}$ is a bisimulation.

□

```
1:  Ref_Check()
2:  begin
3:    if (Diff_Struct()) then return FAILURE
4:    for (all n_1 ∈ N_1)
5:    begin
6:      find n_2 or SDFN_2 to n_1
7:      transform n_1 to NDFST_1
8:      transform n_2 or SDFN2 to NDFST_2
9:      if not (Bisimulation()) then return FAILURE
10:   end
11:   return SUCCESS
12: end
```

Figure 3.2: Algorithm of Refinement Checking

3.3.3 Checking Algorithm

For refinement checking two different algorithms can be used. The first is a pairing algorithm, which is based on the application of Definitions 3.1, 3.3. It iteratively partitions the sets (token set, set of states, set of firings) within the two nodes under investigation. This way it tries to find consistent refinements of the set pairs, i.e. the refinement of token set, set of states, and set of firing rules has to be consistent. This algorithm is NP-complete. The other, more elegant algorithm has already been mentioned in the beginning of this section: it tries to find a bisimulation between the corresponding nodes. Unfortunately this algorithm is NP-complete too. In the following this checking algorithm is presented.

Since there are plenty of tools for finding bisimulation (Caesar-Aldebaran [FGK+96], Concurrency Factory [CGL+94], JACK [BFG96], Concurrency Workbench [CPS89], Mobility Workbench [VM94], TAV, EPSILON [Wan90]), we do not need to provide an algorithm for it. Therefore, the checking algorithm is restricted to the skeleton, presented in Figure 3.2. Note, that the algorithm is not as simple as it looks like; certain steps hide hard mathematical problems, that are NP-complete, but solution algorithms are well known for these problems, e.g. checking the isomorphism of graphs in Step 3, or finding bisimulation in Step 9. The steps of the algorithm are the following:

Step 3 The structure of the network has to be checked by using Definition 3.3. If the structure is unchanged or the changes represent the refinement of a single node to a dataflow sub-network, the algorithm can be carried on. Otherwise it has to be terminated by FAILURE, since DFN_2 is not a refinement of DFN_1. Function Diff_Struct works similarly to a general function that checks isomorphism of graphs.

Step 4 Now the refinement of the network can be checked by checking the refinement of the individual nodes. Therefore, all nodes of DFN_1 have to be checked.

Step 5 The pair of the node n_1 has to be identified. It is either a node n_2 or a sub-

network $SDFN_2$ of DFN_2. In subsequent steps refinement between the elements of the pair has to be checked.

Step 7-8 Checking is based on bisimulation. Therefore, the elements of the pair are transformed to non-deterministic finite-state transducers. n_1 is transformed to $NDFST_1$, while n_2 or $SDFN_2$ is transformed to $NDFST_2$.

Step 9 A bisimulation R has to be found between $NDFST_1$ and $NDFST_2$. It can be done by one of the before-mentioned tools or algorithms. If bisimulation exists, according to Definition 3.13, the changes of n_1 represent refinement. Otherwise the algorithm has to be terminated by FAILURE.

Step 11 If all nodes satisfied Step 9 the checking of DFN_1 is SUCCESSFUL.

Since the presented algorithm, that is able to check consistency between two DFNs without a priori knowledge is very complex, it is necessary to make thougths about a less complkex method. The designer usually knows what and how he had refined, and what he wanted to reach with it. If he tells about it the design framework in form of trace scenarios, the whole process of consistency checking becomes less complex. The trace scenario should contain each refinement step in detail: what kind of refinement has been done, what were the changes of the token set, the state space, and the firings. (I.e. sets before and after refinement and the related elements of the sets.)

This way refinement can be checked step by step. In each step pairing of the sets can be neglected, since it is described in the trace scenario. Bisimulation also becomes superfluous, since correctness of between the corresponding sets can be simply decided by applying the definitions of refinement. This way the complexity of refinement checking becomes linear that makes handling of great models much more efficient.

3.4 Application of Refinement in Modeling

The rules of dataflow refinement have been defined in previous sections. In this section it is shown how these rules can be used in the proposed modeling approach. Examples are given how the basic types of refinement can describe the changes of the model, how switches between levels of modeling can be expressed by means of refinement, and how uncertainty modeling can influence the rules of refinement.

3.4.1 Model Changes and Refinement

Dataflow refinement in general is a mixture of the two basic refinement types. In the extended dataflow model these basic types correspond to the following changes:

- Domain refinement describes the changes of the state space and that of the token set. Therefore, domain refinement is applied to a node in one of these two cases:

  - If the token set was extended, DR is used in order to incorporate the faults of a different node. In this case the node under refinement has to be prepared to propagate the new, faulty tokens.
  - If the behavior of the component has to be described in more detail, or if further faults of the component are considered and therefore further erroneous states are necessary.

  Domain refinement covers the case when both new token types and new states has to be introduced, e.g. the states of a component are changed, but new token types must be used corresponding to the new states.

- Structure refinement is usually applied if the behavior of a component becomes too complex to be able to describe it in terms of firing rules, or if the component is built from finer components of a lower level, or if the fault of a component arises as a correlated fault of two lower level components.

3.4.2 Multi-Level Modeling and Refinement

The proposed modeling approach is intended to be used in different phases of design, i.e. at different levels of abstraction. In current CAD design environments the main levels of abstraction are usually defined as: *uninterpreted modeling, interpreted modeling, and allocated modeling.* The latter one can be defined as the modeling after HW-SW separation, resource allocation and operator binding. Although the effectivity of our modeling approach in allocated modeling is questionable, we do not suggest to use it for such purposes, the effects of resource allocation has to be back-annotated into higher level models (see Figure 1.1).

Refinement within a given level (hierarchical modeling) is already described in previous sections. In the following the relationship between refinement and switching of modeling levels (multi-level modeling) is shown.

From uninterpreted modeling to interpreted modeling

In uninterpreted modeling the functionality of a component is described by the token transformation relation of the dataflow node. Tokens do not have value, i.e. they are uninterpreted, only their quantity in a channel is considered. The relation is defined over the number of tokens, instead of over the values of tokens.

The extension of the model is done by coloring the tokens according to some user defined criteria. The dataflow model becomes interpreted, but modeling is still uninterpreted. The single token type is substituted by a few types of colored tokens. The changes can be described by the rules of refinement.

In interpreted modeling the functionality of a component is described by the token transformation relation, that works with data values of the tokens. However the number of consumed and processed tokens has to satisfy the token transformation relation of

uninterpreted modeling. Therefore, switching from uninterpreted to interpreted modeling corresponds to a domain refinement, i.e. transition from uniformly colored tokens to tokens that have coloring that corresponds to the data value.

In the extended operation the tokens, that have a data value are colored according to the same user defined criteria, as in the case of uninterpreted modeling. A single "valued" token is substituted by a few colored tokens with the same value, and this substitution corresponds to the rules of refinement. The token transformation relation works as follows: the token color transformation corresponds to the token color transformation relation of uninterpreted modeling, while the value of the token is changed according to the basic, interpreted functioning.

From interpreted modeling to allocated modeling

The main problem in the transition from interpreted to allocated modeling concerns the mapping of functional units to the same resource. The process of resource allocation has not been fully automated yet, however it would be very important from the point of view of CAD systems. Some heuristic algorithms are already known, but manual interaction is necessary in all cases. A very interesting approach is given in [DJ97]. It can deal with fault tolerance too, but can only be used for synthesis from the already validated design.

Although resource allocation lies outside the focus of this work, some of the problems can not be overcome: when allocating two independent units to the same resource, independence is lost. Errors of the units that were earlier independent will behave as correlated errors: i.e. a fault of the resource will cause errors in both functional units.

In the step of resource allocation, the principle of top-down design is violated, since information must be propagated to higher level models. On the one hand, due to HW reuse, already implemented components are introduced into the model from bottom-up. This could be handled by multi-level modeling in form of model abstraction. The rules of abstraction are the inverse of that of model refinement. On the other hand during allocation transformations are used that modify the structure of the system. It is even more characteristic in case of fault-tolerant systems, where fault tolerance is usually achieved by additional structural mechanisms. This second effect of resource allocation can not be handled by model refinement/abstraction, from the point of view of refinement/abstraction not allowed model changes have to be done.

However resource allocation and its effects can be presented in the dataflow model the following way: nodes representing the functional units that are allocated to the same resource has to be drawn together into one node. The state space of the drawn together node is the Cartesian-product of the state space of the nodes. The user has to select which erroneous states of the nodes are result of a fault in the resource, and these states will describe the correlated errors. The token set of the drawn together node is the union of token sets of the nodes. Input channels of the nodes are mapped as input channels of

the drawn together node, while output channels are mapped as output channels.

Firings of the drawn together node are constructed from the firings of the nodes. If a firing does not describe an error-free to erroneous, an erroneous to erroneous, or an erroneous to error free state transition or if the above transitions does not effect erroneous states that describe correlated errors, it is simply extended to contain the full state description. It will then only move tokens among the input and output channels of its original node, and will change only the part in the Cartesian-product that comes from the state of the original node. If the previous condition does not hold the firing describes a correlated error. In this case input-output mapping is the same as in the previous case, but the state space is changed in such a way that other parts of the Cartesian-product are also changed, i.e. these other parts represent the state of the other nodes effected by the correlated error.

These changes of the model effect only the structure of the dataflow network, and leave its behavior unaffected, but the changes of the structure do not correspond the rules of refinement. Therefore it is not a refinement, but the consistency (i.e. state consistency) of the models is still maintained.

Although the proposed framework does not aim model evaluation at this low level of abstraction, i.e. at the level after HW-SW separation and resource allocation, the consistency of the model can be further assured and back-annotation of implementation dependent information is possible.

3.4.3 Uncertainty Modeling and Refinement

If uncertainty is present in the token set or in the state space of the nodes, then it has to be taken into account during refinement. In our case this uncertainty is described by elements of the set, that are composed by drawing together two or more other elements, e.g. x in the example. In this case the refinement rules of sets should be changed slightly.

In normal refinement a set A is partitioned in some way. The refined set B is also partitioned. The refinement relation assigns to each partition of B a single partition of A. If uncertainty is present in the modeling, one or more partitions of A and B will contain elements representing uncertainty. Now the refinement relation assigns to each partition of B only one "certain" and one or more "uncertain" partitions of A. It means, that during refinement uncertain elements become either certain, i.e. they will belong to the partition of one of the elements from which they were drawn together, or they remain uncertain, i.e. they will belong to the refined "uncertain" partition.

When the definition of set refinement is modified accoriding this suggestions, the rules of refinement can also be used without modifications for models which contain uncertainty. The system designer can further rely on the automated process of refinement checking, while he can fully exploit the modeling power of the dataflow modeling approach.

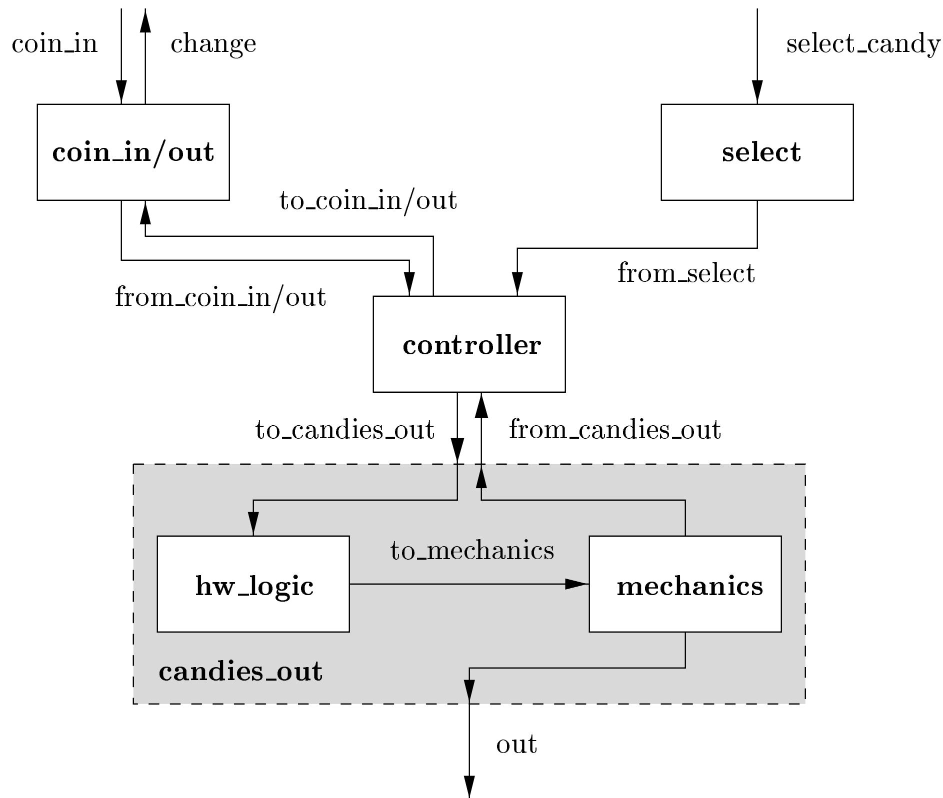


Figure 3.3: DFG of the Refined Candy Automaton

3.5 Refinement of the Example

In our practical example, the candies_out component of the candy automaton is refined into two parts: a hard-wired logic part and a mechanical part. The logic is responsible for controlling the mechanics, and the mechanics delivers the candies. The refined DFG is shown in Figure 3.3, while the complete formal notation is presented in Appendix C.

3.5.1 Refinement of Candies_out

As a result of refinement two additional nodes appear in the dataflow network in place of the candies_out node. A node called hwl represents the hard-wired logic of candies_out, while a node called mechanics describes the mechanical part of candies_out.

They both inherited the token set of candies_out. The input of hwl is the same as that of candies_out, while the outputs of mechanics corresponds to the outputs of candies_out. An additional channel called to_mechanics is established between hwl and mechanics. The states of the nodes (Table 3.1) are explained in the following:

hwl The control fault `ctrl` in the hard-wired logic may cause that a wrong command is delivered to the select unit of the mechanical delivery subsystem.

mechanics A candy may stuck in the mechanics. It leads to state `stk` where the

component:	error
hwl	ok ctrl
mechanics	ok stk

Table 3.1: Set of States of the Components After Refinement

component can not deliver candies any more.

The fault model of the system is the same as before refinement (refer to Chapter 2). Therefore, the token set of the nodes is unchanged.

3.5.2 Refinement checking

In this subsection the refinement checking of the candy automaton is given in order to present the refinement checking algorithm. It will be shown that the changes of candies_out really denote structure refinement. First by applying the definitions then by using the presented algorithm. The three basic steps of the algorithm (structure checking, transformation to NDFST, bisimulation) are discussed separately.

Statement 3.1 *The changes of the candy automaton example represent the structure refinement of the candies_out component and therefore they correspond to the rules of refinement.*

Proof Let the lower index n denote the node candies_out, while hwl denotes the node hwl, $mech$ denotes the node mechanics, and N denotes the sub-network composed of hwl and mechanics. The proof can be done by using Definition 3.3. It will be shown, that the transformation fulfills the four requirements of structure refinement:

- For the token sets is holds that $M_n = \{ok, inc, dead, x\} = M_{hwl} = \{ok, inc, dead, x\} = M_{mech} = \{ok, inc, dead, x\}$.

- The channels are sets for which $(I_n \cup O_n) = \{to_candies_out, from_candies_out, out\} \subset C_N = \{to_candies_out, from_candies_out, to_mechanics, out\}$ such that $I_n = \{to_candies_out\} = I_N = \{to_candies_out\}$ and $O_n = \{from_candies_out, out\} = O_N = \{from_candies_out, out\}$.

- The states are such that $S_{hwl} \times S_{mech} = \{(ok, ok), (ok, stk), (ctrl, ok), (ctrl, stk)\}$ and $\mathcal{R}(s_0) = \{(ok, ok)\}$, $\mathcal{R}(s_1) = \{(ctrl, ok)\}$, $\mathcal{R}(s_2) = \{(ok, stk), (ctrl, stk)\}$ where $s_i \in S_n$ for $i = 0, 1, 2$.

- For the firings and firing sequences holds that $FS_N = \{((ok, ok)$, $r1_{hwl}$, (ok, ok), $r1_{mech}$, $(ok, ok))$, $((ok, stk)$, $r1_{hwl}$, (ok, stk), $r4_{mech}$, $(ok, stk))$, $((ok, ok)$, $r2_{hwl}$,

(ok, ok), $r2_{mech}$, $(ok, ok))$, $((ok, stk)$, $r2_{hwl}$, (ok, stk), $r5_{mech}$, $(ok, stk))$, $((ok, ok)$, $r3_{hwl}$, (ok, ok), $r3_{mech}$, $(ok, ok))$, $((ok, stk)$, $r3_{hwl}$, (ok, stk), $r6_{mech}$, $(ok, stk))$, $((ctrl, ok)$, $r4_{hwl}$, $(ctrl, ok)$, $r2_{mech}$, $(ctrl, ok))$, $((ctrl, stk)$, $r4_{hwl}$, $(ctrl, stk)$, $r5_{mech}$, $(ctrl, stk))$, $((ctrl, ok)$, $r5_{hwl}$, $(ctrl, ok)$, $r3_{mech}$, $(ctrl, ok))$, $((ctrl, stk)$, $r5_{hwl}$, $(ctrl, stk)$, $r6_{mech}$, $(ctrl, stk))$, $((ctrl, ok)$, $r6_{hwl}$, $(ctrl, ok)$, $r3_{mech}$, $(ctrl, ok))$, $((ctrl, stk)$, $r6_{hwl}$, $(ctrl, stk)$, $r6_{mech}$, $(ctrl, stk))\}$.

Moreover, $\mathcal{R}(r1) = \{((ok, ok), r1_{hwl}, (ok, ok), r1_{mech}, (ok, ok))\}$, $\mathcal{R}(r2) = \{((ok, ok), r2_{hwl}, (ok, ok), r2_{mech}, (ok, ok))\}$, $\mathcal{R}(r3) = \{((ok, ok), r3_{hwl}, (ok, ok), r3_{mech}, (ok, ok))\}$, $\mathcal{R}(r4) = \{((ctrl, ok), r4_{hwl}, (ctrl, ok), r2_{mech}, (ctrl, ok))\}$, $\mathcal{R}(r5) = \{((ctrl, ok), r5_{hwl}, (ctrl, ok), r3_{mech}, (ctrl, ok))\}$, $\mathcal{R}(r6) = \{((ctrl, ok), r6_{hwl}, (ctrl, ok), r3_{mech}, (ctrl, ok))\}$, $\mathcal{R}(r7) = \{((ok, stk), r1_{hwl}, (ok, stk), r4_{mech}, (ok, stk)), ((ctrl, stk), r4_{hwl}, (ctrl, stk), r5_{mech}, (ctrl, stk))\}$, $\mathcal{R}(r8) = \{((ok, stk), r2_{hwl}, (ok, stk), r5_{mech}, (ok, stk)), ((ctrl, stk), r5_{hwl}, (ctrl, stk), r6_{mech}, (ctrl, stk))\}$, $\mathcal{R}(r9) = \{((ok, stk), r3_{hwl}, (ok, stk), r6_{mech}, (ok, stk)), ((ctrl, stk), r6_{hwl}, (ctrl, stk), r6_{mech}, (ctrl, stk))\}$ where $r_i \in R_n$ for $i = 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Finally, $s(fs_0) = (ok, ok) \in \mathcal{R}(s_{pre}(r1_{candies_out}) = \{(ok, ok)\}$, $s'(fs_0) = (ok, ok) \in \mathcal{R}(s_{post}(r1_{candies_out}) = \{(ok, ok)\}$, $im(fs_0) = (ok) = im(r1_{candies_out}) = (ok)$, and $om(fs_0) = (ok, ok) = om(r1_{candies_out}) = (ok, ok)$. (The same can be checked by the user $\forall fs_i$ where $i \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$.)

Therefore, the example is the structure refinement of the component candies_out. □

Structure checking

Checking the dataflow network structure is done by comparing the two DFGs. In this case the easiest way to do it is to compare Figure 2.2 and Figure 3.3. Since the set of nodes is increased during refinement, structure refinement of some nodes is suspected. Instead of candies_out two new components hwl and mechanics appeared. The connection of the channels to_candies_out, from_candies_out, and out to the nodes hwl and mechanics corresponds to their connection to candies_out. Connection of the other channels remained unchanged. The additional channel to_mechanics is connected between the two new nodes. Therefore, the changes of the DFG indicate structure refinement of candies_out and correspond to the rules of refinement.

Transformation to NDFST

According to Definition 3.9 the candies_out node is transformed to the following transducer:

NDFST1:

$\Sigma = \{\emptyset, ok, inc, dead, x\}$

$\Gamma = \{(\emptyset, \emptyset), (\emptyset, ok), (\emptyset, inc), (\emptyset, dead), (\emptyset, x), (ok, \emptyset), (ok, ok), (ok, inc), (ok, dead),$

$(ok, x), (inc, \emptyset), (inc, ok), (inc, inc), (inc, dead), (inc, x), (dead, \emptyset), (dead, ok),$
$(dead, inc), (dead, dead), (dead, ok), (x, \emptyset), (x, ok), (x, inc), (x, dead), (x, x)\}$

$S = \{ok, ctrl, stk\}$

$s_0 = ok$

$\delta = \{((ok, ok), ok), ((ok, inc), ok), ((ok, x), ok), ((ctrl, ok), ctrl), ((ctrl, inc), ctrl),$
$((ctrl, x), ctrl), ((stk, ok), stk), ((stk, inc), stk), ((stk, x), stk)\}$

$\omega = \{((ok, ok), (ok, ok)), ((ok, inc), (ok, inc)), ((ok, x), (ok, x)), ((ctrl, ok), (ok, inc)),$
$((ctrl, inc), (ok, x)), ((ctrl, x), (ok, x)), ((stk, ok), (dead, dead)),$
$((stk, inc), (dead, dead)), ((stk, x), (dead, dead))\}$

According to Definition 3.10 the candies_out subnetwork, consisting the nodes hwl and mechanics, is transformed to the following transducer:

NDFST2:

$\Sigma = \{\emptyset, ok, inc, dead, x\}$

$, = \{(\emptyset, \emptyset), (\emptyset, ok), (\emptyset, inc), (\emptyset, dead), (\emptyset, x), (ok, \emptyset), (ok, ok), (ok, inc), (ok, dead),$
$(ok, x), (inc, \emptyset), (inc, ok), (inc, inc), (inc, dead), (inc, x), (dead, \emptyset), (dead, ok),$
$(dead, inc), (dead, dead), (dead, ok), (x, \emptyset), (x, ok), (x, inc), (x, dead), (x, x)\}$

$S = \{(ok, ok, X), (ok, stk, X), (ctrl, ok, X), (ctrl, stk, X)\}$

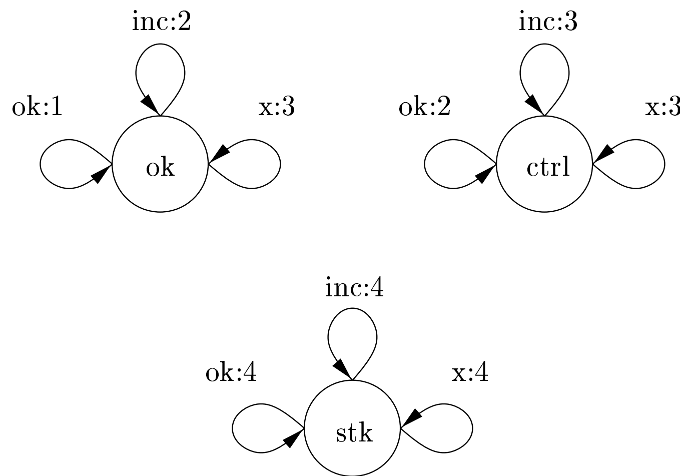
$s_0 = (ok, ok, X)$

$\delta = \{(((ok, ok), ok), (ok, ok)), (((ok, ok), inc), (ok, ok)), (((ok, ok), x), (ok, ok)),$
$(((ok, stk), ok), (ok, stk)), (((ok, stk), inc), (ok, stk)), (((ok, stk), x), (ok, stk)),$
$(((ctrl, ok), ok), (ctrl, ok)), (((ctrl, ok), inc), (ctrl, ok)), (((ctrl, ok), x), (ctrl, ok)),$
$(((ctrl, stk), ok), (ctrl, stk)), (((ctrl, stk), inc), (ctrl, stk)),$
$(((ctrl, stk), x), (ctrl, stk))\}$

$\omega = \{(((ok, ok), ok), (ok, ok)), (((ok, ok), inc), (ok, inc)), (((ok, ok), x), (ok, x)),$
$(((ok, stk), ok), (dead, dead)), (((ok, stk), inc), (dead, dead)),$
$(((ok, stk), x), (dead, dead)), (((ctrl, ok), ok), (ok, inc)), (((ctrl, ok), inc), (ok, x)),$
$(((ctrl, ok), x), (ok, x)), (((ctrl, stk), ok), (dead, dead)),$
$(((ctrl, stk), inc), (dead, dead)), (((ctrl, stk), x), (dead, dead))\}$

Bisimulation

Since many of the mentioned tools are based on reachability analysis, here we use this method to show that bisimulation exists between the two NDFSTs. In reachability analysis the reachability graph of the NDFST is built. The NDFST is started from different states and its state transitions and states are registered. Any of its states can be selected as a starting state and each state will be selected once during the analysis. The reachability graph is a graph, whose nodes denote the states of the NDFST, while the arcs represent the state transitions. Nodes are labeled with the identifier of the state, and arcs are labeled with the input taken by the NDFST and the output produced by the NDFST. In case of NDFSTs the reachability graph is identical to the state-transition



1=(ok,ok) 3=(ok,x)
2=(ok,inc) 4=(dead,dead)

Figure 3.4: Reachability Graph of $NDFST_1$

diagram, but note that the notion of reachability is more general (it is used for Petri nets, DFNs, bisimulation, etc.).

The two reachability graphs are given in Figure 3.4 and Figure 3.5. It is obvious that the two lower nodes in Figure 3.5 can be drawn together into a single node without loss of functionality. After drawing together these nodes, the two reachability graphs are identical, i.e. they have equivalent nodes, and identically labeled arcs. Therefore bisimulation exists between the two NDFSTs.

3.6 Contribution

In this chapter I elaborated the notion of model refinement for the proposed dataflow formalism. If the defined refinement rules are not violated during model refinement then the consistency of the model can be kept. Advantage of the solution is that is is based on formal definitions, and the provided checking algorithm can easily be automated.

- Based on the known idea of refinement, I defined dataflow model refinement in form of two mathematical transformations. One of them is domain refinement $\mathcal{R}_d$, that describes the possible partitioning of the state space and the set of data items. It is done by refining the set of states and tokens of the dataflow nodes and by extending the set of firing rules in order to handle new states and tokens. The other transformation is structure refinement $\mathcal{R}_s$, that describe the splitting of a functional unit into smaller parts. It is done by substituting a dataflow node with a dataflow subnetwork.

- I traced back the problem of dataflow model consistency checking to the known

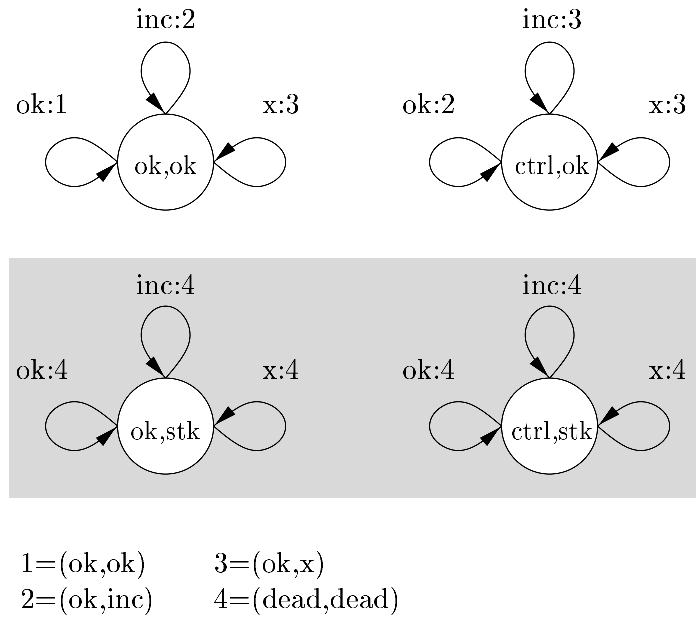


Figure 3.5: Reachability Graph of $NDFST_2$

and already solved problem of finding bisimulation of finite-state automata by means of a dataflow to finite-state automata transformation. I proved that if the two automata $NDFST_1$ and $NDFST_2$ are a result of applying the transformation to either two dataflow nodes n_1 and n_2 or to a dataflow node n_1 and a dataflow DFN, than bisimulation $\mathcal{R}$ between $NDFST_1$ and $NDFST_2$ exists if and only if n_2 or DFN respectively is a refinement of n_1.

- I gave an approach for checking the consistency of the dataflow networks before and after refinement: 1) if no a priori information is present about the refinement, then I use a general algorithm that is based on finding bisimulation between the corresponding finite-state automata 2) if in form of a log file a priori information exists about the steps of refinement, then consistency checking can simply be done by applying the definitions of refinement 3) if the design framework does not allow model changes different from the one allowed by the definitions of refinement, then the rules of refinement are automatically met. All three methods suit for automation.

Chapter 4

Test Generation

Consider some examples from the life-cycle of a fault-tolerant system: in the production phase a system is not declared to be ready for operation, as long as it contains some faulty components; during operation availability could not be realized without on-line reconfiguration of the system by closing out the faulty components; in maintenance the faulty components have to be removed and repaired. In all of the above cases the common process of identifying the faulty components is done by testing.

Usually, testing is based on the execution of the tests of the system. A test in general is described by some input pattern of the system, called test vector. During testing the test vector is applied to the inputs of system, and the result of processing, the so called test result, is delivered on the observable outputs of the system. In order to find out malfunctioning of the system the result of test is compared to the response of the fault-free system. Tests of the system are generated by automated tools, the process of test generation itself is called automatic test pattern generation (ATPG).

4.1 Test Generation Approaches

Test generation for digital systems dates back to the early 60's [RBS67]. Since then, mainly due to the increasing complexity of the addressed systems, a large number of different approaches has been elaborated. Currently test generation approaches can be divided into two groups.

4.1.1 Gate-level Test Generation

Traditionally test design has been carried out at the gate-level [ABF90, Fuj86, BF76]. At this level efficient methods and tools are available for test generation for combinational circuits, e.g. the well known D-algorithm [RBS67] or PODEM [Goe81]. By utilizing the method of the iterative array model [ABF90], these algorithms can be used in test generation for sequential circuits too. Nevertheless, test generation for sequential circuits remains a challenge [Mar78]. On the other hand, complexity problems forced to replace

the sea of interconnected gates by more tractable models in case of large systems.

Replacing gate-level subnetworks by corresponding higher level components has brought in the test generation problem the concept of hierarchy [CP89, Kri87, MH88, Arm92]. It helps to handle complexity in the modeling, but model evaluation does not benefit from it, since the underlying description of the higher level components is still based on gate-level constructs. Further increasing complexity leads to expansion of the search space, that makes evaluation impossible. Additionally, gate-level tests of VLSI components are useless, due to their costly and inefficient implementation.

4.1.2 High-level Test Generation

Using high-level primitives, instead of gate-level structures, has a lot of advantages from the point of view of raising the efficiency of the test generation process: compared to gate-level test generation fewer conflicts are produced; the implication procedure is more powerful; the number of backtrackings is smaller; and sensible paths can be propagated over a complex primitive by a single operation. Test generation methodologies different from those based on knowledge about the gate-level structure of a circuit are called high-level test generation [BMH89, BH85]. The most widely used levels of high-level descriptions are (with typical components):

- register-transfer level (multiplexer, counter, shift register)
- architectural level (CPU, memory, controller)
- multi-level (gate-level, register transfer-level, and architectural level components mixed).

The increasing complexity of high-level models made it necessary to introduce the notion of hierarchy in high-level test generation leading to approaches such as: hierarchical test generation with high-level primitives, hierarchical multi-level test generation.

From the point of view of the applied description method, test generation approaches can be divided roughly into two categories:

structural In the structural approach, the objective for test generation is a system that is described by a network of components.

functional In the functional approach only the behavior of the system is given in terms of inputs, outputs, and internal states.

This distinction is somewhat blurred sometimes, especially when higher description levels are compared to that of gate-level, since high-level models usually are made of a set of interconnected components, i.e. structural description, but the components itself are defined by some input-output mapping, i.e. functional description. Unfortunately, there is no universal high-level test generation algorithm, the used method is highly dependent on the modeling language.

4.2 Dataflow Test Generation

Our method is a good example for the above problem: dataflow networks combine a structural description of the system with a glass-box description of the components. In [BCS91] an algorithm is presented for test pattern generation for dataflow models, but it is limited to deterministic models. Moreover, there are necessary restrictions on the structure of the network: only dataflow nodes with single output can be used, in order to avoid the problem of reconvergent fan-out. Therefore, this algorithm is very inefficient for non-deterministic models.

According to what we stated about the evaluation methods in Chapter 2, there are two possible alternatives. Either the dataflow networks are transformed to other models, or an algorithm must be developed. Although existing high-level algorithms are useful, really effective methods exist mainly at gate-level test generation. Unfortunately dataflow networks can not be transformed to logic-gate level models.

In [TSR93] it is shown that any ATPG network can be described as the object oriented dataflow representation of a constraint network. After this transformation the ever growing set of constraint based test generation tools could be used. The problem with this method is threefold:

1. The semantics of our dataflow notation is unidirectional and not bidirectional, by definition DFNs work in forward direction. It would make backward propagation impossible. This problem can be solved by neglecting the statement about FIFOs, more precisely during backward propagation the FIFOs work as FILOs, and the firing rules are executed in opposite direction. It is similar to the case of logic gates, that can not be driven from the outputs towards the inputs, but logic-gate level ATPG algorithms use backward propagation notwithstanding.

2. The used DF formalism is non-deterministic, while the supposed bidirectional object oriented formalism is deterministic. Likewise all constraint based test generation methods can deal only with deterministic models. The non-determinstic to deterministic conversion of DFNs is rejected in Chapter 2 because of its complexity.

3. Constraint based test generation algorithms are used mainly for logic-gate level test generation. The used heuristics suit probably to the general properties of logic-gate level models and not to DF models.

Therefore this approach is not useful for us. Instead of this we suggest the following approach for the solution of the test generation problem: find as much similarities to powerful gate-level structural methods as possible and try to adapt these techniques to the requirements of dataflow models by exploiting the similarities and considering the differences.

When comparing the dataflow modeling approach to gate-level modeling methods, many similarities can be found. The most important ones are summarized in Table 4.1,

problem	logic gate-level	dataflow
component faults	stuck-at fault model	erroneous messages
structural description	schematic circuit diagram	dataflow graph
functional description	truth/state table	firing rules (transfer relation)
loops/component with states	iterative arrays (flip-flop)	iterative arrays (node)
component with states	self-initialization	self-initialization

Table 4.1: Similarities between gate-level and dataflow ATPG

while below the detailed description is given:

1. One limitation of the suggested fault model, that the faults do not alter the structure of the network. Therefore; errors of the component are modeled as erroneous messages on the outputs of the node. (See more details in Chapter 2.) It is similar to the fault modeling that is used at gate-level, where the faults of a gate are described by using the stuck-at fault assumption at their outputs.

2. The dataflow approach uses the dataflow graph (a directed graph) to describe the structure of the system, while the structure of the logical circuit is described by the schematic circuit diagram, which in turn is a graph too.

3. The nodes of the dataflow graph describe the behavior of the corresponding component by a transfer relation. It is similar to the gate-level approach, that is considered as a structural approach and in which behavior of the logic gates is described by truth or state transition tables.

   In fact both logic gates and functional units are described by forward propagation. (Just as a logic gate does not work in reverse direction by driving its outputs, functional units can not receive messages on their output and produce messages on their inputs.) However during test generation backward propagation is possible by assigning values to the inputs in the knowledge of expected outputs, e.g. if a logical 1 is expected at the output of a 2 input and gate, and one of its input is 1, then from the truth table the necessary value for the other input can be derived.

4. Similarly to gate-level models, the dataflow model can contain feedback loops and components with states. It leads to the problem that the generated tests are composed from multiple steps and re-entering of the same component during testing becomes necessary. Just like in the case of sequential logical circuits the problem can be solved by cutting the loops and constructing the iterative array model[1]. (More about using the iterative array model can be found in the next section).

[1]Note that there are more effective methods for solving this problem than the iterative array model [ABF90].

5. The functional units have states; therefore, testing of a system has to be started from a predefined initial system state. It resembles the initialization of the flip-flops of sequential circuits. (In practical dataflow models examined till yet there was no need for the search of a self-initialization sequence.)

This correspondence provides the practical background when adapting the wide palette of test generation methods (PODEM, D-algorithm, FAN, composite justification [Szi79b, Szi79a], etc.) that were elaborated for sequential logical circuits under the stuck-at fault assumption. As a representative example of the possible methods, we selected the well known PODEM (Path Oriented DEcision Making) and D-algorithms [ABF90, Goe81]. In this work only PODEM is dealt with in detail, for the description of the D-algorithm please refer to [CPS95]. Selection was done in the hope, that by benchmarks PODEM was shown to be more efficient than the D-algorithm for logical circuits with error-correction-and-translation functions and at least as effective for other circuits.

4.3 PODEM for Dataflow Models

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 algorithm is characterized by a direct search process: it directly manipulates the PIs and tries to propagate the error to the POs. In each step of the algorithm checking and implication is done. To keep track of the still open propagation problems a set is maintained during the algorithm: the *D-frontier* contains the components from the outputs of which the error has to be propagated towards the POs. The advantage of PODEM over other test pattern generation algorithms is that due to the direct search:

- backward propagation is not necessary.
- the J-frontier can be eliminated,

4.3.1 PODEM for Sequential Circuits

A very useful property comes from the fully asynchronous nature of the dataflow networks and from the FIFO like behavior of the channels. Tokens can be consumed by a node only in the order as they were produced by the preceding node. It causes that single firings of cyclic execution of a node do not get blurred, they can easily be recognized and separated. Therefore, for the primary aim of test generation, i.e. that of providing the test vector, no special considerations have to be done for handling sequential behavior.

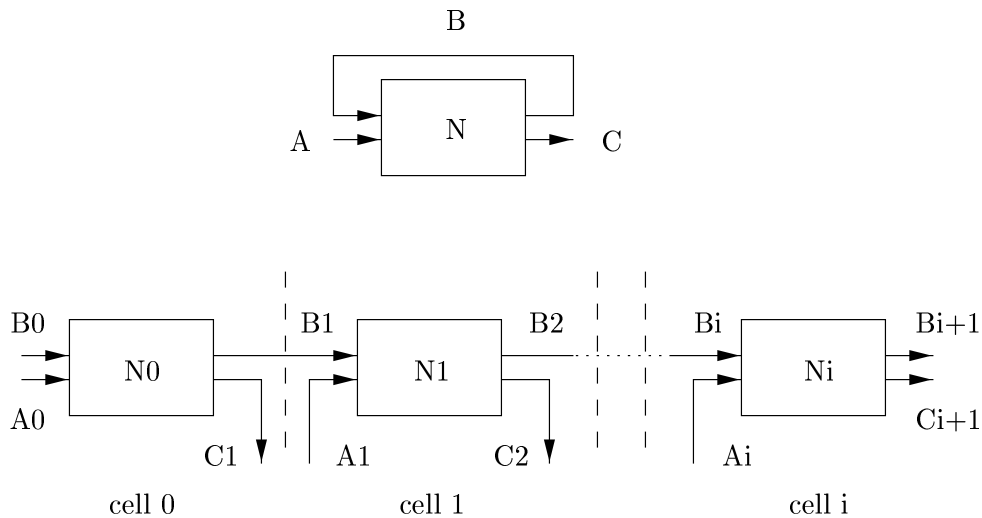


Figure 4.1: Iterative array model of a dataflow network

However, if the user is interested in the token propagation path during testing, a method has to be used, that can provide the necessary information, in our case the unfolded dataflow network of the test. It can either be done by simulating the processing of the test vector, or the test generation algorithm can be modified properly, e.g. by using the iterative array model, that can be implemented very simply.

In this case the loops of the dataflow model are handled similarly to sequential circuits of the logic gates: the dataflow network that consists loops of channels can is replaced by a "unfolded" dataflow network that does not contain loops. An example is shown in Figure 4.1. The two networks are equivalent in the following sense. Each cell of the array is identical to the one given in the upper part of the figure. If an input sequence $a(0), a(1), \ldots, a(k)$ is applied to N in its initial state $s(0)$, and it produces the output sequences $b(1), b(2), \ldots, b(k)$ and $c(1), c(2), \ldots, c(k)$ and state sequence $s(0), s(1), \ldots, s(k)$, then the iterative array will generate the outputs $b(i+1)$ and $c(i+1)$ from cell i, in response to the inputs $a(i)$ and $b(i)$ to cell i.

4.3.2 Adaptation of PODEM

Because of the differences between dataflow and gate-level models, the following problems have to be solved during adaptation:

- Due to the multi-valued fault model the number of possible combinations of the test vector and of the error pairs is larger in the dataflow model. The algorithm has to cope with this increased complexity.

- According to the non-deterministic behavior of a component, multiple firings are enabled at a time in a dataflow node. It increases the complexity of the imply procedure and the number of backtrackings. To get correct result a list based fault propagation is necessary (more details see later).

- Firings describe the predecessor and successor states of a computation. Checking has to ensure, that the consecutive states of the node in the iterative array model correspond to these states.

- Dataflow nodes can produce output without mapping all of the inputs; therefore, the D-frontier contains nodes to the inputs of which errors are propagated instead of containing nodes whose outputs are reached during error propagation.

- Checking has to ensure that the constraints imposed by the global testing requirements are fulfilled, e.g. safe testing, when the faults propagated during testing of a component must not cause an error of other components.

List based fault propagation

In the adapted PODEM algorithm lists of token pairs are processed instead of simple token pairs. This is necessary in order to get correct results of test generation. Please note that it differs from list based fault simulation where a given test vector is simulated for all possible faults of the system.

Similarly to composit justification [Szi79b, Szi79a] a token pair denotes a pair of tokens, one of which describes the error free operation of the system, while the other describes the response of the erroneous system. It the tokens are different the token pair is called error pair. I this case the erroneous behavior of the system can be distinguished from the error free one. Thus the actual aim of test generation is to find an input vector, such that if it is applied to the inputs of the system an error pair appears on the output(s) of the system.

Consider a simple dataflow node with two firings that are concurrently enabled, but produce different outputs; one of them propagates the input error pair, while the other does not. In the operation the two firings are executed randomly, thus the node does not propagate the error pair in each case. Therefore, a single test vector results multiple possible test results. Accordingly an input mapping is only then a valid test vector, if all of the multiple output mappings contain some error pairs. Since this problem can not be decided locally, i.e. decision can only be done when the tokens reached the outputs, the problem of list based processing has to be solved. In the above case the input of the node is a single error pair, while its output is a list of the two possible outputs.

The above exact method delivers the test set of the system. In case if non-determinism in the model is due to the lacking implementation details, e.g. modeling of a HW without knowledge of the corresponding SW, this method can not be used, since in many cases test can not found. In such cases an approximate method can be used in which error propagation is based on error pairs and not on list of error pairs. Of course the computed test is only a potential test, and in a post-evaluation phase the feasibility of the test has to be checked.

As an example consider a system bus of a computer where the CPU executes a write

```
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 token values)
8:    begin
9:      Imply(j,v)
10:     if (Podem()=SUCCESS) then return SUCCESS
11:   end
12:   return FAILURE
13: end
```

Figure 4.2: The PODEM algorithm

operation, but the destination address is not known. In this case the bus arbiter selects randomly one peripheral device and the computed test may pass if during operation a different peripheral device is addressed. Therefore, in the post evaluation-phase a SW has to be written that makes it sure that under all circumstances the given peripheral device is selected by the bus arbiter.

During evaluation of the complex example of this work, i.e. the testability analysis of the MEMSY system this approximate method and post-processing was used.

4.3.3 Description of the Algorithm

Test generation is started with initialization of the channels, when the value `ND` (not defined) is assigned to each channel; with resetting the nodes to their initial state; and with injecting faults into the faulty node. After that the Podem() procedure is called.

The Podem() procedure

The outline of the procedure is given in Figure 4.2. In each step when Podem() is executed some checking is made, 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 procedure are:

Step 3: Check the stop criterion: each time Podem() is called it has to be checked, whether the goal of test generation is reached, i.e. a list, containing error pairs is propagated to a PO.

Step 4: Check whether test generation is still possible: if the goal of test generation is not reached, it has to be ensured, that test generation can be carried on meaningfully. Usually, this is done by checking that:

- the target fault can not be activated, since a different token value has been propagated to the output of the erroneous component,
- no error propagation can be done, since the D-frontier is empty,
- conditions that are imposed by the implementation of the algorithm are true (e.g. the maximum number of elements in the iterative array model is reached, the maximum number of tokens in a channel is reached, etc.)

If one of the above checking gives positive results, Podem() has to be stopped and a backtracking has to be done.

Step 5: Select an objective for error propagation: procedure Objective() is called in order to select an objective, i.e. a channel of the dataflow network. The goal of subsequent steps is then to propagate a list containing error pairs to the selected channel.

Step 6: Select a PI: in PODEM objectives are reached by setting the PIs, therefore a PI has to be selected, that is in connection with the given channel. Backtrace() is called to trace back from the channel to one of the PIs of the dataflow network. Backtracing is based exclusively on the structure of the network, the function of the components is not considered.

Step 7: Due to the multi-valued fault model subsequent steps have to be repeated for all possible token value.

Step 9: Imply value v on PI j: a token with value v is inserted into PI j, and implication is done by propagating the token toward the POs by executing all enabled firings of the dataflow network.

Step 10: Call Podem() to carry on test generation: when all possible implications are done, Podem() is called again to select a new objective and to try to generate a test by justifying this additional objective.

Step 12: Test generation is not successful: if test generation failed for all possible token values backtracking has to be done.

The Objective() procedure

The aim of PODEM is reached by dividing the test generation problem into subgoals of error propagation to a selected channel. Procedure Objective() (see Figure 4.3) is responsible to select the next subgoal for PODEM. The activities of the procedure are:

Step 3,4: Select a node: a node has to be selected that has still unassigned inputs, i.e. inputs with value `ND`. First the inputs of the erroneous node are checked. If all of its inputs are assigned, then a node from the D-frontier is taken.

```
1:  Objective() /* fault is n=f */
2:  begin
3:    if (any input 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 4.3: The Objective() Procedure

```
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 4.4: The Backtrace() Procedure

Step 5: Select a channel: one still unconnected input channel of the selected node is chosen randomly.

Step 6: Return the identifier of the selected channel.

The Backtrace() procedure

Based on the structure of the network and on the behavior of single nodes, the Backtrace() procedure (see Figure 4.4) is responsible for finding the PIs, with adjustment of which an error pair can be propagated to the channel that is selected by Objective(). The activities of the procedure are:

Step 3: Proceed as long as the channel is not a primary input.

Step 5–6: An input j has to be assigned to the output k of the component n. The decision of which input to be assigned to the output is based on local testability measures, i.e. the input is selected from which an error pair is propagated most probably. This is definitely a weak point of PODEM, since in PODEM value assignment is done only during forward propagation. Therefore, the more distant is n from the objective node, the less related will be the decision to actual channel contents.

Step 8: Return the identifier of the PI that was found. The path traced back by the procedure denotes the implication path from the PI to the given objective.

The effectivity of PODEM can significantly be increased by the appropriate selection of local testability measures, upon which the decision in Step 5 of Backtrace() is done. Unfortunately the testability measures used for logic-gate level circuits can not be generalized for DFNs, since they were characteristic for the properties of logic-gate circuits. In the simplest case a random decision can be used that, although works without problems, is nothing else than a combinatorial experimenting of all possible cases. The suggested method should work more effective, at least during evaluation of our industrial size models the execution times were withing the range of acceptability. To elaborate the most effective measures lies outside the scope of this work.

4.3.4 Possible Simplification

In sequential logic the test will place test sequences on the inputs of the system instead of single test values. The iterative array model can used to help to generate each value of the sequence similarly to the case of combinational logic. Previously we showed that this method could be used for dataflow models too.

However; if the user is not interested in the state of the different nodes of the network in each execution step of the test, but only in their initial and final states, the usage of the iterative array model can be neglected. This is due to the fact, that the dataflow nodes are connected by channels with depth, and to the functioning they do not need the value on the inputs continuously. It means that the same instances of channels and nodes can be used in subsequent steps of test generation.

In this case the algorithm will be the same as presented in Figure 4.2 only the data structures will become different. The functioning of the algorithm for both with and without the iterative array model is explained in the next section.

4.4 Complexity Issues

In [Fuj86] the original algorithm [Goe81], is proved to be NP-complete, since in worst case it tends to exhaustive search. Since the adapted algorithm has identical outline (control structure) to that of the original one, PODEM for the dataflow networks is expected to be the at least the same complexity. The number of choices during the search is additionally increased due to the multi-valued token model and uncertainty modeling. Similarly to the original algorithm, our algorithm will always find a solution for the problem, as long as a solution is given. This is again due to the exhaustive search that is done in worst case. Finally if the algorithm delivers a result, it is a solution for the problem, since the algorithm is based on a provably correct implication.

4.5 Test Generation for the Example

To enlighten the previously described algorithm the example of the candy automaton is considered. The steps of test generation are presented in detail for the erroneous state `int` of the controller. The test generation procedure is shown in Figure 4.5. Vertical partitioning of the figure corresponds to the actual steps of test generation:

Step 0: Initialization of the network. State of the controller is set to `ok/int`, while the state of coin_in/out, select, and candies_out is set to `ok`. The value `ND` is assigned to each channel and the D-frontier is emptied. The PODEM algorithm can be started.

Step 1: First call of Podem(). Error propagation can be done, since no error pair reached the outputs (error_at_po() returns 0) and no value conflict is detected at the outputs of the controller (test_not_possible() returns 0). The inputs of controller, `from_coin_in/out` and `from_select`, are still unassigned. Objective() selects the channel `from_coin_in/out` randomly to be the objective of line justification. Backtrace() finds the PI `coin_in` to be in connection with `from_coin_in/out`. Implication is started by assigning the value `ok` to `coin_in`. After executing firing r1 of coin_in/out the value `ok` is propagated to channel `from_coin_in/out`. Error propagation terminates at this step, since there are no more enabled firings. The controller is put into the D-frontier.

Step 2: Second call of Podem(). The POs are still unassigned (error_at_po() returns 0) and no value conflict is detected at the outputs of the controller (procedure test_not_possible() returns 0), thus error propagation can be carried on. The controller still has an unassigned input channel. Objective() selects `from_select` to be the objective of line justification. Note that the controller was not selected by Objective() because it is in the D-frontier. Procedure Backtrace() selects the PI `select_candy` that is in connection with the channel `from_select`. Implication is started by setting the value of `select_candy` to `ok`. After executing firing r1 of select the value `ok` is assigned to `from_select`. Now firings r1 and r21 of the controller can fire. After execution of the firings, the error pair `ok/inc` appears on channel `to_candies_out`. Firings r1 and r2 of candies_out deliver a token `ok` on `from_candies_out` and an error pair `ok/inc` on `out`. The second firing of the controller (firings r17 and r37) puts an error pair `ok/inc` into the channel `to_coin_in/out`. The controller already fired once, therefore the second firing belongs to the second block of the iterative array model. After execution of the firings r5 and r6 of coin_in/out, the error pair `ok/inc` is propagated to `change`. It is the second firing of coin_in/out, thus is is presented in block 1 of the iterative array model. Error propagation is terminated for lack of enabled firings. The D-frontier becomes empty, since both the inputs and the outputs of the controller are assigned.

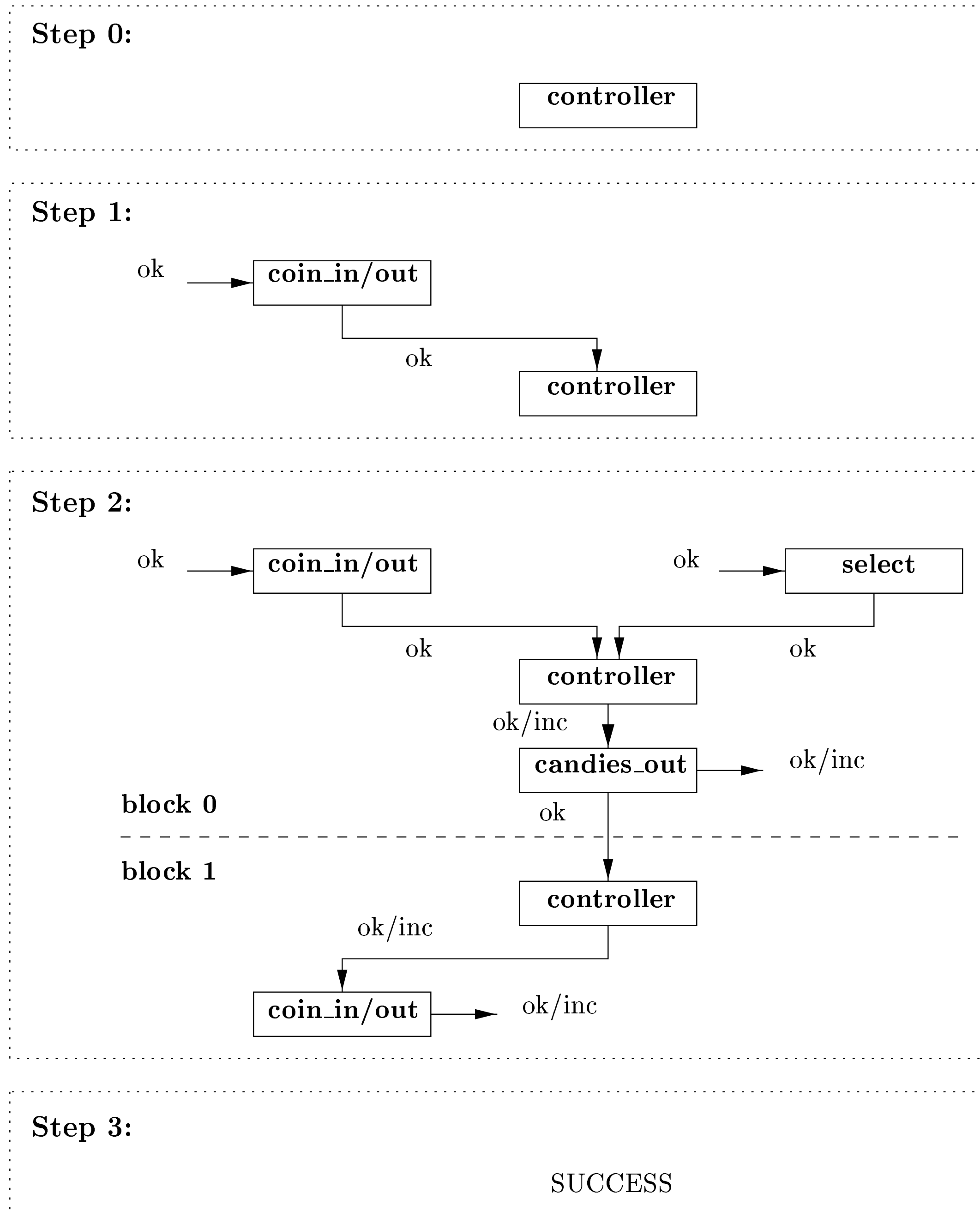


Figure 4.5: PODEM for the Candy Automaton

Step 3: The last call of Podem(). The error pair `ok/inc` reached the outputs `change` and `out`; therefore, error_at_po() returns 1 and Podem() terminates the recursion with SUCCESS.

The generated test pattern maps `ok` on both `coin_in` and `select_candy`. The dataflow model of the test can be seen in Step 2 of Figure 4.5. Note that the test is free of loops. If the test is executed in an error-free system `ok` appears on both `out` and `change`. If the components coin, candy, and select are error-free and the controller is in state `inc`, the execution of the test results `inc` on both outputs of the system.

If we do not use the iterative array model, the loop between `controller` and `candies_out` will not be cut. Therefore; the state of `controller` in the first block will be overwritten by its state in the second block, and it can not be read out.

4.6 Contribution

In this chapter I gave an algorithm for test generation for the suggested dataflow models by adapting logic gate-level automatic test pattern generation algorithms. These become able to handle the multi-valued fault model and the uncertainty that characterize the non-deterministic dataflow formalism.

- I gave a method to adapt the efficient logic gate-level test generation algorithms by solving the problem of the multi-valued fault model and uncertainties of the model. The multi-valued token set is handled similarly then multi-valued logic in case of logic gate-level algorithms: during token propagation the tokens that describe the fault–free and the erroneous system are propagated together in form of a value pair. To cope with uncertainty I use list based propagation: all value pairs that can possibly sent by a dataflow node are kept together in a list. Therefore, the list contains the tree of possible execution paths of the dataflow network. Test generation is successful when on all leaves of the tree an error pair reaches the output.

- I carried out the adaptation for a given logic gate-level test generation algorithm and showed its functioning by experiments. The selected algorithm is the well known PODEM that was shown to be efficient for circuits containing a large number of fault- detecting and correcting circuits. Another advantage of PODEM is that it is relatively simple. It is based on a direct search: it tries to propagate faults from the fault site towards the outputs by manipulating the inputs of the system, therefore backpropagation can be neglected. The adaptation gave an experimental but fully functioning algorithm for test generation of the dataflow model.

Chapter 5

Testability Analysis

The aim of testability analysis was always to provide quantitative measures about the difficulty of testing a given system. By identifying the problem places, i.e. places with poor testability measures, testability analysis done in early phases of the design is the basic issue of design for testability.

In the era of LSI digital circuits testability was considered to estimate the difficulty of generating a test for a given fault of a circuit element, e.g. stuck-at-1 fault at the input of a NAND gate [AM82, Gol79]. The algorithms assumed that testability is an inherent property of the circuit based solely upon the circuit structure. This assumption allowed to estimate the circuit's testability before test generation was started. Therefore the basic requirement of testability analysis was that it should be computationally simpler than test generation. Figure 5.1 shows the process of this type of testability analysis. Testability in this context is defined by the testability measures, the six most important of which are: combinational zero controllability (CC0), combinational one controllability (CC1), combinational observability (CO), sequential zero controllability (SC0), sequential one controllability (SC1), sequential observability (SO).

As digital circuits and systems became more complex, test generation was shifted from gate-level to higher levels. Testability analysis followed this trend, but in case of

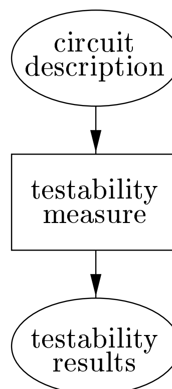


Figure 5.1: Testability Analysis Based on System Structure

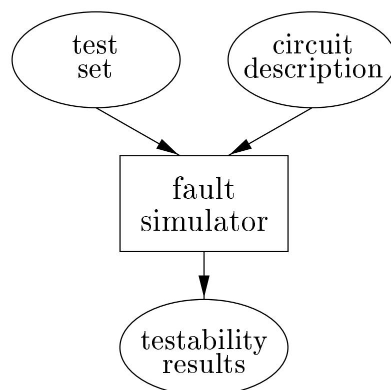


Figure 5.2: Testability Analysis Based on the Test Set of the System

high-level models that contain components with complex behavior, testability can not be measured solely based upon the structure without taking behavior into consideration [HG87, dPMCKS89, SS91c, SS94]. Therefore, testability analysis can not be executed without the knowledge of the test set, that is generated by a highly sophisticated ATPG. The typical process of this type of testability analysis is shown in Figure 5.2. In this context testability is defined as: "... a design characteristic which allows the state (operable, inoperable, or degraded) of an item to be determined and the isolation of faults within the item to be performed in a timely and efficient manner." [Mil85]. Typical testability measures are detectability and diagnosability.

In our approach testability analysis is less complex than the *final* test generation, i.e. test generation that produces the final test set of the system after resource allocation. Therefore, testability analysis can be done before this sophisticated test generation process is carried out. Indeed testability analysis will deliver guiding attributes for this test generation process. On the other hand the behavior of system components can be very complex, thus approaches that rely purely on the system structure can not be applied. Therefore, we have to use the second approach for testability analysis. The task of test generation is taken over by the high-level version of PODEM, that delivers the superset of system tests. Fault simulation has to be followed by a step that further processes simulation results, since from these results the testability measures and the diagnostic strategy can not be read out directly.

For testability analysis and diagnostic design, we use integrated diagnostics, that is defined as: "... a structured process which maximizes the effectiveness of diagnostics by integrating the individual diagnostic elements of testability, automatic testing, manual testing, training, maintenance aiding, and technical information." [Kei90].

5.1 On Integrated Diagnostics

The main advantage of integrated diagnostics over traditional testability analysis methods is that it covers the whole life-cycle of the product and integrate different aspects

of maintainability, such as: minimized mean time to isolate system faults, minimized mean time to repair, training, documentation [Cig89, JRF91, Ofs91].

One of the best elaborated approaches for integrated diagnostics is presented by Sheppard et al. in [SS94]. (The book was preceded by a series of articles [SS92c, SS91c, SS93a, SS93b, SS91a, SS91d, SS92b, SS92a, SS91b].) This approach is based on the conclusion-test and test-test dependency relations, where conclusion is the isolation of a fault, in our case fault of a component, and test is any information source relevant to the diagnostic problem. Dependency relationships among tests and conclusions are described in the form of a dependency graph.

In the dependency graph tests and conclusions are represented by nodes (graphically tests are denoted by circles and 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. In general 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:

- isolation level (the ratio of diagnosable faults)
- non-detection (fault coverage, the ratio of detectable faults)
- test leverage (robustness of the test set)
- overtesting (the ratio of uniquely diagnosable faults to the number of tests is relatively high)
- undertesting (the ratio of uniquely diagnosable faults to the 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 produce identical syndromes as a different fault)

The main drawback of this approach is, that it does not use the physical model of the system at all. Therefore only ad hoc methods exist for the assembly of the dependency graph.

5.2 Integrated Diagnostics and the Dataflow Approach

A natural idea is to try to combine integrated diagnostics and the presented dataflow approach, in order to support testability analysis and diagnostic design. For this reason the input model of integrated diagnostics has to be extracted from the dataflow model and the test set of the system. It was shown in [CGPT94], that all dependency relations that are necessary to build the dependency graph of the system can be obtained by means of concurrent fault simulation. The algorithm of concurrent fault simulation is very similar to discrete event simulation algorithms [ABF90, GA85, SS87, WA90], but it lies outside the scope of this work. Therefore, the algorithm is not presented here.

5.2.1 Extracting the Input Model of Integrated Diagnostics

During concurrent fault simulation the tests of the system are evaluated. In a single run one of the tests is taken, and test results are computed for all possible fault hypotheses of the fault model, e.g. all single faults of the system. Simulation is done by propagating the tokens from the primary inputs of the system to its primary outputs. Tokens that are put onto the primary inputs correspond to the input mapping of the test vector, while tokens that appear on the primary outputs of the system denote the results. This way test results are assigned to each fault hypothesis of the fault model.

In integrated diagnostics the tests correspond to the tests of the system, while the conclusions are associated with the fault hypotheses. The fault-free case is denoted by conclusion C_0 and test results of the fault-free system are used as reference. Consider a test T_1 and a conclusion C_1: if the test result of T_1 in case of the given fault differ from the reference result of T_1 then the test depends on the conclusion and the dependency $C_1 \mapsto T_1$ can be added to the dependency graph. Otherwise no dependency relation exists between C_1 and T_1.

5.2.2 Dealing with Uncertainty

In order to cope with the non-determinism of the model a probabilistic approach is presented in [SS91b]. It incorporates methods from fuzzy logic, Dempster-Shafer evidential reasoning, and neural networks in order to be able to reason with incomplete and uncertain data. The main problem with this method is that is was developed for hybrid, dominantly analog, systems. Therefore, it can not be used for testability analysis and diagnostic design of digital systems, where one has to cope with the large structural complexity. Moreover, the approach, similarly to stochastic processes, relies on known probabilistic values of various model parameters. This hinders the usage of the method in high-level modeling. Since our goal is to try to avoid using stochastic processes when dealing with non-determinism, uncertainty computation has to be solved differently. Therefore, in accordance with the modeling approach, the notion of strong and weak dependency is introduced.

Definition 5.1 *A dependency relation between test* T_1 *and conclusion* C_1 *is called* strong dependency *iff* T_1 *always fails when* C_1 *fails.*

Definition 5.2 *A dependency relation between test* T_1 *and conclusion* C_1 *is called* weak dependency *iff* T_1 *fails only with respect of the data values when* C_1 *fails.*

Consider the candy automaton example, to show the difference between strong and weak dependency: let C_1 denote the erroneous state `rec` of coin_in/out, T_1 the test generated in Chapter 4, and T_{1a} and T_{1b} the results on outputs `out` and `change` respectively. Fault simulation delivers T_{1a} = `inc` and T_{1b} = `x`. In this case $C_1 \mapsto T_{1a}$ is a strong dependency, since `inc` is always interpreted as incorrect, whereas $C_1 \mapsto T_{1b}$ is a weak dependency, since `x` can be interpreted either as correct or incorrect depending on the actual, as yet unknown value of data.

In the analysis this twofold interpretation of dependency causes that integrated diagnostics has to be executed twice: once when the input is the dependency graph containing only the strong dependencies and once when the input is the dependency graph that contains edges for both strong and weak dependency relations. The results of analysis are referred to as *pessimistic* in the case when only strong dependencies are considered and *optimistic* when both strong and weak dependencies are taken into account. From the point of view of dependability measures the pessimistic case means the worst case, i.e. the lower bound of the measures, and the optimistic case means the best case, i.e. the upper bound of the measures. By estimating the lower and upper bound of the measures an interval, the so called uncertainty interval, is given. The final value of the measures (the one characterizing the fully specified system) is within this interval.

5.3 Testability Measures after Refinement

The main cornerstone of the above statement is, whether the final measures really fall into the interval defined by the results of higher level analysis. For this reason it has to be shown that during successive steps of model refinement the lower- and upper bounds of the interval converge to each other thus the size of the interval decreases. To fulfill this requirement it is necessary (and as it turns out sufficient too) that if a fault was not detectable before refinement, the corresponding faults remain not detectable after refinement and if a fault was detectable before refinement, the corresponding faults are detectable after refinement. By proving this it can be assured, that if the rules of refinement in subsequent steps of modeling are not violated, the interval of testability measures monotonically decreases and the measures converge to their final value. However to prove this behavior some formalization is necessary.

Definition 5.3 *The* computation of a dataflow network *is a relation* $CP : IM^\star \times S \mapsto OM^\star$*, where* $M = \bigcup_{\forall n \in N} M_n$ *and*

IM - *input mapping,* $IM = M^i$, $i = \|I\|$
OM - *output mapping,* $OM = (M^*)^o$, $o = \|O\|$

Informally the computation of the dataflow network can also be described as a finite or infinite firing sequence that starts from the initial state s_0.

Definition 5.4 *The* fault set F *contains all possible fault hypotheses of the given fault model. Let* $f_0 \in F$ *denote the fault-free system.*

According to the definition, F defines the set of faults, for which tests have to be generated, and in presence of which the testability measures of the system have to be evaluated. For example in case of a single-fault model, the elements of F are all possible single-faults of the system. If each component has only one fault, the number of hypotheses in F is equal to the number of components plus one.

Definition 5.5 *A* test t *for fault* $f \in F$ *is defined by* tv *the test vector (that is indeed a sequence of vectors).* $tv \in IM^\star$.

T defines the test set, that consists the tests, which are generated for the different faults of the system. Cardinality of F and T however is not equal, it is possible, that more than one test is generated for a given fault. $t \in T$ is a test for fault f iff the test results tr_0 and tr_f differ, where $cp_0 = ((tv, s_0), tr_0)$ and $cp_f = ((tv, s), tr_f)$ are the computations of the dataflow network when test vector tv is applied to the primary inputs in the fault-free state as well as in the faulty state of the system. tr_0 and tr_f differ iff $|tr_0| = |tr_f|$ and $\exists a \in tr_0$, $b \in tr_f$ such that a and b differ. Since $|a| = |b|$, a and b differ iff $\exists c \in a$, $d \in b$ such that $c \neq d$, where $a \in (M^*)^o$, $b \in (M^*)^o$, $c \in M^*$, $d \in M^*$. In the above case cp_0 and cp_f are two possible executions of test t with respect to the states of DFN.

Definition 5.6 *Test* t *is called a* certain/uncertain *test for fault* f, *iff* tr_0 *and* tr_f *differ not only/only by uncertain tokens, i.e. by* x *tokens.*

Test certainty has to be introduced according to the non-deterministic modeling approach. This is in accordance with the informal description of strong and weak dependencies.

Definition 5.7 *A fault* f *is* not detectable/uncertainly detectable/certainly detectable *if no test exists/an uncertain test exists/a certain test exists for it. The sets of not detectable faults/uncertainly detectable faults/certainly detectable faults are denoted by* F_n, F_u, F_c *respectively, where* $F_n \cap F_u = F_n \cap F_c = F_u \cap F_c = \emptyset$ *and* $F_n \cup F_u \cup F_c = F$.

The introduction of the notion of fault detection is necessary since calculation of the testability measures are based on the cardinality of the set of detectable/not detectable

faults, or with the terms of integrated diagnostics on the existence or absence of dependencies. If a fault is not detectable, it is not an element of any dependency relation. If the fault is uncertainly detectable it is element of a weak dependency relation. Finally if a fault is certainly detectable, it is element of a strong dependency relation.

Now we can formulate and prove the theorem, which states that after refinement if the cardinality of the fault set remains the same the number of detectable faults under pessimistic assumptions will be larger then before refinement, while the number of detectable faults under optimistic assumptions will be smaller then before refinement.

Theorem 5.1 *Let $L1$ and $L2$ be two different levels of modeling, DFN^1 and DFN^2 the two corresponding dataflow networks, and $\mathcal{R}$ the transformation from L_1 to L_2. If $\mathcal{R}$ is a refinement and $\|F^2\| = \|F^1\|$, then*

$$\|F_c^2\| \geq \|F_c^1\| \text{ and } \|F_n^2\| \geq \|F_n^1\|$$

Proof The theorem will be proven in three steps: (1.) It will be shown, that if f^1 is certainly detectable, then $\forall f^2$ is also certainly detectable, where $f^2 \in \mathcal{R}(f^1)$. (2.) If f^1 is not detectable, then $\forall f^2$ is also not detectable, where $f^2 \in \mathcal{R}(f^1)$. (3.) If f^1 is uncertainly detectable, then $\forall f^2$ is either not detectable, or certainly detectable, or uncertainly detectable, where $f^2 \in \mathcal{R}(f^1)$. If these three steps are proved, the statement of the theorem is proved.

Step 1: According to Definition 5.7 if f^1 is certainly detectable then t^1 a certain test exists. Definition 5.6 says that in this case DFN^1 will deliver some non-x tokens, i.e. some certain tokens. The definitions of refinement state, that if a firing rule r^1 delivers a non-x token, then $\forall r^2 \in \mathcal{R}(r^1)$ also delivers a non x token. (Elements from the partition of certain tokens will not be moved to the partition of uncertain tokens.) By exploiting the composition of nodes, it can be proven by induction, that if DFN^1 produces some non x tokens, than DFN^2 also produces non x tokens. Thus $\forall t^2 \in \mathcal{R}(t^1)$ is a certain test for f^2; therefore, f^2 is certainly detectable.

Step2: The detailed proof is omitted since it can be done similarly as in Step 1, except that here we have to argue that no test exists for f^1, i.e. the test results for f_0^1 and f^1 do not differ.

Step3: If f^1 is uncertainly detectable, then the test results for f_0^1 and f^1 differ only by x tokens (see Definition 5.7 and Definition 5.6). According to the definitions of refinement, if firing rule r^1 delivers x tokens, then $r^2 \in \mathcal{R}(r^1)$ can deliver both x and non x tokens. Again using the composition of nodes it can be shown by induction, that when x tokens in r^1 are refined to non x tokens in r^2, f^2 gets either certainly detectable (test results differ by not only x tokens), or not detectable (test results do not differ). When x tokens in r^1 remain x tokens in r^2, f^2 remains uncertainly detectable. □

The restriction that the cardinality of the fault set must remain the same during refinement is important from the point of view of the theorem. If this condition does not hold, then it is insufficient to reason about the cardinality of F_n, F_u, and F_c. Instead,

a weighted sum of the faults belonging to these sets has to be computed, and we can reason only about the convergence of the measures regarding the original model, i.e. the one before refinement.

The reason for this is that the suggested method for testability analysis computes the so-called combinatorial testability measures. It has to be done in cases when no a priori statistical information about fault probabilities is available, e.g. the probabilities of fault detection can not be computed as the testability evaluation is typically performed prior to the resource allocation phase. This is the case in our modeling approach, which does not use fault probabilities. Note, that every fault coverage based testability measure in the literature still assumes implicitly the same premise of equiprobable faults. In our method testability measures are computed according to this assumption. For example if the system has 10 faulty states the conditional probability that the system is in one of them, supposed that it is faulty, 10%.

After model refinement not only the uncertainty of the model can decrease, but it is also possible that faulty states of the dataflow nodes are refined. Let us suppose a system that consists a single node which has three states *ok*, fty_1, and fty_2 for which $F_n = \{fty_1\}$, $F_u = \emptyset$, and $F_c = \{fty_2\}$. According to the assumption of testability analysis both fty_1 and fty_2 can occur with the same 50% probability. Detectability of the system in this case is 50%.

Now let us further suppose that the system is refined into a node which has now four states *ok*, fty_{1a}, fty_{1b}, and fty_2, which means that state fty_1 has been refined into the states fty_{1a} and fty_{1b} such that $F_n = \{fty_{1a}, fty_{1b}\}$, $F_u = \emptyset$, and $F_c = \{fty_2\}$. Now testability analysis has to be executed and since there is no uncertainty inherent to the model, we expect a detectability of 50%. However classical testability analysis assuming equiprobable faults, i.e. $P(fty_{1a}) = 1/3$, $P(fty_{1b}) = 1/3$, and $P(fty_2) = 1/3$ results a detectability of 33%. It clearly contradicts the previous assumption that $P(fty_2) = 1/2$. Therefore, the fault probabilities have to be computed regarding the fault probabilities before model refinement, i.e. $P(fty_{1a}) = P(fty_{1b}) = P(fty_1)/2$.

According to the probabilities the weighted sum of the fault sets has to be used in computation of testability measures. In the example $\|F_n^1\|_w = 1$, $\|F_n^2\|_w = 1$, $\|F_d^1\|_w = 1$, and $\|F_d^2\|_w = 1$ that yields a detectability of 50% where $\| \, \|_w$ denotes the weighted cardinality of the sets with respect to the same sets before refinement. These computation will result the testability measures regarding the original model. If computation of testability measures is done in this way then our statement about convergence of the measures and decreasing uncertainty interval is true independently of the cardinality of fault sets.

5.4 Testability Analysis of the Example

The candy automaton example is used to explain the process of testability analysis. The test set of the system consists three tests, generation of one of which was shown in

	primary inputs	
test	coin_in	select_candy
T_1	ok	ok
T_2	inc	ok
T_3	ok	inc

Table 5.1: Test Vectors for the Candy Automaton

Chapter 4. The input mappings, i.e. the test vectors, of the tests are given in Table 5.1, while the interpretation of the test is the following:

T_1 We select a candy by typing the identifier of the candy correctly. Then we put in the correct amount of money in valid coins. This test was generated for the `int` error of the controller.

T_2 We select the candy correctly by typing a valid candy identifier. Then we put in some invalid coins, e.g. Hungarian Forints instead of Cents. The test was generated for the `int` error of the controller.

T_3 We type an invalid candy identifier, that leads to incorrect candy selection. Then we put in the right amount of money in valid coins. This test was generated for the `ctrl` error of candies_out.

In the next step fault simulation is done under the assumptions that only one component can be faulty at a time (single fault model), the erroneous state `stk` of candies_out is not allowed, and the faults of the components are permanent. The results of fault simulation are summarized in Table 5.2. The rows of the table show the test vector of the actual test (inputs), the current state of the system (system state), and the results of the test (outputs).

According to the notation of integrated diagnostics and to the fault hypotheses five possible conclusions can be considered: C_1 is coin_in/out=`rec`, C_2 is select=`cont`, C_3 is controller=`int`, C_4 is candies_out=`ctrl`, and finally the conclusion no fault is denoted by C_0. The primary outputs `out` and `change`, that are the actual information sources in integrated diagnostics, are referred to as T_{1a}, T_{2a}, T_{3a} and as T_{1b}, T_{2b}, T_{3b}.

The extracted dependency graph of the example is given in Figure 5.3, while the dependency matrix for the example is shown in Table 5.3. In the dependency matrix F denotes "strong" dependency and f denotes "weak" dependency. In the dependency graph solid lines present "strong" dependency, and dashed lines present "weak" dependency.

As an example for the extraction of the dependency graph consider the following case: each time the tokens `ok, ok` are assigned to the inputs and coin_in/out fails, test result T_{1a} will also fail as denoted by the `inc` result on `out`. In the dependency graph this strong dependency is represented by the solid line from C_1 to T_{1a}, while in

inputs		system state				outputs	
coin_in	select_candy	coin_in/out	select	controller	candies_out	out	change
ok	ok	ok	ok	ok	ok	ok	ok
ok	ok	rec	ok	ok	ok	inc	x
ok	ok	ok	cont	ok	ok	inc	ok
ok	ok	ok	ok	int	ok	inc	inc
ok	ok	ok	ok	ok	ctrl	inc	ok
inc	ok	ok	ok	ok	ok	inc	ok
inc	ok	rec	ok	ok	ok	inc	x
inc	ok	ok	cont	ok	ok	inc	ok
inc	ok	ok	ok	int	ok	x	inc
inc	ok	ok	ok	ok	ctrl	x	ok
ok	inc	ok	ok	ok	ok	inc	ok
ok	inc	rec	ok	ok	ok	inc	x
ok	inc	ok	cont	ok	ok	inc	ok
ok	inc	ok	ok	int	ok	x	inc
ok	inc	ok	ok	ok	ctrl	x	ok

Table 5.2: Results of Fault Simulation

	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 5.3: Dependency Matrix for the Example

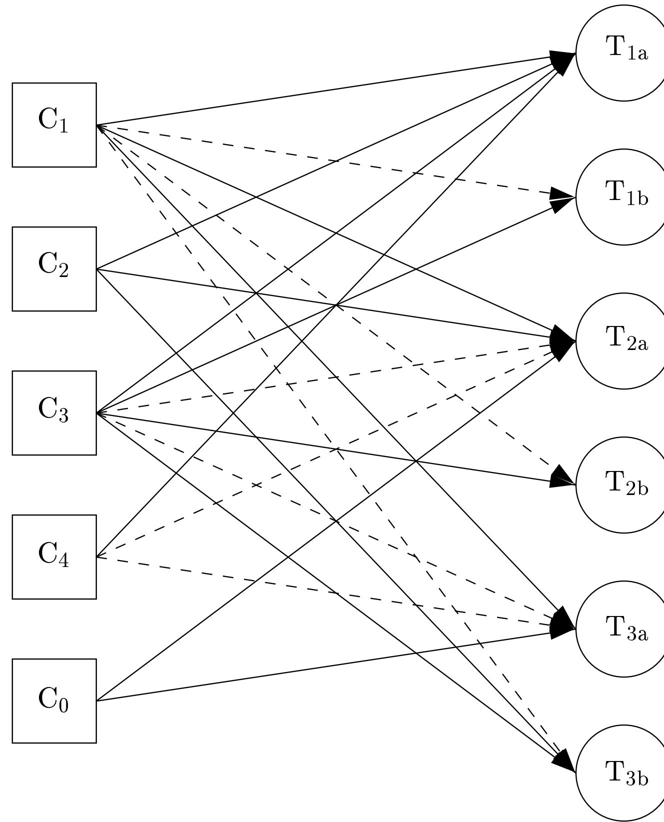


Figure 5.3: Dependency Graph for the Example

the dependency matrix by the element [7,1]=F. In the same case test result T_{1b} is data dependent (output `change` is `x`). This weak dependency is represented by the dashed line from C_1 to T_{1b} and by the matrix element [7,2]=f.

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

- It can be seen from the dependency matrix, that none of the tests produces the same test result in the error-free case and in the erroneous case. Thus fault coverage is 100%, all faults can be detected in both the pessimistic and in the optimistic case.

- Isolation level equals to the ratio of uniquely isolatable groups to all fault conclusions and denotes the ratio of diagnosable faults. In the pessimistic case conclusions can be divided into three groups according to the test results enlisted in the dependency matrix. Group1 is C_1 and C_3, group2 isC_2, group3 is C_4. Therefore, 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_2 and T_3 are redundant, since they deliver the same result in all cases. Hence, one of them can be left out during the system test.

5.5 Contribution

In this chapter I gave an approach for testability analysis and diagnostic design. It is based on integrated diagnostics whose inputs model is generated by concurrent fault simulation. I extended integrated diagnostics in order to cope with non-determinism. I proved that the uncertainty interval of the testability measures regarding the original model monotonously decreases if the rules of model refinement are not violated.

- I gave an approach to extract the input model of integrated diagnostics from the dataflow model. Concurrent fault simulation has to be executed in advance for each element of the fault set, in order to examine the effects of the faults on the tests. The results of the different simulation runs together builds the fault dictionary of the system. It consists the information whether a test depends on a given fault. These dependencies in form of a test-dependency matrix form the input model of integrated diagnostics that can now be executed.

- I extended the method of integrated diagnostic in order to deal with modeling and functional uncertainty. I introduced the notion of strong- and weak dependency. If a test always detects a fault, then the dependency is strong. If in some cases the test does not detect a fault, but in some other cases it does, then the dependency is weak. Integrated diagnostics has to be executed two time. Once by considering only strong dependencies and once by considering both strong- and weak dependencies. In the former case the pessimistic testability measures, while in the latter case the optimistic measures are computed. The final values of the measures lie within this uncertainty interval.

- I proved that the uncertainty interval of diagnostic measures regarding the original model decreases and the measures monotonously converge to their final value if and only if the rules of model refinement are kept. In order to prove it the faults are grouped into three categories: not detectable, uncertainly detectable, detectable. I proved that after refinement: 1) if a fault was not detectable it remains not detectable, 2) if a fault was detectable it remains detectable, 3) if a fault was uncertainly detectable it falls into one of the three categories. As a result the pessimistic testability measures regarding the original model monotonously increase, the optimistic measures monotonously decrease. Therefore the uncertainty interval monotonously decreases.

Chapter 6

An Application Study

The candy automaton presented in the previous chapters is a very small example, that is only useful to demonstrate the features of the framework but it has to be shown on an industrial size pilot application that the developed methods and algorithms are also useful in realistic applications. For this reason the model of a multiprocessor system, called MEMSY [CHG+94], is presented. The detail of modeling is the same as in [Cse97], but in this work due to space limitations only some selected parts of the system are shown that are characteristic for the application and for the approach.

6.1 MEMSY

MEMSY is a modular expandable multiprocessor system. The basic unit of the system is a single node, which is a multiprocessor, shared memory machine consisting of up to 4 processors (CPU), 8 memory management units and cache memories (CMMU), hard disc, LAN adapter, communication memory adapters, and additional support circuits. The CPUs and CMMUs can work either stand-alone or can be connected pairwise in master-checker (M/C) configuration.

Nodes are able to work stand-alone too, but five such nodes connected to one pyramid, build the base of expandability of MEMSY. Four of the five nodes are the workers and the fifth is a supervisor. In the fifth node a watchdog processor can be found, that helps the work of the supervisor node. Communication among nodes within and outside of pyramids happens by message passing via memory boxes (so called communication memory).

MEMSY is a system already developed and implemented system; therefore, modeling and evaluation aim at a post design analysis. This fact does not restrict the expressiveness of the example, but some considerations are necessary in the model construction phase. Since it is a general purpose system, modeling is done from the perspective of hardware. The neglected software aspects appear as random choice among multiple enabled firings in the model of software related components. For example the target of a CPU write, memory or peripheral device, can not be decided uniquely. It is explained in

more detail in the subsequent section of CPU modeling. This increased uncertainty does not affect the results of evaluation, it just increases the execution time of the evaluation.

6.1.1 Modeling

The system architecture allows multiple system configurations, depending on how the CPUs and CMMUs are used, i.e. in stand-alone or in master-checker mode. The evaluation aims at comparing the testability measures of the basic configurations. For this reason the following three detailed models are built:

MEMSY-I Within the simplest setup the processor card contains 1 CPU. The 8 CMMUs are split into 4 code and 4 data CMMUs, that work in stand-alone mode.

MEMSY-II Using this setup the processor card contains 2 CPUs that are connected in master-checker configuration. 4 CMMUs are assigned to each CPU. The 4 CMMUs of a CPU are split into 2 code and 2 data CMMU that work in stand-alone mode.

MEMSY-III Applying this configuration the processor card contains 2 CPUs that work in stand-alone mode. To each processor 4 CMMUs are assigned that work pairwise in master-checker configuration. One pair is used as a code CMMU and the other is as data CMMU.

The system is modeled at functional block level where data and control paths are separated and dataflow nodes represent hardware components like: CPU, CMMU, hard disc, hard disc controller, LAN controller, watchdog, terminal, etc. Due to space limitation only one of the above models, MEMSY-I, is explained in this work.

Fault model

The fault model is very simple, only the faults in the data processing and propagation parts are considered. Therefore, tokens are classified similarly to the candy automaton example according to the correctness of the data value that is represented by the token:

ok This token denotes that the data value is correct. For the sake of simplicity ok tokens are coded by 0 in the formal description.

inc This type of token denotes that the data value is incorrect as a result of some component fault. Inc tokens are denoted by 1 in the description of firing rules.

x This token is necessary to model uncertainty, i.e. when correctness of the data value can not be decided. These tokens are denoted by 2 in the formal notation.

In most of the components the error propagation function is supposed to correspond to the PMC fault invalidation model. Table 6.1 shows the basic cases of this type of error propagation.

input	state	output
ok	error-free	ok
ok	erroneous	x
inc	error-free	inc
inc	erroneous	x

Table 6.1: PMC-like Error Propagation Function of the Components

Description of MEMSY-I

The model of MEMSY-I is presented in Figure 6.1. The five main parts of the system are: processor card, main memory, VME subsystem, utility subsystem, and communication subsystem. The S-BUS arbiter (S-ARB) is depicted in the central part of the figure. It is responsible for arbitrating the system bus and this way connecting the different parts of the system. Each valid S-bus access is indicated by the arbiter via a run LED.

The upper part of the figure (above the arbiter component) shows the processor card. It contains the processor (CPU), the master-checker unit of the CPU (CPU-MC), the cache and memory management units (CMMU) and their master-checker units (CMMU-MC), a selector that selects one of the 8 CMMUs (CMMU-SEL8), and an address-data demultiplexer (AD-DMPX). The demultiplexer is necessary since on the processor bus (P-BUS) address and data are multiplexed but on the system bus (S-bus) they are not multiplexed. An additional component that is only necessary in the modeling is the CPU selector (CPU-SEL), that sends the signals of one of the CMMUs to the CPU.

The main memory of the system (MEM) is a normal dynamic RAM. It is possible to use error correcting memory as the main memory of the system, but the modeled configuration at the University of Erlangen contained normal memory.

The VME subsystem is connected to the S-bus by an S-bus VME bus converter (S-VME). To the VME bus are attached the SCSI disc (DISC) via a controller (SCSI), the watchdog processor (WDP) if the node is a supervisor node, the ethernet controller (LAN), and an interrupt controller (ITC). The interrupt controller receives the interrupts from neighboring nodes in case they placed a message into the communication memory.

The communication subsystem contains the communication memory controller (CMC), the communication memory (CM), and the coupling units (COUPL) that connect the multiple CMs to the controller by providing a simple fault-tolerant routing. The communication memory is a dual ported RAM.

The utility bus (U-bus) connects the components of the utility subsystem. They are the static RAM (SRAM), the NVRAM that contains information about system configuration, a CIO, a DUART, the interrupt controller (IT-CONV), and the reset logic (RESET). Although the master-checker interface (MCI) belongs to the processor card, it can be found in this lower part of the figure. The interrupt controller is connected to the interrupt controller for neighborhood communication, handles the interrupt signals

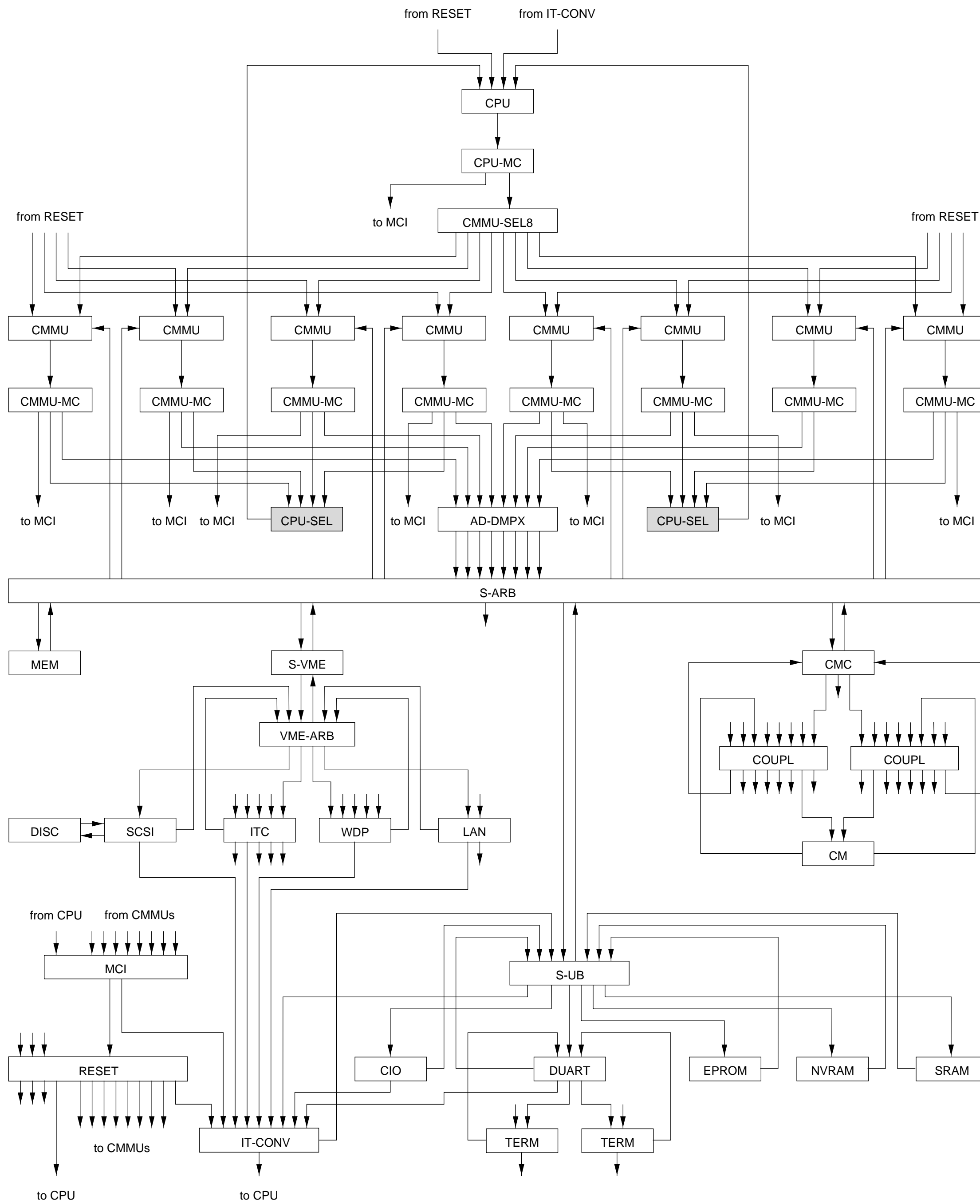


Figure 6.1: A MEMSY node with 1 CPU and 8 CMMUs

item:	basic	extended
components	29	29
nodes	46	46
channels	382	382
PIs	6/22	6/22
POs	6/22	6/22
states (Σ)	72	152
states	$\approx$ 6x10^6	$\approx$ 6x10^{16}
firing rules	678	6799
PN states	1132	7333
PN transitions	1356	13598

Table 6.2: Statistics of MEMSY-I

of the VME bus as well as of the utility subsystem, e.g. from the DUART.

6.1.2 Model Statistics

To get a feeling about the complexity of the model Table 6.2 shows some statistics about the size of the dataflow network. The first column contains the items of the network, while the second column gives the number of items in the basic, error-free system. Finally the third column contains the parameters of the extended system model.

The number of PIs and POs should be interpreted in the following way: x/y means that when the node is stand-alone the value x is valid, e.g. the inputs of the watchdog do not count in this case, otherwise value y is valid. The sum of the states of the components is given by state (Σ), whereas the state of the network, that is composed from the states of the components is denoted by states.

In the last two rows the parameters of the isomorphic Petri net are given. The Petri net is composed by using the DFN-PN transformation suggested in [Cse93, CBBS94]. Note that the number of the Petri net objects is really huge! A Petri net of this size is hardly editable in contrast to the size of the dataflow network of Figure 6.1.

Many of the parameters of the network are computed from the parameters of the individual nodes, whose parameters are given in Table 6.3. The table consists of 3 columns: the name of the node, the parameters of the error-free node, and the parameters of the erroneous node. Parameters of the nodes are the following: number of inputs, number of outputs, number of states, and the number of firing rules.

6.2 Model of the CPU

According to its Harvard architecture the CPU, a Motorola 88000, is connected to two different sets of CMMUs. One is the set of code CMMUs and the other is the set of data

component:	basic				extended			
AD-DMPX	32	24	1	32	32	24	2	160
CIO	2	2	2	3	2	2	3	9
CM	6	6	1	4	6	6	5	106
CMC	9	11	1	9	9	11	2	98
CMMU	6	12	8	22	6	12	24	168
CMMU-MC	12	7	2	8	12	7	4	368
CMMU-SEL8	6	12	1	24	6	12	2	80
COUPL	24	24	5	32	24	24	5	160
CPU	6	6	4	23	6	6	20	222
CPU-MC	6	4	2	6	6	4	4	44
CPU-SEL	8	4	1	8	8	4	1	16
DISC	3	2	1	2	3	2	5	53
DUART	4	4	1	5	4	4	3	39
EPROM	1	1	1	1	1	1	2	2
IT-CONV	10	3	1	10	10	3	2	24
ITC	6	6	1	9	6	6	2	18
LAN	3	4	2	5	3	4	5	23
MCI	10	2	11	20	10	2	12	30
MEM	3	3	1	2	3	3	5	53
NVRAM	2	1	1	2	2	1	3	12
RESET	4	14	1	4	4	14	2	8
S-ARB	37	38	10	132	37	38	10	660
S-UB	9	14	1	17	9	14	2	138
S-VME	6	6	1	7	6	6	2	54
SCSI	4	6	3	5	4	6	8	50
SRAM	2	1	1	2	2	1	3	12
TERM	2	2	1	2	2	2	2	12
VME-ARB	8	10	4	18	8	10	8	84
WDP	10	2	2	12	10	2	4	156

Table 6.3: Statistics of the Components

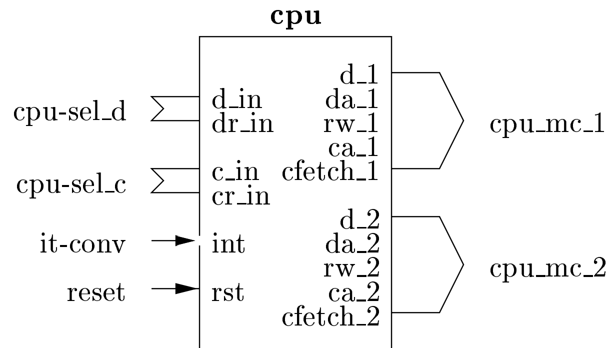


Figure 6.2: Dataflow Model of the CPU

ID	name	description
0	cpu_ok_end_fetch	end of a fetch cycle
1	cpu_ok_end_read	end of a read cycle
2	cpu_ok_end_write	end of a write or read-modify-write cycle
3	cpu_ok_end_read(ex,it)	end of an exception or IT handler address read

Table 6.4: States of the CPU (Basic Operation)

CMMUs. The CPU supports master-checker configuration; in modeling the CPU and its master-checker (MC) interface are separately modeled as two functional units. The node called CPU operates like a normal CPU without master-checker feature and the node called CPU-MC is responsible for the comparison of the signals of two CPUs. In this section only the modeling of the CPU is discussed. The dataflow model of the CPU is given in Figure 6.2, while the rather long, detailed presentation is left to Appendix D.

The inputs of the CPU are connected to the data and code CMMUs, to the reset logic (RESET), and to the interrupt controller (IT-CONV). The outputs of the CPU are connected to the MC units (CPU-MC). The inputs are (on the left side of the Figure from top to the bottom): data input, data status, code input, code status, interrupt signal, reset signal. The outputs are (on the right side of the Figure from top to the bottom): data, data address, read/write, code address, code fetch. According to the master-checker scheme the outputs are doubled and lead to the two MC units.

6.2.1 Basic Operation of the CPU

The CPU is supposed to be able to execute the following operations: code fetch, data read, data write, data read-modify-write, reset, interrupt handling, and exception handling. ALU functions are only implicitly modeled, they are hidden between data read and data write cycles. The FSM model that describes these operations is presented in Figure 6.3. The identifier and the description of the states are given in Table 6.4. The firing rules that describe the above operations are copied here from Appendix D.

In the simplest operation, when a code fetch cycle is finished, the CPU starts either

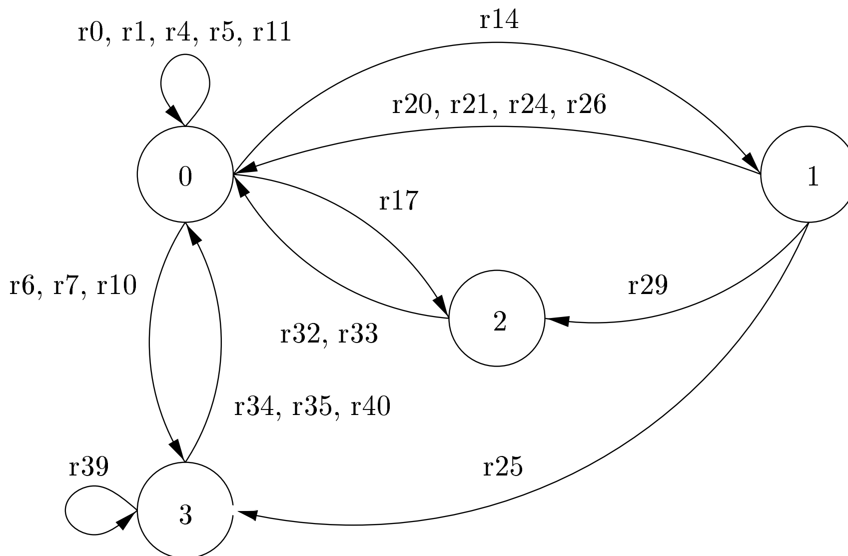


Figure 6.3: FSM Model of the CPU (Basic Operation)

a new fetch cycle, or a read cycle, or a write cycle, or a read-modify-write cycle. (The latter one is described by consecutive read and write cycles, without code fetch between them.) After finishing the read, write, or read-modify-write cycle the CPU carries on its operation by fetching the code of the next instruction.

A somewhat more complex operation is characterized by handling the reset, interrupt, and exception signals. In case of a reset signal the CPU starts a code fetch cycle. Reset can be risen in any state of the CPU and will be processed immediately. Interrupt can be signaled in every state of the CPU, but processing is delayed until finishing the current fetch, read, write, or read-modify-write cycle. Processing of the interrupt starts with a read cycle, that reads the address of the interrupt handler, and is carried on by fetching the first instruction of the interrupt handler routine. In our model the only exception is the erroneous memory or IO read that is signaled by the CMMUs (transaction fault). In this case the address of the exception handler is read immediately by a read cycle, and then the first instruction of the exception handler routine is fetched.

code fetch

Firing r11 denotes the simplest case of code fetch. The CPU is in its initial state (that is the one after a previous code fetch), receives the instruction code on `c_in`, remains in the same state and rises the `cfetch_1` and `cfetch_2` signals.

$$\mathtt{r11} = < 0; \mathtt{c_in} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 0 >$$

data read

The data read cycle is started after a fetch cycle (firing r14), in the initial state of the CPU. The lines `rw_1` and `rw_2` are driven, and the node goes into state 1 and waits for

the data to arrive. Firing r26 denotes the second phase of read, when data is received on `d_in`. As a result the CPU goes into the initial state and starts a code fetch.

$$\mathtt{r14} =< 0; \mathtt{c_in} = 0; 1; \mathtt{rw_1} = 0, \mathtt{rw_2} = 0; 0 >$$
$$\mathtt{r26} =< 1; \mathtt{d_in} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 0 >$$

data write

Data write is started after a successful code fetch. Firing r17 denotes the phase when the lines `rw_1` and `rw_2` are driven and the data is put on `d_1` and `d_2`. The write cycle is carried on by starting a code fetch, that is described by firing r33.

$$\mathtt{r17} =< 0; \mathtt{c_in} = 0; 2; \mathtt{d_1} = 0, \mathtt{d_2} = 0, \mathtt{rw_1} = 0, \mathtt{rw_2} = 0; 0 >$$
$$\mathtt{r33} =< 2; ; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 0 >$$

data read-modify-write

In case of read-modify-write, after reading the data from `d_in` the CPU executes the first phase of a write cycle instead of initiating a code fetch: lines `rw_1` and `rw_2` are driven and data is put on `d_1` and `d_2`. The second phase of write is denoted by firing r33.

$$\mathtt{r29} =< 1; \mathtt{d_in} = 0; 2; \mathtt{d_1} = 0, \mathtt{d_2} = 0, \mathtt{rw_1} = 0, \mathtt{rw_2} = 0; 0 >$$

reset

Reset has to be executable in any state of the CPU at any possible input mapping when the line `rst` is driven by another component. Upon reset the CPU starts a code fetch, and enters its initial state.

$$\mathtt{r0} =< 0; \mathtt{rst} = 0, \mathtt{c_in} = 0, \mathtt{cr_in} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 6 >$$
$$\mathtt{r1} =< 0; \mathtt{rst} = 0, \mathtt{c_in} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 5 >$$
$$\mathtt{r4} =< 0; \mathtt{rst} = 0, \mathtt{it} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 5 >$$
$$\mathtt{r5} =< 0; \mathtt{rst} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 4 >$$
$$\mathtt{r20} =< 1; \mathtt{rst} = 0, \mathtt{d_in} = 0, \mathtt{dr_in} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 4 >$$
$$\mathtt{r21} =< 1; \mathtt{rst} = 0, \mathtt{d_in} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 3 >$$
$$\mathtt{r24} =< 1; \mathtt{rst} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 2 >$$
$$\mathtt{r32} =< 2; \mathtt{rst} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 1 >$$
$$\mathtt{r34} =< 3; \mathtt{rst} = 0, \mathtt{d_in} = 0, \mathtt{dr_in} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 3 >$$
$$\mathtt{r35} =< 3; \mathtt{rst} = 0, \mathtt{d_in} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 2 >$$
$$\mathtt{r38} =< 3; \mathtt{rst} = 0; 0; \mathtt{cfetch_1} = 0, \mathtt{cfetch_2} = 0; 2 >$$

interrupt handling

The interrupt signal `it` is sampled only in the initial state of the CPU. The signal can arrive alone (firing r10) or with the instruction code together (firing r7). As a result

the CPU starts a read cycle by setting the outputs rw_1 and rw_2 and enters state 3, in order to read the interrupt vector.

$$\texttt{r7} =< 0; \texttt{it} = 0, \texttt{c_in} = 0; 3; \texttt{rw_1} = 0, \texttt{rw_2} = 0; 2 >$$
$$\texttt{r10} =< 0; \texttt{it} = 0; 3; \texttt{rw_1} = 0, \texttt{rw_2} = 0; 1 >$$

exception handling

In our case exception is the transaction fault signaled by the CMMU on the line cr_in (firing r6) or on dr_in (firing r25). As in case of interrupt the address of the handler routine has to be read by setting the signals rw_1 and rw_2. The CPU enters state 3 and waits for the data to arrive. Afterwards the first instruction of the handler routine is fetched.

$$\texttt{r6} =< 0; \texttt{c_in} = 0, \texttt{cr_in} = 0; 3; \texttt{rw_1} = 0, \texttt{rw_2} = 0; 3 >$$
$$\texttt{r25} =< 1; \texttt{d_in} = 0, \texttt{dr_in} = 0; 3; \texttt{rw_1} = 0, \texttt{rw_2} = 0; 1 >$$
$$\texttt{r39} =< 3; \texttt{d_in} = 0, \texttt{dr_in} = 0; 3; \texttt{rw_1} = 0, \texttt{rw_2} = 0; 1 >$$
$$\texttt{r40} =< 3; \texttt{d_in} = 0; 0; \texttt{cfetch_1} = 0, \texttt{cfetch_2} = 0; 0 >$$

6.2.2 Extended Operation of the CPU

The extended operation of the CPU corresponds the PMC fault model. The set of states is augmented by the states that describe the behavior of the CPU in the following cases:

- The CPU is error-free and the read data is corrupted (erroneous colored tokens on d_in or c_in). Erroneous control signals are not considered. The additional states are: s4, s5, s6.

- The CPU is error-free and the read data is possibly corrupted (as uncertain colored tokens on d_in or c_in). Erroneous or possible erroneous control signals are not considered. The additional states are: s7, s8, s9, s10.

- The CPU is erroneous and error-free tokens are received on the data and control lines. In this states the CPU is supposed to deliver error-free control signals rw, cfetch, da, ca, and erroneous data signals d. The additional states are: s11, s12, s13.

- The CPU is erroneous and the read data is corrupted (erroneous colored tokens on d_in or c_in). The input and output control signals are error-free. The additional states are: s14, s15, s16.

- The CPU is erroneous and the read data is possibly corrupted. Input and output control signals are error-free, data signals are denoted by mostly uncertainly colored tokens. The additional states are: s17, s18, s19.

The set of firing rules is extended in order to describe the erroneous behavior of the system. For each set of additional states about the same number of rules is produced as for the basic operation. Due to space limitation the firing rules are not explained here in detail, nor the FSM describing the behavior of the extended operations.

6.2.3 Uncertainty Effects

As already mentioned, the model of MEMSY is hardware related and the neglection of software causes further uncertainties. A good example for this additional uncertainty is presented in the following.

Consider the initial state s0 of the CPU. The code fetch is finished, but there is no differentiation between the instructions, we do not know whether a fetch, a read, a write, or a read-modify-write cycle has to be started. Therefore, a random selection is necessary among the firing rules that describe the above cycles.

Another example is the addressing on the S-bus. Since data values are not considered, the bus arbiter can not decide to which peripheral device the tokens have to be sent. Selection is done randomly again.

If we know the software details the sequence of instructions could be decided and the addressing of peripherals on the buses would have been unique.

6.3 Evaluation

The evaluation of the three MEMSY models aims at the comparison of the testability measures of the systems. In the first step tests are generated for the erroneous states of the nodes under the single fault assumption. The next step is concurrent fault simulation, that is done under the single fault assumption too. Finally, in the third step the dependency graph is constructed and integrated diagnostics is executed. The first two steps of the evaluation are automated. The programs were run on an AlphaServer 2100 under OSF 3.2 in user mode at the Information Science III Department of the University of Erlangen. Unfortunately the results of the evaluation could not been compared to the results delivered by other, known tools, since in the knowledge of the authors no similar testability analysis environment exist yet.

6.3.1 Test Generation

In the MEMSY example the operating SW is either neglected or its model is embedded into the DF description of the HW components (implicit SW model). Together with the PMC-like fault propagation model it results a considerably high degree of nondeterminism, that leads to a high memory consumption in case of list based test generation. Therefore; the presented tests have been generated by a modified version of the PODEM algorithm (see Chapter 4), in which simple tokens are propagated instead of lists of tokens.

MEMSY-I	MEMSY-II	MEMSY-III
AD-DMPX	AD-DMPX	AD-DMPX
CMMU-D1	CMMU-D1	CMMU-D1
CMMU-MC-D1	CMMU-MC-D1	CMMU-MC-D1
	CMMU-SEL	
CPU1	CPU1	CPU1
CPU-MC1	CPU-MC1	CPU-MC1
DUART	DUART	DUART
EPROM	EPROM	EPROM
IT-CONV	IT-CONV	IT-CONV
MEM	MEM	MEM
NVRAM	NVRAM	NVRAM
S-UB	S-UB	S-UB
S-VME	S-VME	S-VME
SCSI	SCSI	SCSI
SRAM	SRAM	SRAM
TERM1	TERM1	TERM1
TERM2	TERM2	TERM1
VME-ARB	VME-ARB	VME-ARB

Table 6.5: Components for Which Test Generation Was Successful

Therefore, the generated networks are only potential test candidates, i.e. it is possible that the result of test generation is not a test since error propagation does not deliver error pairs in each possible case of execution (refer to Definition 5.5). For this reason the feasibility of the results has to be controlled by fault simulation. In case of a positive answer a test is found. Tests in the MEMSY model seem to be pure hardware tests, but they also need some software support to be executed, i.e. the propagation path through the bus arbiter was selected randomly, but it has to be made sure by software, that during testing the same path is used.

In the MEMSY example tests could not have been generated for each fault of the system. Table 6.5 shows the components, for which test generation was successful. Concerning the results two important observation can be done: 1) test exists only for about 40% of the components, 2) the components for which test exists are the same for all three systems. This, at first sight, low efficiency of test generation is a result of many factors:

- very pessimistic fault propagation model (PMC)
- high degree of uncertainty (modeling of SW)
- relatively small number of POs

- nontestable components (fault tolerance)
- lack of information for modeling (CPU card)
- modeling of a single node

Since MEMSY is a fault-tolerant system, there are components that support to achieve fault tolerance. Such components are usually nontestable. These components are mainly the mater-checker units (7 of them), the master-checker interface (MCI), the WDP, the CMMUs (7 of them), and the reset logic (RESET). Unfortunately, there were components for which very few information was available, due to Motorola's reluctance. They are related mainly to the CPU card, and to the disc sub-system. Finally there are components, which can meaningfully function only in a multi-node configuration. These components are the ethernet controller (LAN), the watchdog processor (WDP), and the inter-node communication components as well (CMC, COUPL1, COUPL2, CM). Taking the above facts into consideration, the efficiency of test generation could not be characterized too bad any more.

In the three different setups of MEMSY the same components are used, they differ only in some minor configuration modifications that aim increased fault tolerance. Therefore; tests of the three systems are very much the same, and components for which test generation delivered positive results are almost the same. It shows that testability in our example is independent of the structure and is determined by the function of the components. (Although the structures of the three systems are similar.)

Finally, Figure 6.4 shows the dataflow model of one of the tests. It has been generated for the fault of Terminal #1 (TERM1). The test is started by typing a character on TERM1. As a result the DUART component initiates an interrupt at the CPU. The CPU reads the interrupt vector, that shows to the lower part of the memory. Suppose that the EPROM is mapped to this part of the memory. Therefore, the CPU starts to execute the interrupt routine by fetching code from the EPROM. In case of the test, the interrupt routine commands the CPU to read the character from TERM1 and to send it to TERM2. The comparison of the two characters is left to the user, who, by comparing the typed in characters with the printed ones can recognize a possible fault of TERM1. Note that the loops are not cut on the Figure in order to save space.

6.3.2 Testability Analysis

Testability analysis is executed using the results of concurrent fault simulation. The dependency matrices of the models are omitted, since they are too large, e.g. in case of MEMSY-I a size of 17 (# of tests) by 41 (# of components). In integrated diagnostics only single fault testability measures are computed, since test generation and fault simulation have been done according to the single fault assumption. The three most important measures are:

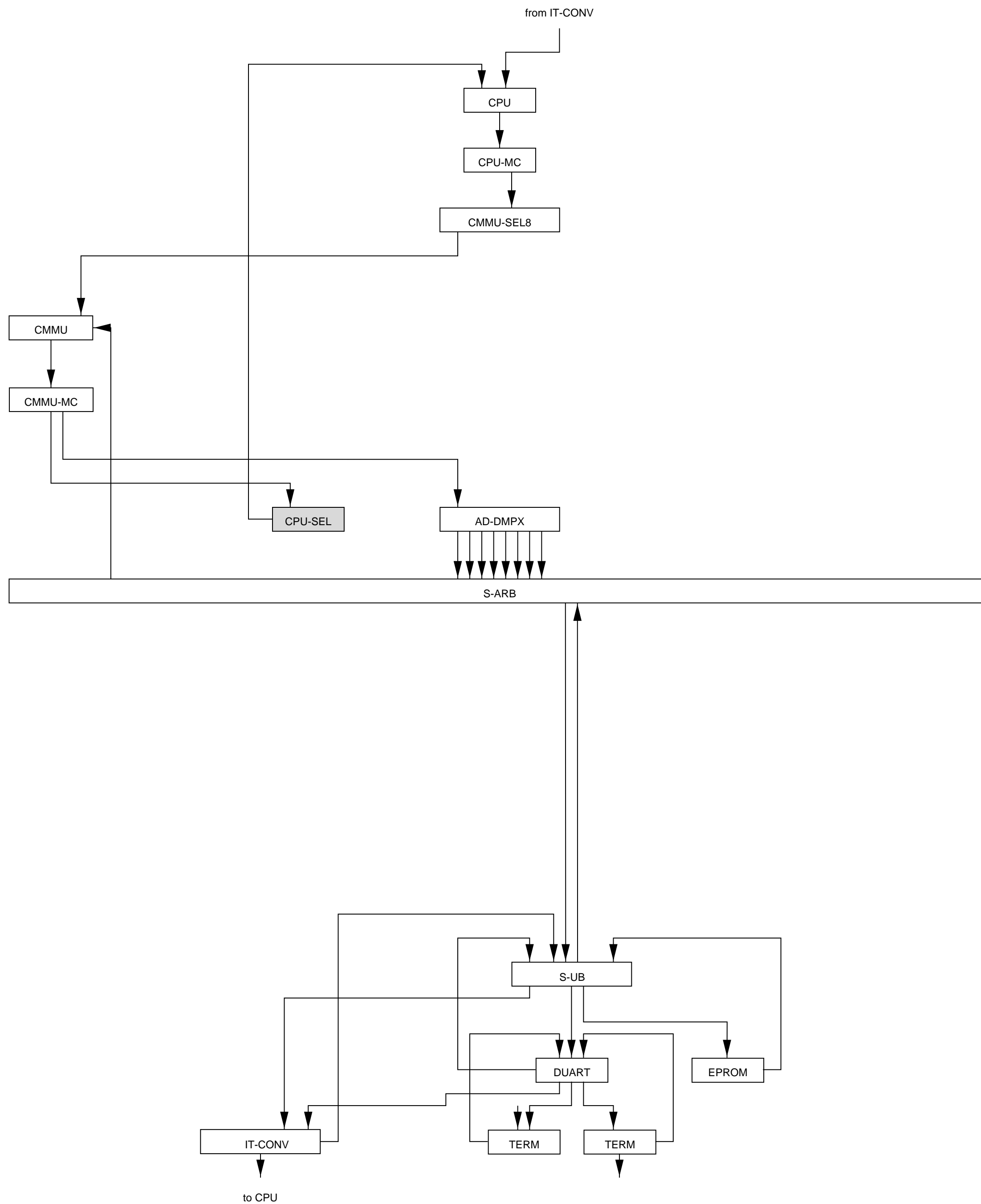


Figure 6.4: Test for the Data Fault of Terminal #1

testability [%]	MEMSY-I		MEMSY-II		MEMSY-III	
	pessimistic	optimistic	pessimistic	optimistic	pessimistic	optimistic
detectability	32	46	26	47	27	44
isolation level	22	37	21	33	14	18
diagnosability	15	34	19	21	7	9

Table 6.6: Results of Testability Analysis

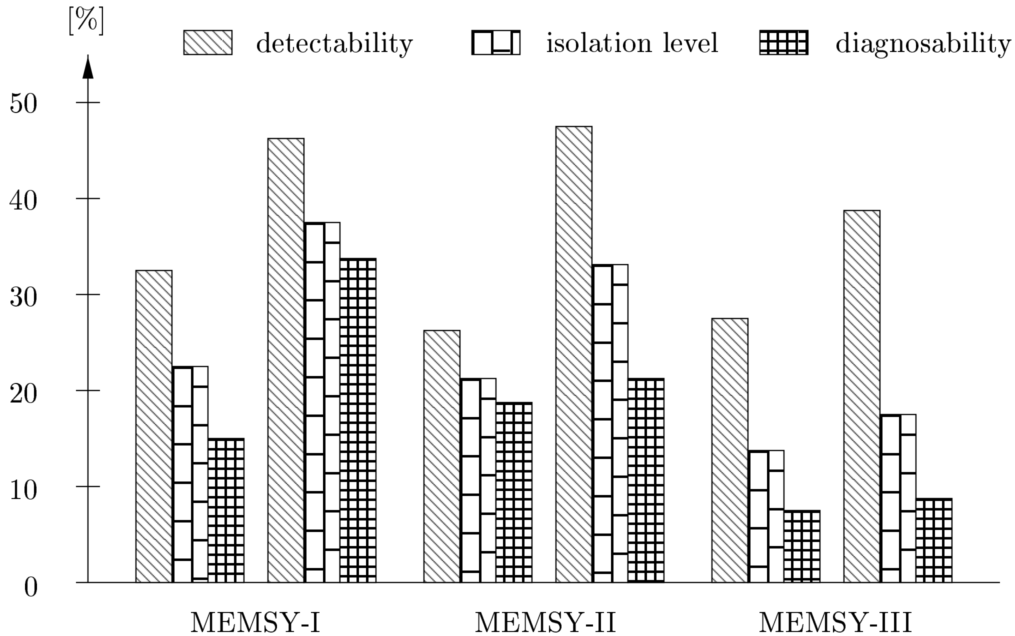


Figure 6.5: Results of Testability Analysis

- detectability
- isolation level
- diagnosability

Table 6.6 and Figure 6.5 show the results of the analysis. The diagram is divided into three parts according to the three systems. Within one part the pessimistic measures are given on the left and the optimistic measures on the right. It can be seen, as expected, that the optimistic measures are always better. The relatively poor testability measures are due to the following:

- small number of tests
- bad quality of tests

Reasoning on the small number of tests can be found in the previous subsection. It could be increased by: 1) increasing the number of POs by adding some POs for testability, 2) decreasing the number of nontestable components by altering their function

#tests	MEMSY-I		MEMSY-II		MEMSY-III	
	pessimistic	optimistic	pessimistic	optimistic	pessimistic	optimistic
before	17	17	18	18	17	17
after	9	11	7	8	4	5

Table 6.7: Results of Test Set Optimization

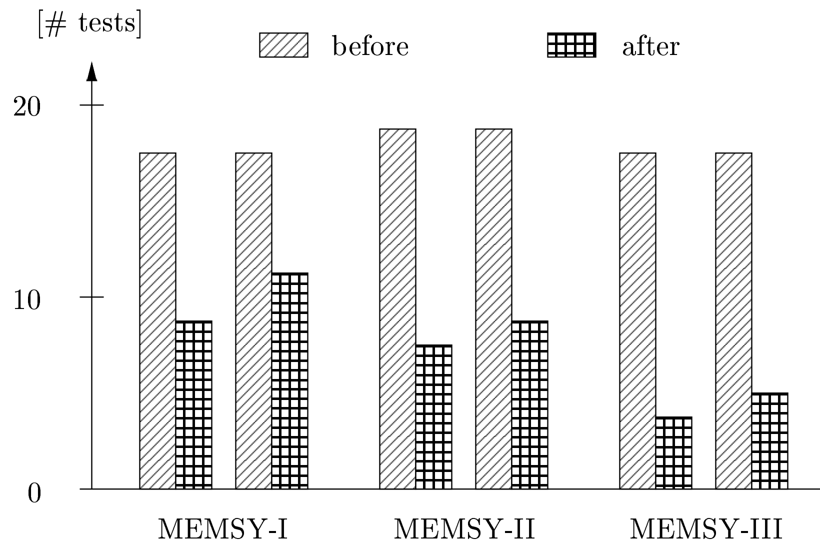


Figure 6.6: Results of Test Set Optimization

(it is not very useful), 3) by decreasing uncertainty by defining SW or by using a less pessimistic fault propagation model then PMC. Unfortunately the firing rules of the dataflow nodes have been generated by hand, thus generating another model just for comparison would be a huge overhead.

The second factor that can influence testability is the quality of the tests. Test quality is not a very well defined notion. It can be defined with terms of number of components participating in a given test, number of iteration in the iterative array model (sequentiality of a test), or by the number of faults the test can detect (a test can detect other faults then it has been generated for). A useful experiment would be to generate tests for the MEMSY system by an expert by hand, but at this size of complexity it can not be performed.

Two examples for in our opinion good and bed tests are given below. The test given in Figure 6.4 is very close to the optimal solution. On the other hand a test for the same fault have been generated once, that started from the RESET component, went through SRAM, VME-BUS, SCSI controller, DISC, EPROM, TERM1, MEM, TERM2. Clearly this second test used too much components unnecessarily.

Since the above mentioned effects are valid for all three systems, there is no significant difference between the testability measures of MEMSY-I, MEMSY-II, and MEMSY-III.

The second part of the analysis aims at optimizing the test set by removing excess tests, i.e. tests that deliver equivalent result for each fault assumption. The result of the

exec. time [s]	MEMSY-I	MEMSY-II	MEMSY-III
testgen	18	20	25
parsim	27	30	36

Table 6.8: Execution Time of a Single Run

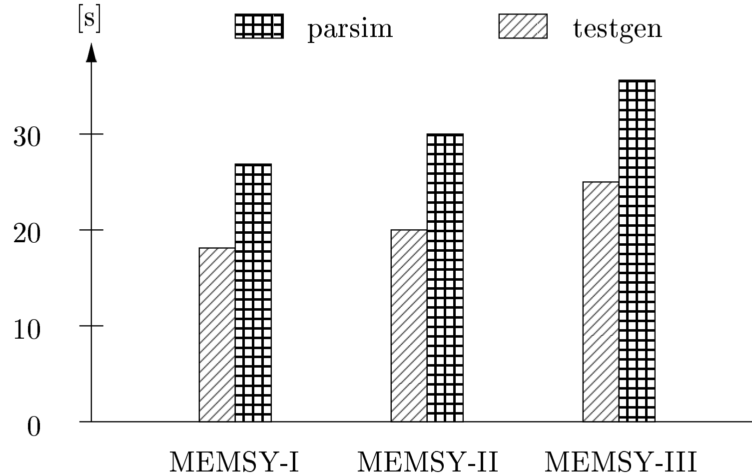


Figure 6.7: Execution Time of a Single Run

analysis is given in Table 6.7 and Figure 6.6. The structure of the Figure is the same as in the first part of the analysis.

It is interesting, that the number of tests can significantly be reduced by removing excess tests. Before optimization the number of tests is roughly the same for all three systems. Optimization yields fewer tests in the pessimistic case as in the optimistic case. This is obvious, since the better testability measures of the optimistic case can be guaranteed only by a slightly larger number of tests.

Another observation is, that the number of tests in the optimized test set is smaller for the more complex systems (i.e. MEMSY-III) than for the simpler ones. This is due to the fact, that the worse testability measures of the complex system can be accomplished by a smaller number of tests.

6.3.3 Time Complexity of the Evaluation

In order to show the complexity of the test generation and concurrent fault simulation task, execution time measurements were done on the AlphaServer. Table 6.8 and Figure 6.7 show the results of the measurements. The figure is divided into three parts according to the three evaluated systems.

Execution time is measured for the single runs of test generation and concurrent fault simulation. The full evaluation for MEMSY-I consists of 41 test generation runs for the 41 possible single faults, and 17 concurrent fault simulation runs for the 17 found tests, that would give a total of 19 minutes 57 seconds.

It can be seen on the figure, that the evaluation of the slightly more complex models, e.g. MEMSY-II and MEMSY-III, takes slightly more time than in case of MEMSY-I.

6.4 Contribution

In this chapter I presented the detailed analysis of three different setups of MEMSY. The aim of the analysis was to compare the testability measures of the different architectures of the system. The size of MEMSY can truly be considered as a real industrial size problem that can be encountered during the application of a CAD tool based on the presented framework. The evaluation of the model showed, that the execution of such an analysis is possible even with desktop computers within acceptable time.

Chapter 7

Conclusion and Future Work

In this work we presented a dataflow modeling approach for the design of digital computing systems. The most important achievement of my work is, that a novel, systematic methodology is given for fault modeling with dataflow networks. This method is able to handle the functional nondeterminism of the system and modeling uncertainty as well. Therefore, the modeling approach can be used in early phases of system design. CAD tools using the suggested approach can support testability analysis and test design as an integral part of the design process, since in the proposed dataflow model information is incorporated for both the functional and the error propagation behavior of the components. By means of a simple example we have shown the feasibility of the method. It showed that even in this phase of system engineering test strategy design and testability analysis can be done concurrently with the system design. An application study, i.e. the analysis of MEMSY, showed the usability of the approach for industrial size systems.

Future work can be focused on three main fields of theoretical research and system implementation:

- Based on the presented modeling framework a CAD tool for dependable codesign is under implementation at the Department of Measurement and Information Systems of the Technical University of Budapest. This task contains mainly implementation related work: programming the still not implemented modules of the system; fitting to the other modules and optimization existing modules. In order to present the full functionality of the tool component and requirement library for some selected application areas, e.g. biomedical systems, railway interlocking systems, have to be created.

- Some planned modules need further theoretical research. That is the solution of reliability analysis at an abstraction level, where the assignment of fault probabilities is feasible. In this case the usage of stochastic processes is clearly not avoidable. The extraction of the input model of FMEA from the results of fault simulation is still unsolved. Although some investigations have already been done [TCP95] the problem of scheduling the generated tests more efficiently has to be

solved. An interesting problem is to investigate the effects of the testing criteria and the requested testability as well as the generated tests on the HW-SW separation process of the system.

- Theoretical investigation is necessary on the possible automatic derivation of the fault model. Since the type of fault model used during modeling is highly dependent on the application area and its requirements, i.e. general fault-tolerant application, safety critical application, security centered application, libraries of components can be built for these application areas. Once the application area and the corresponding fault model have been defined automatic generation of the firing rules should be possible for many components. It should even be possible to generate the dataflow description from other high-level description such as Statecharts or UML, that are the two most widely used modeling languages for the formal description in embedded system design.

- Although in Chapter 4 a fully functioning PODEM based algorithm is given for test generation, the problem has to be further investigated. First of all other automatic logic gate-level test pattern generation algorithm has to be adapted to the dataflow approach. A comparison of these algorithms can then be done in order to evaluate which of the algorithms is the most efficient. Efficiency of the algorithms can be increased by suiting them to the needs of dataflow networks by using heuristics. One such possible improvement can be to invent a clever mechanism to control the selection of still unassigned inputs of the dataflow nodes in PODEM. However the elaboration of heuristics presumes a great number of modeled and evaluated systems.

- Last but not least considerations have to be done to use the suggested approach for other dataflow like languages that are currently being used in automated system design tools. One of the most promising of them is UML (Unified Modeling Language), which is an object oriented modeling language currently under standardization by ISO.

Bibliography

[ABC+96] B. Antal, A. Bondavalli, Gy. Csertán, I. Majzik, and L. Simoncini. Reachability and Timing Analysis in Data Flow Networks: A Case Study. In *Proceedings of the 22nd Euromicro Conference*, pages 193–200, September Prague, Czech Republic, 1996.

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

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.

[AM82] V. D. Agrawal and M. R. Mercer. Testability Measures — What Do They Tell Us? In *Proceedings of the IEEE International Test Conference, ITC'82*, pages 391–396, 1982.

[And89] A. J. Anderson. Data Flow Systems. In *Multiple Processing: A systems overview*, chapter 10, pages 441–488. Prentice Hall, UK, 1989.

[Arm92] J. R. Armstrong. Hierarchical Test Generation: Where We Are and Where We Should Be Going. In J. M. Schoen, editor, *Performance and Fault Modeling with VHDL*, pages 434–439. Prentice Hall Inc., 1992.

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

[BBC+97] C. Bernardeschi, A. Bondavalli, Gy. Csertán, I. Majzik, and L. Simoncini. Temporal Analysis of Data Flow Control Systems. *IFAC Automatica*, 38(2), 1997.

[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.

[BCJ+97] F. Balarin, M. Chiodo, A. Jurecska, H. Hsieh, A. L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tab-

bara. Hardware-Software Co-Design of Embedded Systems: A Polis Approach. To be published by Kluwer Academic Press, June 1997.

[BCS91] G. Buonanno, C. Costi, and D. Sciuto. Fault Modeling and Test Pattern Generation in the Design of Array Processors. In *Proceedings of the IEEE International Test Conference, ITC'91*, 1991.

[BF76] M. A. Breuer and A. D. Friedman. *Diagnosis & Reliable Design of Digital Systems.* Computer Science Press, Rockville, Maryland, 1976.

[BFG96] C. Bernardeschi, A. Fantechi, and S. Gnesi. JACK goes Industrial. *ECRIM News*, (25), April 1996.

[BH85] D. Bhattacharya and J. P. Hayes. High-Level Test Geneartion Using Bus Faults. In *Proceedings of the 15th IEEE International Fault-Tolerant Computing Symposium*, pages 65–70, June 1985.

[BMH89] D. Bhattacharya, B. T. Murray, and J. P. Hayes. High-Level Test Generation for VLSI. *IEEE Computer*, pages 16–24, April 1989.

[Bo93] C. Bounes, et al. SURF-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 668–673, Toulouse, France, 1993.

[Bro89] M. Broy. *Towards a Formal Foundation of the Specification and Description Language SDL.* Universität Passau, Germany, 1989.

[Bro95] M. Broy. (Inter-)Action Refinement: The Easy Way. In J. Rozenblit and K. Buchenrieder, editors, *Codesing*, pages 67–97. IEEE Press, 1995.

[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.

[Cam79] R. Camposano. From Behavior to Structure: High-Level Synthesis. *IEEE Design & Test of Computers*, 7(5):8–19, November 1979.

[Can93] N. Cannata. Un Modello Formale di Reti Dataflow per Sistemi di Controllo. Master's thesis, Department of Information Engineering, University of Pisa Pisa, Italy, 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.

[CGL+94] R. Cleaveland, J.N. Gada, P.M. Lewis, S.A. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory - Practical Tools for Specification, Simulation, Verification, and Implementation of Concurrent Systems. In *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, New Jersey, USA, May 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.

[CHG+94] M. Dal Cin, W. Hohl, A. Grygier, H. Hessenauer, U. Hildebrand, J. Honig, F. Hofmann, C. U. Linster, E. Michel, A. Pataricza, T. Thiel, and S. Turowski. Architecture and Realization of the Modular Expandable Multiprocessor System MEMSY. In *Proceedings of the 1st Internation Conference on Massively Parallel Computing Sytems MPCS'94, Ischia, Italy*, pages 7–15. IEEE CS Press, May 1994.

[Cig89] J. L. Cigler. Integrated diagnostic support system approach to fault isolation in the operational environment. In *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference*, Dayton, Ohio,1989.

[CL89] J. Carroll and D. Long. *Theory of Finite Automata.* Prentice-Hall International, 1989.

[CP89] S. J. Chandra and J. H. Patel. A Hierarchical Approach to Test Vector Generation. In *Proceedings of the ACM/IEEE International Test Conference*, pages 480–490, 1989.

[CP96] Gy. Csertán and A. Pataricza. On Diagnosability in HW-SW Codesign: A Case Study. In *Proceedings of the IEEE International Workshop on Embedded Fault-Tolerant Systems, EFTS'96*, September Dallas, USA, 1996.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-based Verification Tool for Finite-state Systems. In *Proceedings of the Workshop on Automated Verification Methods for Finite-state Systems*, volume 407 of *LNCS*. Springer-Verlag, 1989.

[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, IPDS'95*, pages 316–325, April Erlangen, Germany, 1995.

[CPS96] Gy. Csertán, A. Pataricza, and E. Selényi. Design for Testability with HW-SW Codesign. *Periodica Polytechnica*, 40(1):25–37, 1996.

[Cse93] Gy. Csertán. Temporal Analisys of Dataflow Computational Paradigm Based Control Systems. Technical Report GyCs–LS 1/1993, Department of Information Engineering of the University of Pisa, Via Diotisalvi 2, I–56126 Pisa, Italy, 1993.

[Cse96] Gy. Csertán. System Diagnostics in HW-SW Codesign. In B.Straube and J. Schönherr, editors, *Proceedings of the 4th GI/ITG/GME Workshop*, pages 51–60, March Kreischa, Germany, 1996. ISBN 3–8265–1321–5.

[Cse97] Gy. Csertán. Dependability Analysis in HW-SW Codesign. Technical Report 3/97, Institute of Computer Science III, University of Erlangen-Nürnberg, Martenstr. 3, D–91058 Erlangen, Germany, 1997.

[CW91] R. Camposano and W. Wolf, editors. *High-Level VLSI Synthesis*. Kluwer Academic Publisers, Norwell, Mass., 1991.

[dBdRR90] J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[Den85] J. B. Dennis. Data flow computation. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*, volume F14 of *NATO ASI Series*, pages 345–397. Springer-Verlag, 1985.

[DJ97] B. P. Dave and N. K. Jha. COFTA: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Architectures for Low Overhead Fault Tolerance. In *Proceedings of the 27th IEEE International Conference on Fault-Tolerant Systems ,FTCS-27*, Dallas, Texas, USA, July 1997.

[dPMCKS89] d. P. M. Crastes, M. Karam, and G. Saucier. Testability Expertise and Test Planning from High-Level Specification. In *Proceedings of the IEEE International Test Conference, ITC'89*, pages 692–699, 1989.

[ELWW95] B. L. Evans, E. A. Lee, M. C. Williamson, and D. Wilson, et al. Ptolemy Tutorial. In *Proceedings of the 2nd Annual ARPA Rapid Prototyping of Application Specific Signal Processors Conference*, Alexandria, VA, July 1995.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CAESAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, New Brunswick, New Jersey, USA, August 1996. Springer-Verlag.

[Fuj86] H. Fujiwara. *Logic Testing and Design for Testability.* MIT Press Series in Computer Systems. MIT Press, 1986.

[FUT95] Technical REPORT FUTEG-4/1995. Contract number CP93:9624, 1995.

[GA85] A. G. Gupta and J. R. Armstrong. Functional Fault Modeling and Simulatoin for VLSI Devices. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pages 720–726, 1985.

[Gen86] H. J. Genrich. Predicate/Transition Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1986, Lecture Notes on Computer Science*, volume 254, pages 207–247. Springer Verlag, 1986.

[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.

[Gol79] L. H. Goldstein. Controllability/Observability Analysis of Digital Circuits. *IEEE Transactions on Circuits and Systems*, 26:685–693, 1979.

[Har87] D. Harel. Statecharts: A Visual Formalism of Complex Systems. *Scientific Computer Programming*, (8):231–274, 1987.

[Har97] D. Harel. Some Thought on Statecharts, 13 Years Later. In *Proceedings of the Workshop on Formal Design of Safety Critical Embedded Systems, FEmSys'97*, Munich, Germany, April 1997.

[HG87] H. Hofestädt and M. Gerner. Qualitative Testability Analysis and Hierarchical Test Pattern Generation — A New Approach to Desigh for Testability? In *Proceedings of the IEEE International Test Conference, ITC'87*, pages 538–546, 1987.

[IEE93] *IEEE Standard VHDL Language Reference Manual*, IEEE Standard 1076–1993, 1993.

[JA91] R. Jagannathan and E.A. Ashcroft. Fault Tolerance in Parallel Implementations of Functional Languages. In *Proceedings of FTCS-21*, pages 256–263, Montreal, Canada, 1991.

[Jen90] K. Jensen. Coloured Petri Net: A High Level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990, Lecture Notes on Computer Science*, volume 483, pages 207–247. Springer Verlag, 1990.

[Jon89a] B. Jonsson. On Decomposing and Refining Specifications of Distributed Systems. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors,

Stepwise Refinement of Distributed Systems, number 430 in Lecture Notes in Computer Science, pages 361–385. Springer-Verlag, May 1989.

[Jon89b] 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.

[JRF91] Jr. J. R. Franco. Using integrated diagnostics on automatic test equipment. In *Proceeedings of the IEEE Systems Readiness Technology Conference*, Anaheim, California, 1991.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the IFIP '74*, pages 471–475. North Holland, 1974.

[KBB86] K. M. Kavi, B. P. Buckles, and U.N. Bhat. A Formal Definition of Dataflow Graph Models. *IEEE Transactions on Computers*, 35(11):940–948, November 1986.

[Kei90] W. Keiner. A Navy Approach ti Integrated Diagnostics. In *Proceedings of the IEEE Automatic Test Conferenct, AUTOTESTCON'90*, pages 443–450, 1990.

[Kri87] B. Krishnamurthy. Hierarchical Test Generation: Can AI Help? In *Proceedings of the International Test Conference*, pages 694–700, September 1987.

[KS92] K. M. Kavi and B. Shirazi. Dataflow architecture. *IEEE Potentials*, 11(3):27–30, October 1992.

[LCSC94] R. R. Leitch, M. J. Chantler, Q. Shen, and G. M. Coghill. *Modelling Properties and Classification of Approaches*, 1994. Deliverable Report: D1, Mind project.

[Lee91] E. A. Lee. Consistency in Dataflow Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceeding of the IEEE*, 75(9):35–45, 1987.

[Maj94] I. Majzik. On the Semantic of Dataflow Computational Paradigm Based Control Systems. Technical Report IM–LS 1/1993, Department of Information Engineering of the University of Pis a, Via Diotisalvi 2, I–56126 Pisa, Italy, 1994.

[Mar78] R. A. Marlett. EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits. In *Proceedings of the I15th Design Automation Conference*, pages 335–339, June 1978.

[MH88] B. T. Murray and J. P. Hayes. Hierarchical Test Generation Using Precomputed Tests for Modules. In *Proceedings of the International Test Conference*, pages 221–229, September 1988.

[Mil85] *Testability Program for Electronic Systems and Equipment*, Mil-Std 2165, 1985.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

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

[Pet62] C. A. Petri. Kommunikation mit Automaten. Schriften des IIM Nr. 3, Institut für Instrumentelle Mathematik, Bonn, 1962.

[PMC78] Preparata, Metzge, and Chien. Binary decision diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.

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

[RBS67] 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.

[RK78] Read and King. Binary decision diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.

[RS80] C. Robach and G. Sausier. Functional Testing. In *Proceedings of the International Test Conference, ITC'80*, 1980.

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

[Sel85] E. Selényi. Generalization of System-Level Diagnosis, 1985. In Hungarian.

[SIQW95] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko. The UltraSAN Modeling Environment. *Performance Evaluation*, 24(1):89–115, October 1995.

[SPTP96] B. Sallay, A. Petri, K. Tilly, and A. Pataricza. High Level Test Pattern Generation for VHDL Circuits. In *Proceedings of the IEEE European Test Workshop, ETW'96*, pages 202–206, Montpellier, France, June 1996.

[SS87] G. M. Silberman and I. Spillinger. Test Generation Using Functional Fault Simulation. In *Proceedings of the International Test Conference, ITC'87*, pages 400–407, 1987.

[SS91a] J. W. Sheppard and W. R. Simpson. A Mathematical Model for Integrated Diagnostics. *IEEE Design & Test of Computers*, 8(4):25–38, December 1991.

[SS91b] J. W. Sheppard and W. R. Simpson. Uncertainty Computations in Model-based Diagnostics. In *Proceedings of the IEEE Systems Readiness Technology Conference*, Anaheim, California, 1991.

[SS91c] W. R. Simpson and J. W. Sheppard. System Complexity and Integrated Diagnostics. *IEEE Design & Test of Computers*, 8(3):16–30, September 1991.

[SS91d] W. R. Simpson and J. W. Sheppard. Partitioning Large Diagnostic Problems. In *Proceedings of the IEEE Systems Readiness Technology Conference*, pages 329–335, Anaheim, California, 1991.

[SS92a] J. W. Sheppard and W. R. Simpson. Applying testability analysis for integrated diagnostics. *IEEE Design & Test of Computers*, 9(3):65–78, September 1992.

[SS92b] J. W. Sheppard and W. R. Simpson. Fault Diagnosis Under Temporal Constraints. In *Proceedings of the IEEE AUTOTESTCON Conference*, pages 151–159, 1992.

[SS92c] W. R. Simpson and J. W. Sheppard. System Testability Assessment for Integrated Diagnostics. *IEEE Design & Test of Computers*, 9(1):40–54, March 1992.

[SS93a] J. W. Sheppard and W. R. Simpson. Performing Effective Fault Isolation in Integrated Diagnostics. *IEEE Design & Test of Computers*, 10(2):78–90, June 1993.

[SS93b] W. R. Simpson and J. W. Sheppard. Fault Isolation in an Integrated Diagnostic Environment. *IEEE Design & Test of Computers*, 10(1):52–66, March 1993.

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

[Szi79a] J. Sziray. A Test Calculation Algorithm for Module-level Combinational Networks. *Digital Processes*, 5(1–2):17–26, 1979.

[Szi79b] J. Sziray. Test Calculation for Logic Network by Composite Justification. *Digital Processes*, 5:3–16, 1979.

[TCP95] I. Turcsány, Gy. Csertán, and A. Pataricza. Model Based Diagnostic-Test Scheduling. In *Proceedings of the Technical Conference on CAD Methods in Electronic and Information Processing System Design and its Education*, pages 27–30, June Budapest, Hungary, 1995.

[TSR93] K. Tilly, L. Surján, and Gy. Román. Automatic Test Pattern Generation can be Solved as a Constraint Satisfaction Problem. *Microprocessing and Microprogramming*, 38:715–722, 1993.

[Uba92] R. Ubar. Multi-Level Test Generation and Fault Diagnosis in Digital Systems. Technical report, TIM3, 46, avenue Félix Viallet, F–38031 Grenoble, France, March 1992.

[Uba94] R. Ubar. Test Generation for Digital Systems based on Alternative Graphs. In *Proceedings of the 1st European Dependable Computing Conference, EDDC-1*, pages 151–164, 1994.

[VM94] B. Victor and F. Moller. The Mobility Workbench - A Tool for the π-Calculus. In D. Dill, editor, *Proceedings of the 6th Conference on Computer-Aided Verification*, volume 818 of *LNCS*, pages 428–440. Springer-Verlag, 1994.

[WA90] P. C. Ward and J. R. Armstrong. Behavioral Fault Simulation in VHDL. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 587–593, 1990.

[Wan90] Y. Wang. A calculus of Real-Time Systems. In *In Proceedings of CONCUR*, volume 458 of *LNCS*. Springer-Verlag, 1990.

Appendix A

Abbreviations and Symbols

A.1 Abbreviations

AG	alternative graph
ALU	arithmetic logic unit
ASIC	application specific integrated circuit
ATPG	automatic test pattern generation
CAD	computer aided design
CC0	combinational zero controllability
CC1	combinational one controllability
CFSM	codesign finite-state machine
CMMU	cache memory manager unit
CO	combinational observability
CP	computation of a dataflow network
CPU	central processing unit
CSP	constraint satisfaction problem
DF	dataflow
DFG	dataflow graph
DFN	dataflow network
DR	domain refinement
DSP	digital signal processing
EPROM	electrically programmable read only memory
FIFO	first in first out
FMEA	failure mode and effect analysis
FPGA	field programmable gate array
FS	firing sequence
FSM	finite-state machine
GSPN	generalized stochastic Petri net
HW	hardware
ID	integrated diagnostics

LAN	local area network
LED	light emitting diode
LSI	large scale integration
M/C	master-checker
MC	Markov chain
MEMSY	modular expandable multiprocessor system
MG	marked graph
MTBF	mean-time between failure
MTTF	mean-time to first failure
MTTR	mean-time to repair
ND	not defined
NDFST	nondeterministic finite-state transducer
NVRAM	non-volatile random access memory
NaR	not a refinement
PA	process algebra
PC	personal computer
PI	primary input
PLD	programmable logic device
PMC	Preparata-Metzge-Chien
PN	Petri net
PO	primary output
PODEM	path oriented decision making
RAM	random access memory
ROM	read only memory
SAN	stochastic activity network
SC0	sequential zero controllability
SC1	sequential one controllability
SCSI	small computer system interface
SDL	specification description language
SO	sequential observability
SR	structure refinement
SRAM	static random access memory
SW	software
TA	testability analysis
TCDG	test conclusion dependency graph
TPG	test pattern generation
UML	unified modeling language
VDM	Vienna development method
VHDL	very high-scale hardware description language
VLSI	very large-scale integration
WDP	watchdog processor

A.2 Symbols

A.2.1 Dataflow Notation

c	channel of a dataflow network
C	set of channels of a dataflow network
cp	computation of a dataflow network
CP	set of computations of a dataflow network
fs	firing sequence of a dataflow network
FS	set of firing sequences of a dataflow network
i	input channel of a dataflow network
I	set of input channels of a dataflow network
im	input mapping of a dataflow network
IM	set of input mappings of a dataflow network
im_n	input mapping of node n
IM_n	set of input mappings of node n
in	internal channel of a dataflow network
IN	set of internal channels of a dataflow network
i_n	input channel of node n
I_n	set of input channels of node n
m	token of a dataflow network
M	token set of a dataflow network
m_n	token of node n
M_n	token set of node n
n	dataflow node of a dataflow network
N	set of nodes of a dataflow network
o	output channel of a dataflow network
O	set of output channels of a dataflow network
om	output mapping of a dataflow network
OM	set of output mappings of a dataflow network
om_n	output mapping of node n
OM_n	set of output mappings of node n
o_n	output channel of node n
O_n	set of output channels of node n
π	priority of a firing rule
r_n	firing rule of node n
R_n	set of firing rules of node n
$\mathcal{R}$	refinement of a dataflow network
$\mathcal{R}_d$	domain refinement of a dataflow network
$\mathcal{R}_s$	structure refinement of a dataflow network
S	set of states of a dataflow network

s state of a dataflow network

s_n^0 initial state of node n

s_0 initial state of a dataflow network

S_n set of states of node n

s_n state of node s

s_{post} successor state of a firing

s_{pre} predecessor state of a firing

X_{in} input mapping of a firing rule

X_{out} output mapping of a firing rule

A.2.2 Nondeterministic Finite-State Transducer Notation

Σ input alphabet

, output alphabet

S set of states

s_0 initial state

δ state transition function

ω output function

$\bar{\omega}$ extended output function

A.2.3 Set and String Notation

A set

$\|A\|$ cardinality of A

$\|A\|_w$ weighted cardinality of A

A^* power set of A

$A \times B$ Cartesian product of A and B

A^n string from the elements of A in length n

$|A^n|$ length of string A^n

$A^\star$ set of strings constructed from A

A.2.4 Testability Notation

C set of conclusions in integrated diagnostics

C_i a given conclusion in integrated diagnostics

f a given fault

F set of faults

f_0 notation of the fault-free system

F_c set of certainly detectable faults

F_n set of not detectable faults

F_u set of uncertainly detectable faults

t a given test

T	set of tests
tr	test result
tv	test vector
T_i	a given test in integrated diagnostics

Appendix B

Definitions and Derivations

B.1 Definitions

Definition 2.1 Dataflow node 18

Definition 2.2 Dataflow channel 19

Definition 2.3 Dataflow network 19

Definition 3.1 Domain refinement 37

Definition 3.2 Firing sequence 39

Definition 3.3 Structure refinement 39

Definition 3.4 Nondeterministic finite-state transducer 41

Definition 3.5 String 42

Definition 3.6 Length of a string 42

Definition 3.7 String refinement 42

Definition 3.8 Set refinement 42

Definition 3.9 Node to NDFST transformation 42

Definition 3.10 DFN to NDFTS transformation 42

Definition 3.11 Extended output function 43

Definition 3.12 Refinement of EOF 43

Definition 3.13 Bisimulation 43

Definition 5.1 Strong dependency relation 75

Definition 5.2 Weak dependency relation 75

Definition 5.3 Computation of a dataflow network 75

Definition 5.4 Fault set 76

Definition 5.5 Test 76

Definition 5.6 Test certainty 76

Definition 5.7 Fault detection 76

B.2 Derivations

Theorem 3.1 Refinement and bisimulation equivalence 43

Theorem 5.1 Convergence of testability measures 77

Statement 3.1 The example is a refinement 51

Appendix C

Formal Description of the Candy Automaton

In this chapter the formal definition of the candy automaton is given. Similarly to Chapter 2, first the dataflow network that denotes the error-free computation is presented, and then the dataflow network that defines the erroneous operation is given. In the definition of networks and subnetworks the state of channels is represented by an X symbol in the state of the network. For the structure of the DFGs refer to Figure 2.2 in case of the original automaton and to Figure 3.3 in case of the refined automaton.

C.1 candy automaton

basic operation

nodes N={coin_in/out, select, controller, candies_out}
channels C={coin_in, change, to_coin_in/out, from_coin_in/out,
from_select, select_candy, to_candies_out, from_candies_out, out}
states S={(ok,ok,ok,ok,X)}

extended operation

nodes N={coin_in/out, select, controller, candies_out}
channels C={coin_in, change, to_coin_in/out, from_coin_in/out,
from_select, select_candy, to_candies_out, from_candies_out, out}
states S={(ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X),
(ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ctrl,X),
(ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,stk,X),
(ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,ok,X),
(ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,ctrl,X),
(ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,stk,X),
(ok,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X),

(ok,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ctrl,X),
(ok,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,stk,X),
(ok,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,ok,X),
(ok,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,ctrl,X),
(ok,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,stk,X),
(rec,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X),
(rec,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ctrl,X),
(rec,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,stk,X),
(rec,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,ok,X),
(rec,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,ctrl,X),
(rec,ok,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,stk,X),
(rec,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ok,X),
(rec,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,ctrl,X),
(rec,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,ok,ok,X), (ok,ok,ok,stk,X),
(rec,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,ok,X),
(rec,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,ctrl,X),
(rec,ok,ok,ok,X), (ok,cont,ok,ok,X), (ok,ok,int,ok,X), (ok,ok,ok,stk,X)}

C.2 coin_in/out

basic operation

input channels	I={coin_in, to_coin_in/out}
output channels	O={from_coin_in/out, change}
states	S={ok}
initial state	s_0=ok
firings	R={r1,r2}
token set	M={ok}

extended operation

input channels	I={coin_in, to_coin_in/out}
output channels	O={from_coin_in/out, change}
states	S={ok, rec}
initial state	s_0=ok
firings	R={r1 ...r10}
token set	M={ok, inc, dead, x}

firings

```
r1=<ok;coin_in=ok;ok;from_coin_in/out=ok;0>
r2=<ok;to_coin_in/out=ok;ok;change=ok;0>
r3=<ok;to_coin_in/out=inc;ok;change=inc;0>
```

```
r4=<ok;to_coin_in/out=dead;ok;change=dead;0>
r5=<ok;to_coin_in/out=x;ok;change=x;0>
r6=<rec;coin_in=ok;rec;from_coin_in/out=x;0>
r7=<rec;to_coin_in/out=ok;rec;change=x;0>
r8=<rec;to_coin_in/out=inc;rec;change=x;0>
r9=<rec;to_coin_in/out=dead;rec;change=dead;0>
r10=<rec;to_coin_in/out=x;rec;change=x;0>
```

C.3 select

basic operation

input channels	I={select_candy}
output channels	O={from_select}
states	S={ok}
initial state	s_0=ok
firings	R={r1}
token set	M={ok}

extended operation

input channels	I={select_candy}
output channels	O={from_select}
states	S={ok, cont}
initial state	s_0=ok
firings	R={r1, r2}
token set	M={ok, inc}

firings

```
r1=<ok;select_candy=ok;ok;from_select=ok;0>
r2=<cont;select_candy=ok;cont;from_select=inc;0>
```

C.4 controller

basic operation

input channels	I={from_coin_in/out,from_select,from_candies_out}
output channels	O={to_coin_in/out,to_candies_out}
states	S={ok}
initial state	s_0=ok
firings	R={r1,r10}

token set M={ok}

extended operation

input channels	I={from_coin_in/out,from_select,from_candies_out}
output channels	O={to_coin_in/out,to_candies_out}
states	S={ok, int}
initial state	s_0=ok
firings	R={r1 ... r22}
token set	M={ok, inc, dead, x}

firings

```
r1=<ok;from_coin_in/out=ok,from_select=ok;ok;to_candies_out=ok;0>
r2=<ok;from_coin_in/out=ok,from_select=inc;ok;to_candies_out=inc;0>
r3=<ok;from_coin_in/out=ok,from_select=x;ok;to_candies_out=x;0>
r4=<ok;from_coin_in/out=inc,from_select=ok;ok;to_candies_out=inc;0>
r5=<ok;from_coin_in/out=inc,from_select=inc;ok;to_candies_out=inc;0>
r6=<ok;from_coin_in/out=inc,from_select=x;ok;to_candies_out=inc;0>
r7=<ok;from_coin_in/out=x,from_select=ok;ok;to_candies_out=x;0>
r8=<ok;from_coin_in/out=x,from_select=inc;ok;to_candies_out=inc;0>
r9=<ok;from_coin_in/out=x,from_select=x;ok;to_candies_out=x;0>
r10=<ok;from_candies_out=ok;ok;to_coin_in/out=ok;0>
r11=<ok;from_candies_out=dead;ok;to_coin_in/out=dead;0>
r12=<int;from_coin_in/out=ok,from_select=ok;int;to_candies_out=inc;0>
r13=<int;from_coin_in/out=ok,from_select=inc;int;to_candies_out=x;0>
r14=<int;from_coin_in/out=ok,from_select=x;int;to_candies_out=x;0>
r15=<int;from_coin_in/out=inc,from_select=ok;int;to_candies_out=x;0>
r16=<int;from_coin_in/out=inc,from_select=inc;int;to_candies_out=inc;0>
r17=<int;from_coin_in/out=inc,from_select=x;int;to_candies_out=x;0>
r18=<int;from_coin_in/out=x,from_select=ok;int;to_candies_out=x;0>
r19=<int;from_coin_in/out=x,from_select=inc;int;to_candies_out=x;0>
r20=<int;from_coin_in/out=x,from_select=x;int;to_candies_out=x;0>
r21=<int;from_candies_out=ok;int;to_coin_in/out=inc;0>
r22=<int;from_candies_out=dead;int;to_coin_in/out=dead;0>
```

C.5 candies_out

basic operation

input channels	I={to_candies_out}
output channels	O={from_candies_out,out}
states	S={ok}
initial state	s_0=ok

firings	R={r1}
token set	M={ok}

extended operation

input channels	I={to_candies_out}
output channels	O={from_candies_out,out}
states	S={ok, ctrl, stk}
initial state	s_0=ok
firings	R={r1 ... r9}
token set	M={ok, inc, dead, x}

firings

```
r1=<ok;to_candies_out=ok;ok;from_candies_out=ok,out=ok;0>
r2=<ok;to_candies_out=inc;ok;from_candies_out=ok,out=inc;0>
r3=<ok;to_candies_out=x;ok;from_candies_out=ok,out=x;0>
r4=<ctrl;to_candies_out=ok;ctrl;from_candies_out=ok,out=inc;0>
r5=<ctrl;to_candies_out=inc;ctrl;from_candies_out=ok,out=x;0>
r6=<ctrl;to_candies_out=x;ctrl;from_candies_out=ok,out=x;0>
r7=<stk;to_candies_out=ok;stk;from_candies_out=dead,out=dead;0>
r8=<stk;to_candies_out=inc;stk;from_candies_out=dead,out=dead;0>
r9=<stk;to_candies_out=x;stk;from_candies_out=dead,out=dead;0>
```

C.6 candies_out (refined)

basic operation

nodes	N={hwl, mechanics}
channels	C={to_candies_out, from_candies_out, to_mechanics, out}
states	S={(ok,ok,X)}

extended operation

nodes	N={hwl, mechanics}
channels	C={to_candies_out, from_candies_out, to_mechanics, out}
states	S={(ok,ok,X), (ok,stk,X), (ctrl,ok,X), (ctrl,stk,X)}

C.7 hwl

basic operation

input channels	I={to_candies_out}
output channels	O={to_mechanics}
states	S={ok}

initial state	s_0=ok
firings	R={r1}
token set	M={ok}

extended operation

input channels	I={to_candies_out}
output channels	O={to_mechanics}
states	S={ok, ctrl}
initial state	s_0=ok
firings	R={r1 ... r6}
token set	M={ok, inc, dead, x}

firings

```
r1=<ok;to_candies_out=ok;ok;to_mechnics=ok;0>
r2=<ok;to_candies_out=inc;ok;to_mechnics=inc;0>
r3=<ok;to_candies_out=x;ok;to_mechnic=x;0>
r4=<ctrl;to_candies_out=ok;ctrl;to_mechnics=inc;0>
r5=<ctrl;to_candies_out=inc;ctrl;to_mechnics=x;0>
r6=<ctrl;to_candies_out=x;ctrl;to_mechnics=x;0>
```

C.8 mechanics

basic operation

input channels	I={to_meachanics}
output channels	O={from_candies_out,out}
states	S={ok}
initial state	s_0=ok
firings	R={r1}
token set	M={ok}

extended operation

input channels	I={to_mechanics}
output channels	O={from_candies_out,out}
states	S={ok, stk}
initial state	s_0=ok
firings	R={r1 ... r9}
token set	M={ok,inc,dead,x}

firings

```
r1=<ok;to_mechanics=ok;ok;from_candies_out=ok,out=ok;0>
r2=<ok;to_mechanics=inc;ok;from_candies_out=ok,out=inc;0>
r3=<ok;to_mechanics=x;ok;from_candies_out=ok,out=x;0>
r4=<stk;to_mechanics=ok;stk;from_candies_out=dead,out=dead;0>
r5=<stk;to_mechanics=inc;stk;from_candies_out=dead,out=dead;0>
r6=<stk;to_mechanics=x;stk;from_candies_out=dead,out=dead;0>
```

Appendix D

Formal Description of MEMSY

In this chapter the formal definition of MEMSY and of the cpu component is given.

D.1 MEMSY-I

basic operation

token set	M={0: ok}
nodes	N={AD-DMPX, CIO, CM, CMC, CMMU, CMMU-MC, CMMU-SEL8, COUPL, CPU, CPU-MC, CPU-SEL, DISC, DUART, EPROM, IT-CONV, ITC, LAN, MCI, MEM, NVRAM, RESET, S-ARB, S-UB, S-VME, SCSI, SRAM, TERM, VME-ARB, WDP}
channels	C={see Figure 6.1}
states	S={combination of the error-free states of the nodes}
initial state	s_0=combinations of the error-free initial states of the nodes
firings	R={firings of the nodes in basic operation}

extended operation

token set	M={0: ok, 1: inc, 2: x}
nodes	N={AD-DMPX, CIO, CM, CMC, CMMU, CMMU-MC, CMMU-SEL8, COUPL, CPU, CPU-MC, CPU-SEL, DISC, DUART, EPROM, IT-CONV, ITC, LAN, MCI, MEM, NVRAM, RESET, S-ARB, S-UB, S-VME, SCSI, SRAM, TERM, VME-ARB, WDP}
channels	C={see Figure 6.1}
states	S={combination of the states of the nodes}
initial state	s_0=combinations of initial states of the nodes
firings	R={firings of the nodes}

D.2 CPU

basic operation

input channels I={d_in, dr_coin_in, c_in, cr_in, it, rst}
output channels O={d_1, rw_1, cfetch_1, d_2, rw_2, cfetch_2}
states S={0: cpu_ok_data_ok_end_fetch
1: cpu_ok_data_ok_end_read
2: cpu_ok_data_ok_end_write
3: cpu_ok_data_ok_end_read(ex_it)}
initial state s_0=cpu_ok_data_ok_end_fetch
firings R={r0, r1, r4, r5, r6, r7, r10, r11, r14, r17, r20, r21, r24,
r25, r26, r29, r32, r33, r34, r35, r38, r39, r40}

extended operation

input channels I={d_in, dr_coin_in, c_in, cr_in, it, rst}
output channels O={d_1, rw_1, cfetch_1, d_2, rw_2, cfetch_2}
states S={0: cpu_ok_data_ok_end_fetch
1: cpu_ok_data_ok_end_read
2: cpu_ok_data_ok_end_write
3: cpu_ok_data_ok_end_read(ex_it)
4: cpu_ok_data_faulty_end_fetch
5: cpu_ok_data_faulty_end_read
6: cpu_ok_data_faulty_end_write
7: cpu_faulty_data_ok_end_fetch
8: cpu_faulty_data_ok_end_read
9: cpu_faulty_data_ok_end_write
10: cpu_faulty_data_ok_end_read(ex_it)
11: cpu_faulty_data_faulty_end_fetch
12: cpu_faulty_data_faulty_end_read
13: cpu_faulty_data_faulty_end_write
14: cpu_ok_data_poss.faulty_end_fetch
15: cpu_ok_data_poss.faulty_end_read
16: cpu_ok_data_poss.faulty_end_write
17: cpu_faulty_data_poss.faulty_end_fetch
18: cpu_faulty_data_poss.faulty_end_read
19: cpu_faulty_data_poss.faulty_end_write}
initial state s_0=cpu_ok_data_ok_end_fetch
firings R={r0 ... r221}

firings

```
r0=<0;rst=0,c_in=0,cr_in=0;0;cfetch_1=0,cfetch_2=0;6>
r1=<0;rst=0,c_in=0;0;cfetch_1=0,cfetch_2=0;5>
r2=<0;rst=0,c_in=1;0;cfetch_1=0,cfetch_2=0;5>
r3=<0;rst=0,c_in=2;0;cfetch_1=0,cfetch_2=0;5>
r4=<0;rst=0,it=0;0;cfetch_1=0,cfetch_2=0;5>
r5=<0;rst=0;0;cfetch_1=0,cfetch_2=0;4>
r6=<0;cr_in=0,c_in=0;3;rw_1=0,rw_2=0;3>
r7=<0;it=0,c_in=0;3;rw_1=0,rw_2=0;2>
r8=<0;it=0,c_in=1;3;rw_1=0,rw_2=0;2>
r9=<0;it=0,c_in=2;3;rw_1=0,rw_2=0;2>
r10=<0;it=0;3;rw_1=0,rw_2=0;1>
r11=<0;c_in=0;0;cfetch_1=0,cfetch_2=0;0>
r12=<0;c_in=1;4;cfetch_1=0,cfetch_2=0;0>
r13=<0;c_in=2;14;cfetch_1=0,cfetch_2=0;0>
r14=<0;c_in=0;1;rw_1=0,rw_2=0;0>
r15=<0;c_in=1;5;rw_1=0,rw_2=0;0>
r16=<0;c_in=2;15;rw_1=0,rw_2=0;0>
r17=<0;c_in=0;2;d_1=0,d_2=0,rw_1=0,rw_2=0;0>
r18=<0;c_in=1;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r19=<0;c_in=2;16;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r20=<1;rst=0,d_in=0,dr_in=0;0;cfetch_1=0,cfetch_2=0;4>
r21=<1;rst=0,d_in=0;0;cfetch_1=0,cfetch_2=0;3>
r22=<1;rst=0,d_in=1;0;cfetch_1=0,cfetch_2=0;3>
r23=<1;rst=0,d_in=2;0;cfetch_1=0,cfetch_2=0;3>
r24=<1;rst=0;0;cfetch_1=0,cfetch_2=0;2>
r25=<1;dr_in=0,d_in=0;3;rw_1=0,rw_2=0;1>
r26=<1;d_in=0;0;cfetch_1=0,cfetch_2=0;0>
r27=<1;d_in=1;4;cfetch_1=0,cfetch_2=0;0>
r28=<1;d_in=2;14;cfetch_1=0,cfetch_2=0;0>
r29=<1;d_in=0;2;d_1=0,d_2=0,rw_1=0,rw_2=0;0>
r30=<1;d_in=1;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r31=<1;d_in=2;16;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r32=<2;rst=0;0;cfetch_1=0,cfetch_2=0;1>
r33=<2;;0;cfetch_1=0,cfetch_2=0;0>
r34=<3;rst=0,d_in=0,dr_in=0;0;cfetch_1=0,cfetch_2=0;3>
r35=<3;rst=0,d_in=0;0;cfetch_1=0,cfetch_2=0;2>
r36=<3;rst=0,d_in=1;0;cfetch_1=0,cfetch_2=0;2>
r37=<3;rst=0,d_in=2;0;cfetch_1=0,cfetch_2=0;2>
r38=<3;rst=0;0;cfetch_1=0,cfetch_2=0;2>
r39=<3;dr_in=0,d_in=0;3;rw_1=0,rw_2=0;1>
r40=<3;d_in=0;0;cfetch_1=0,cfetch_2=0;0>
r41=<3;d_in=1;4;cfetch_1=0,cfetch_2=0;0>
r42=<3;d_in=2;14;cfetch_1=0,cfetch_2=0;0>
r43=<4;rst=0,c_in=0,cr_in=0;0;cfetch_1=0,cfetch_2=0;6>
```

```
r44=<4;rst=0,c_in=0;0;cfetch_1=0,cfetch_2=0;5>
r45=<4;rst=0,c_in=1;0;cfetch_1=0,cfetch_2=0;5>
r46=<4;rst=0,c_in=2;0;cfetch_1=0,cfetch_2=0;5>
r47=<4;rst=0,it=0;0;cfetch_1=0,cfetch_2=0;5>
r48=<4;rst=0;0;cfetch_1=0,cfetch_2=0;4>
r49=<4;cr_in=0,c_in=0;3;rw_1=0,rw_2=0;3>
r50=<4;it=0,c_in=0;3;rw_1=0,rw_2=0;2>
r51=<4;it=0,c_in=1;3;rw_1=0,rw_2=0;2>
r52=<4;it=0,c_in=2;3;rw_1=0,rw_2=0;2>
r53=<4;it=0;3;rw_1=0,rw_2=0;1>
r54=<4;c_in=0;4;cfetch_1=0,cfetch_2=0;0>
r55=<4;c_in=1;4;cfetch_1=0,cfetch_2=0;0>
r56=<4;c_in=2;4;cfetch_1=0,cfetch_2=0;0>
r57=<4;c_in=0;5;rw_1=0,rw_2=0;0>
r58=<4;c_in=1;5;rw_1=0,rw_2=0;0>
r59=<4;c_in=2;5;rw_1=0,rw_2=0;0>
r60=<4;c_in=0;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r61=<4;c_in=1;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r62=<4;c_in=2;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r63=<5;rst=0,d_in=0,dr_in=0;0;cfetch_1=0,cfetch_2=0;4>
r64=<5;rst=0,d_in=0;0;cfetch_1=0,cfetch_2=0;3>
r65=<5;rst=0,d_in=1;0;cfetch_1=0,cfetch_2=0;3>
r66=<5;rst=0,d_in=2;0;cfetch_1=0,cfetch_2=0;3>
r67=<5;rst=0;0;cfetch_1=0,cfetch_2=0;2>
r68=<5;dr_in=0,d_in=0;3;rw_1=0,rw_2=0;1>
r69=<5;d_in=0;4;cfetch_1=0,cfetch_2=0;0>
r70=<5;d_in=1;4;cfetch_1=0,cfetch_2=0;0>
r71=<5;d_in=2;4;cfetch_1=0,cfetch_2=0;0>
r72=<5;d_in=0;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r73=<5;d_in=1;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r74=<5;d_in=2;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r75=<6;rst=0;0;cfetch_1=0,cfetch_2=0;1>
r76=<6;;4;cfetch_1=0,cfetch_2=0;0>
r77=<14;rst=0,c_in=0,cr_in=0;0;cfetch_1=0,cfetch_2=0;6>
r78=<14;rst=0,c_in=0;0;cfetch_1=0,cfetch_2=0;5>
r79=<14;rst=0,c_in=1;0;cfetch_1=0,cfetch_2=0;5>
r80=<14;rst=0,c_in=2;0;cfetch_1=0,cfetch_2=0;5>
r81=<14;rst=0,it=0;0;cfetch_1=0,cfetch_2=0;5>
r82=<14;rst=0;0;cfetch_1=0,cfetch_2=0;4>
r83=<14;cr_in=0,c_in=0;3;rw_1=0,rw_2=0;3>
r84=<14;it=0,c_in=0;3;rw_1=0,rw_2=0;2>
r85=<14;it=0,c_in=1;3;rw_1=0,rw_2=0;2>
r86=<14;it=0,c_in=2;3;rw_1=0,rw_2=0;2>
r87=<14;it=0;3;rw_1=0,rw_2=0;1>
r88=<14;c_in=0;14;cfetch_1=0,cfetch_2=0;0>
r89=<14;c_in=1;4;cfetch_1=0,cfetch_2=0;0>
```

```
r90=<14;c_in=2;14;cfetch_1=0,cfetch_2=0;0>
r91=<14;c_in=0;15;rw_1=0,rw_2=0;0>
r92=<14;c_in=1;5;rw_1=0,rw_2=0;0>
r93=<14;c_in=2;15;rw_1=0,rw_2=0;0>
r94=<14;c_in=0;16;d_1=0,d_2=0,rw_1=0,rw_2=0;0>
r95=<14;c_in=1;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r96=<14;c_in=2;16;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r97=<15;rst=0,d_in=0,dr_in=0;0;cfetch_1=0,cfetch_2=0;4>
r98=<15;rst=0,d_in=0;0;cfetch_1=0,cfetch_2=0;3>
r99=<15;rst=0,d_in=1;0;cfetch_1=0,cfetch_2=0;3>
r100=<15;rst=0,d_in=2;0;cfetch_1=0,cfetch_2=0;3>
r101=<15;rst=0;0;cfetch_1=0,cfetch_2=0;2>
r102=<15;dr_in=0,d_in=0;3;rw_1=0,rw_2=0;1>
r103=<15;d_in=0;14;cfetch_1=0,cfetch_2=0;0>
r104=<15;d_in=1;4;cfetch_1=0,cfetch_2=0;0>
r105=<15;d_in=2;14;cfetch_1=0,cfetch_2=0;0>
r106=<15;d_in=0;16;d_1=0,d_2=0,rw_1=0,rw_2=0;0>
r107=<15;d_in=1;6;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r108=<15;d_in=2;16;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r109=<16;rst=0;0;cfetch_1=0,cfetch_2=0;1>
r110=<16;;0;cfetch_1=0,cfetch_2=0;0>
r111=<7;rst=0,c_in=0,cr_in=0;7;cfetch_1=0,cfetch_2=0;6>
r112=<7;rst=0,c_in=0;7;cfetch_1=0,cfetch_2=0;5>
r113=<7;rst=0,c_in=1;7;cfetch_1=0,cfetch_2=0;5>
r114=<7;rst=0,c_in=2;7;cfetch_1=0,cfetch_2=0;5>
r115=<7;rst=0,it=0;7;cfetch_1=0,cfetch_2=0;5>
r116=<7;rst=0;7;cfetch_1=0,cfetch_2=0;4>
r117=<7;cr_in=0,c_in=0;10;rw_1=0,rw_2=0;3>
r118=<7;it=0,c_in=0;10;rw_1=0,rw_2=0;2>
r119=<7;it=0,c_in=1;10;rw_1=0,rw_2=0;2>
r120=<7;it=0,c_in=2;10;rw_1=0,rw_2=0;2>
r121=<7;it=0;10;rw_1=0,rw_2=0;1>
r122=<7;c_in=0;7;cfetch_1=0,cfetch_2=0;0>
r123=<7;c_in=1;11;cfetch_1=0,cfetch_2=0;0>
r124=<7;c_in=2;17;cfetch_1=0,cfetch_2=0;0>
r125=<7;c_in=0;8;rw_1=0,rw_2=0;0>
r126=<7;c_in=1;12;rw_1=0,rw_2=0;0>
r127=<7;c_in=2;18;rw_1=0,rw_2=0;0>
r128=<7;c_in=0;9;d_1=1,d_2=1,rw_1=0,rw_2=0;0>
r129=<7;c_in=1;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r130=<7;c_in=2;19;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r131=<8;rst=0,d_in=0,dr_in=0;7;cfetch_1=0,cfetch_2=0;4>
r132=<8;rst=0,d_in=0;7;cfetch_1=0,cfetch_2=0;3>
r133=<8;rst=0,d_in=1;7;cfetch_1=0,cfetch_2=0;3>
r134=<8;rst=0,d_in=2;7;cfetch_1=0,cfetch_2=0;3>
r135=<8;rst=0;7;cfetch_1=0,cfetch_2=0;2>
```

```
r136=<8;dr_in=0,d_in=0;10;rw_1=0,rw_2=0;1>
r137=<8;d_in=0;7;cfetch_1=0,cfetch_2=0;0>
r138=<8;d_in=1;11;cfetch_1=0,cfetch_2=0;0>
r139=<8;d_in=2;17;cfetch_1=0,cfetch_2=0;0>
r140=<8;d_in=0;9;d_1=1,d_2=1,rw_1=0,rw_2=0;0>
r141=<8;d_in=1;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r142=<8;d_in=2;19;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r143=<9;rst=0;7;cfetch_1=0,cfetch_2=0;1>
r144=<9;;7;cfetch_1=0,cfetch_2=0;0>
r145=<10;rst=0,d_in=0,dr_in=0;7;cfetch_1=0,cfetch_2=0;3>
r146=<10;rst=0,d_in=0;7;cfetch_1=0,cfetch_2=0;2>
r147=<10;rst=0,d_in=1;7;cfetch_1=0,cfetch_2=0;2>
r148=<10;rst=0,d_in=2;7;cfetch_1=0,cfetch_2=0;2>
r149=<10;rst=0;7;cfetch_1=0,cfetch_2=0;2>
r150=<10;dr_in=0,d_in=0;10;rw_1=0,rw_2=0;1>
r151=<10;d_in=0;7;cfetch_1=0,cfetch_2=0;0>
r152=<10;d_in=1;11;cfetch_1=0,cfetch_2=0;0>
r153=<10;d_in=2;17;cfetch_1=0,cfetch_2=0;0>
r154=<11;rst=0,c_in=0,cr_in=0;7;cfetch_1=0,cfetch_2=0;6>
r155=<11;rst=0,c_in=0;7;cfetch_1=0,cfetch_2=0;5>
r156=<11;rst=0,c_in=1;7;cfetch_1=0,cfetch_2=0;5>
r157=<11;rst=0,c_in=2;7;cfetch_1=0,cfetch_2=0;5>
r158=<11;rst=0,it=0;7;cfetch_1=0,cfetch_2=0;5>
r159=<11;rst=0;7;cfetch_1=0,cfetch_2=0;4>
r160=<11;cr_in=0,c_in=0;10;rw_1=0,rw_2=0;3>
r161=<11;it=0,c_in=0;10;rw_1=0,rw_2=0;2>
r162=<11;it=0,c_in=1;10;rw_1=0,rw_2=0;2>
r163=<11;it=0,c_in=2;10;rw_1=0,rw_2=0;2>
r164=<11;it=0;10;rw_1=0,rw_2=0;1>
r165=<11;c_in=0;11;cfetch_1=0,cfetch_2=0;0>
r166=<11;c_in=1;11;cfetch_1=0,cfetch_2=0;0>
r167=<11;c_in=2;11;cfetch_1=0,cfetch_2=0;0>
r168=<11;c_in=0;12;rw_1=0,rw_2=0;0>
r169=<11;c_in=1;12;rw_1=0,rw_2=0;0>
r170=<11;c_in=2;12;rw_1=0,rw_2=0;0>
r171=<11;c_in=0;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r172=<11;c_in=1;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r173=<11;c_in=2;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r174=<12;rst=0,d_in=0,dr_in=0;7;cfetch_1=0,cfetch_2=0;4>
r175=<12;rst=0,d_in=0;7;cfetch_1=0,cfetch_2=0;3>
r176=<12;rst=0,d_in=1;7;cfetch_1=0,cfetch_2=0;3>
r177=<12;rst=0,d_in=2;7;cfetch_1=0,cfetch_2=0;3>
r178=<12;rst=0;7;cfetch_1=0,cfetch_2=0;2>
r179=<12;dr_in=0,d_in=0;10;rw_1=0,rw_2=0;1>
r180=<12;d_in=0;11;cfetch_1=0,cfetch_2=0;0>
r181=<12;d_in=1;11;cfetch_1=0,cfetch_2=0;0>
```

```
r182=<12;d_in=2;11;cfetch_1=0,cfetch_2=0;0>
r183=<12;d_in=0;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r184=<12;d_in=1;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r185=<12;d_in=2;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r186=<13;rst=0;7;cfetch_1=0,cfetch_2=0;1>
r187=<13;;11;cfetch_1=0,cfetch_2=0>
r188=<17;rst=0,c_in=0,cr_in=0;7;cfetch_1=0,cfetch_2=0;6>
r189=<17;rst=0,c_in=0;7;cfetch_1=0,cfetch_2=0;5>
r190=<17;rst=0,c_in=1;7;cfetch_1=0,cfetch_2=0;5>
r191=<17;rst=0,c_in=2;7;cfetch_1=0,cfetch_2=0;5>
r192=<17;rst=0,it=0;7;cfetch_1=0,cfetch_2=0;5>
r193=<17;rst=0;7;cfetch_1=0,cfetch_2=0;4>
r194=<17;cr_in=0,c_in=0;10;rw_1=0,rw_2=0;3>
r195=<17;it=0,c_in=0;10;rw_1=0,rw_2=0;2>
r196=<17;it=0,c_in=1;10;rw_1=0,rw_2=0;2>
r197=<17;it=0,c_in=2;10;rw_1=0,rw_2=0;2>
r198=<17;it=0;10;rw_1=0,rw_2=0;1>
r199=<17;c_in=0;17;cfetch_1=0,cfetch_2=0;0>
r200=<17;c_in=1;11;cfetch_1=0,cfetch_2=0;0>
r201=<17;c_in=2;17;cfetch_1=0,cfetch_2=0;0>
r202=<17;c_in=0;18;rw_1=0,rw_2=0;0>
r203=<17;c_in=1;12;rw_1=0,rw_2=0;0>
r204=<17;c_in=2;18;rw_1=0,rw_2=0;0>
r205=<17;c_in=0;19;d_1=1,d_2=1,rw_1=0,rw_2=0;0>
r206=<17;c_in=1;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r207=<17;c_in=2;19;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r208=<18;rst=0,d_in=0,dr_in=0;7;cfetch_1=0,cfetch_2=0;4>
r209=<18;rst=0,d_in=0;7;cfetch_1=0,cfetch_2=0;3>
r210=<18;rst=0,d_in=1;7;cfetch_1=0,cfetch_2=0;3>
r211=<18;rst=0,d_in=2;7;cfetch_1=0,cfetch_2=0;3>
r212=<18;rst=0;7;cfetch_1=0,cfetch_2=0;2>
r213=<18;dr_in=0,d_in=0;10;rw_1=0,rw_2=0;1>
r214=<18;d_in=0;17;cfetch_1=0,cfetch_2=0;0>
r215=<18;d_in=1;11;cfetch_1=0,cfetch_2=0;0>
r216=<18;d_in=2;17;cfetch_1=0,cfetch_2=0;0>
r217=<18;d_in=0;19;d_1=1,d_2=1,rw_1=0,rw_2=0;0>
r218=<18;d_in=1;13;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r219=<18;d_in=2;19;d_1=2,d_2=2,rw_1=0,rw_2=0;0>
r220=<19;rst=0;7;cfetch_1=0,cfetch_2=0;1>
r221=<19;;7;cfetch_1=0,cfetch_2=0;0>
```

Appendix E

Publications

[1] A. Ferscha and Gy. Csertán. Sequential Simulation Engine for Distributed Timed Transition Petri Net Simulation. Technical report, Institut für Statistik und Informatik, Universität Wien, Lenaugasse 2/8, A–1080 Wien, Austria, May 1992.

[2] Gy. Csertán. Optimistic Distributed Discrete Event Simulation. Master's thesis, University of Vienna, Department of Statistics and Computer Science, Lenaugasse 2/8, A–1080 Wien, Austria, 1992.

[3] G. Haring, A. Ferscha, Gy. Csertán, P. Ferrara, B. Gruber, W. Müllner, and P. Ramoser. Parallel Petri Net Simulation. Final Report of the Research Project of the Austrian Ministery of Science and Research (GZ 613.525/2–26/90), December 1992.

[4] A. Ferscha and Gy. Csertán. Timed Transition Petri Net Simulation. In *Proceedings of the Conference on Parallel and Distributed Computing in Education*, pages 45–47, March Miskolc, Hungary, 1993.

[5] Gy. Csertán. Temporal Analisys of Dataflow Computational Paradigm Based Control Systems. Technical Report GyCs–LS 1/1993, Department of Information Engineering of the University of Pisa, Via Diotisalvi 2, I–56126 Pisa, Italy, 1993.

[6] Gy. Csertán. Dataflow Networks for Temporal Analysis of Control Systems. In *Proceedings of the 1994 MMT Mini-Symposium*, pages 15–17, January Budapest, Hungary, 1994.

[7] Gy. Csertán. Hibaterjedés vizsgálata digitális berendezésekben számítógépes szimulációval. Mérési útmutató a Hibatűrő számítógéparchitektúrák laboratóriumhoz, Budapesti Műszaki Egyetem, Műszer- és Méréstechnika Tanszék, 1994. (In Hungarian).

[8] 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.

[9] Gy. Csertán, C. Bernardeschi, and L. Simoncini. From Dataflow Networks to Petri Nets. In *Proceedings of the 8th Symposium on Microcomputers and Applications, uP'94*, pages 25–34, Budapest, Hungary, 1994.

[10] 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.

[11] Gy. Csertán. Testability Analysis of Data Flow Networks Modeled Computing Systems. In *Proceedings of the 1995 MMT Mini-Symposium*, pages 5–7, January Budapest, Hungary, 1995.

[12] B. Sallay, K. Tilly, A. Pataricza, Gy. Csertán, Z. Hegedűs, A. Petri, L. Surján, and J. Sziray. Applications of AI Methods in High-Level Test Generation. Technical Report FUTEG-3/1995, Department of Measurement and Instrument Engineering of the Technical University of Budapest, February, 1995.

[13] Gy. Csertán, A. Pataricza, and E. Selényi. Dependability Analysis in HW-SW codesign. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium, IPDS'95*, pages 316–325, April Erlangen, Germany, 1995.

[14] C. Bernardeschi, A. Bondavalli, Gy. Csertán, I. Majzik, and L. Simoncini. Temporal Analysis of Data Flow Real-Time Control Systems. Technical Report 1/1995, Department of Information Engineering of the University of Pisa, Via Diotisalvi 2, I–56126 Pisa, Italy, 1995.

[15] I. Turcsány, Gy. Csertán, and A. Pataricza. Model Based Diagnostic-Test Scheduling. In *Proceedings of the Technical Conference on CAD Methods in Electronic and Information Processing System Design and its Education*, pages 27–30, June Budapest, Hungary, 1995.

[16] 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, 1995.

[17] Gy. Csertán. System Diagnostics in HW-SW Codesign. In B.Straube and J. Schönherr, editors, *Proceedings of the 4th GI/ITG/GME Workshop*, pages 51–60, March Kreischa, Germany, 1996. ISBN 3–8265–1321–5.

[18] B. Antal, A. Bondavalli, Gy. Csertán, I. Majzik, and L. Simoncini. Reachability and Timing Analysis in Data Flow Networks: A Case Study. In *Proceedings of the 22nd Euromicro Conference*, pages 193–200, September Prague, Czech Republic, 1996.

[19] Gy. Csertán and A. Pataricza. On Diagnosability in HW-SW Codesign: A Case Study. In *Proceedings of the IEEE International Workshop on Embedded Fault-Tolerant Systems, EFTS'96*, September Dallas, USA, 1996.

[20] Gy. Csertán, A. Pataricza, and E. Selényi. Design for Testability with HW-SW Codesign. *Periodica Polytechnica*, 40(1):30–50, 1996.

[21] C. Bernardeschi, A. Bondavalli, Gy. Csertán, I. Majzik, and L. Simoncini. Temporal Analysis of Data Flow Control Systems. Submitted to Automatica.