Chapter 2

Modeling of Fault-Tolerant Computing Systems

2.1 Introduction

The availability of low-cost, but highly complex off-the-shelf programmable components (PLDs) 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 1:20 reduction in design time, for a price of an about 40% hardware overhead due to the automated design. Last but not least, the use of automated design technologies radically improves the product's design quality.

As applications include even safety related applications, like process control, transport and automation systems, 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, testability analysis.

In order to avoid costly re-design cycles, diagnostic design has to be pushed into early phases of system design as well. Diagnostic design provides the system's diagnostic plan, the aim of which is to locate the faults within the system. Unfortunately even those CAD tools, that support dependability analysis, are useless for automated diagnostic design. In [SS94] a method is presented for doing testability analysis as part of integrated diagnostics in early design phases, but the other problems of diagnostic design, such as test generation, test set design, and construction of the input model of integrated diagnostics, remain still unsolved.

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 it has to give support for the solution of modeling, model checking, test generation, fault simulation, and testability analysis related problems that arise during the design of fault-tolerant digital 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 testability analysis 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.
- The modeling approach must also support traditional, functional evaluations, e.g. validation, performance analysis, temporal analysis.
- Modeling of uncertainty has to be solved in the approach, since in early phases of a top-down 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 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.

This work focuses on the field of digital computing and control systems, where dependability is of primary concern. Emphasis is put on the first, early step of system design that consists of the iterative process of model construction, model evaluation and model refinement. Figure 2.1 shows the typical design flow in the suggested environment, the shaded area identifies the topics to which this work contributes.

The depicted design flow is an unfolded version of the design flow of hardware-sofware codesign [BBC+93], that is defined as the the joint specification, design, and synthesis of mixed hardware-software systems. HW-SW codesign is one of the most promising new design methodologies for embedded systems consisting of mixed hardware and software components. Nowadays HW-SW codesign is suitable for the design of highly complex systems, and not only for design of small embedded applications like a few years ago. Its main advantage lies in the support for design automation and design verification through the whole design cycle.

At the beginning of HW-SW codesing 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

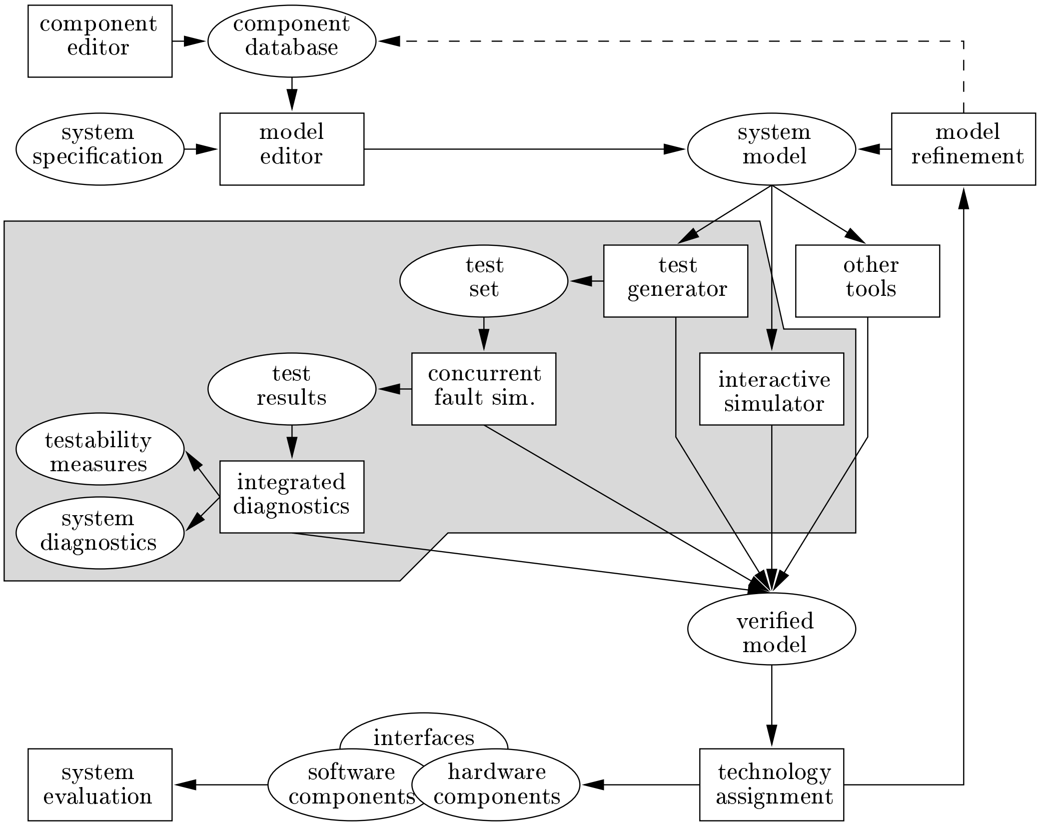

Figure 2.1: Design Flow in a Modern CAD Environment

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 automated synthesis can be started. The "in-loop" evaluation has to make sure, that the current model meets the requirements stated in the specification. In Figure 2.1 the evaluation step is shown from the point of view of our work. Finally, using the lowest-level model, technology assignment is done, that replaces the hardware-sofware separation step of earlier used design methods.

In our version of an integrated design environment, in the first step of system design the system is modeled by extended dataflow networks. (Reasoning about the advantages of dataflow modeling can be found in subsequent chapters.) At this level of abstraction 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.

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 tests set of the system.

In the second step, after all functional details became fully specified, system partitioning and automated hardware and software synthesis can be started. After technology assignment a back-annotation step has to be done. It feeds back information from the implemented system to the functional model, and assigns the functional components to HW and/or SW components. As a result, multiple behavioral faults, i.e. faults originating in the same single physical fault due to hardware reuse, can be taken into account in the model (two or more functional units are implemented by the same HW or SW component) and a refinement step can be done, that further restricts the nondeterminism within the model. Test generation and testability analysis is still possible, and the result of test generation is the low-level test set of the components, e.g. self test of a component.

Our approach is based on the idea of modeling the fault effects and their propagation similarly to the flow of data in the functional model. 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 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 this global overview of the system effectively supports test generation procedures by radically restricting the search space to the solutions of those from the coarse model [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.

2.2 The 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 [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$^+$].

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

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

approach are given, while the right column gives the advantages these properties provid. 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.

2.2.1 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 nondeterministic as showed by [Jon89b]. The latter formalism extended by the notion of time and priorities [Can93] is the starting point of our modeling approach.

2.2.2 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. 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.

2.2.3 Formalism of Dataflow Networks

Dataflow nodes are defined in form of 6-tuples that consists also 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 concise and more precise. 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*, $\pi \in I\!N$

$\emptyset$, 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.

Note that this notation of firing rules is not very user friendly, but it is part of the formalism. If the formalism is used 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 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 one firings have the same priority, then selection is done randomly. This causes a nondeterministic 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* $\emptyset$.

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

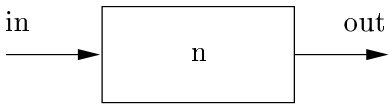

Figure 2.2: An Example Dataflow Network

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.

In Figure 2.2 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 ∅ 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.

2.2.4 Fault Modeling

The dataflow formalism itself suits 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).

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 logic. 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 unacceptable long evaluation times. As an example of 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 (in a small surrounding of the correct value)
- acceptable (within a given 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 has to be extended too.

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.

2.2.5 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 our approach uncertainty is modeled by utilizing the nondeterministic behavior of the proposed dataflow networks.

In digital systems one usually has to differentiate between operational nondeterminism and nondeterminism due to underspecification. The first term relates to the planned nondeterministic 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 neglect of data values.

It is desirable that this two types of nondeterministic behavior are handled in a uniform fashion by the modeling method. In many engineering approaches it is solved by using the notion of stochastic processes. This is a probabilistic method in which probability variables are assigned to the states and state transitions of a system. They express the likelihood that the system is in the given state or that the system executed the given action. The two tools, SURF-2 and Depend, are mentioned in Section 2.1 to be able to deal with uncertainty modeling. Both are based on stochastic processes. In this work we intentionally avoid to use 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.

Instead of using a probabilistic approach we suggest the following method. Operational nondeterminism and the effects of faults are expressed by the firing nondeterminism 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 nondeterminism of dataflow nodes, i.e. the value of the token sent by the firing is uncertain.

For this reason, we further extend the set of tokens, node states, and firing rules to describe uncertain behavior of the system. It means for example, that when a component is faulty and there is no information available whether it delivers error-free or erroneous results, 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.

A typical example of uncertainty caused by the simplification of control structures is the uninterpreted description of 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. function is described by two firing rules that has the same input mapping, pre-state, and priority.

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.

2.2.6 Features of the Approach

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 happen e.g. arrival of a token on 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 nondeterministically. This alternative is used in the above description of repair, while the first alternative is used in the examples thorough this work, in order to decrease the number of firings.

The dataflow modeling approach and the possibility to extract the conditional error propagation paths allows us to execute the following evaluation:

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 consturct 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

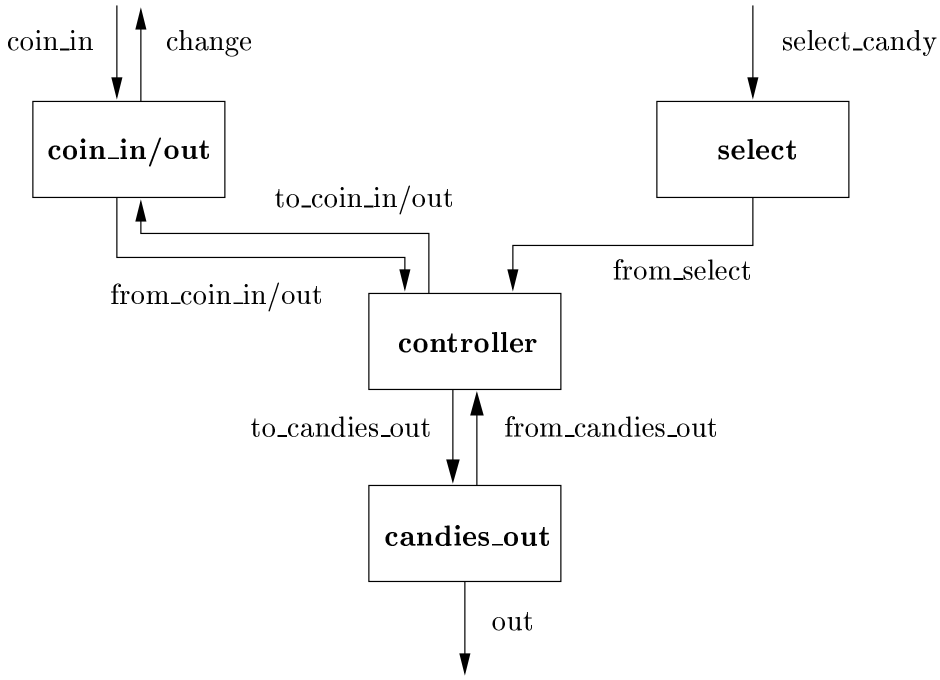

Figure 2.3: DFG of the Candy Automaton

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.

2.2.7 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 2.3, while the formal description of the network is given in Appendix A.1. A much more complex example is given in [Cse95]. 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.

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.

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.

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

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.

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:

$$\mathtt{r1} =< \mathtt{ok}; \mathtt{to\_candies\_out} = \mathtt{ok}; \mathtt{ok}; \mathtt{from\_candies\_out} = \mathtt{ok}, \mathtt{out} = \mathtt{ok}; 0 >$$
$$\mathtt{r2} =< \mathtt{ok}; \mathtt{to\_candies\_out} = \mathtt{inc}; \mathtt{ok}; \mathtt{from\_candies\_out} = \mathtt{ok}, \mathtt{out} = \mathtt{inc}; 0 >$$
$$\mathtt{r3} =< \mathtt{ok}; \mathtt{to\_candies\_out} = \mathtt{x}; \mathtt{ok}; \mathtt{from\_candies\_out} = \mathtt{ok}, \mathtt{out} = \mathtt{x}; 0 >$$

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:

$$\mathtt{r4} =< \mathtt{ctrl}; \mathtt{to\_candies\_out} = \mathtt{ok}; \mathtt{ctrl}; \mathtt{from\_candies\_out} = \mathtt{ok}, \mathtt{out} = \mathtt{inc}; 0 >$$
$$\mathtt{r5} =< \mathtt{ctrl}; \mathtt{to\_candies\_out} = \mathtt{inc}; \mathtt{ctrl}; \mathtt{from\_candies\_out} = \mathtt{ok}, \mathtt{out} = \mathtt{x}; 0 >$$
$$\mathtt{r6} =< \mathtt{ctrl}; \mathtt{to\_candies\_out} = \mathtt{x}; \mathtt{ctrl}; \mathtt{from\_candies\_out} = \mathtt{ok}, \mathtt{out} = \mathtt{x}; 0 >$$

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:

$$\mathtt{r7} =< \mathtt{stk}; \mathtt{to\_candies\_out} = \mathtt{ok}; \mathtt{stk}; \mathtt{from\_candies\_out} = \mathtt{dead}, \mathtt{out} = \mathtt{dead}; 0 >$$
$$\mathtt{r8} =< \mathtt{stk}; \mathtt{to\_candies\_out} = \mathtt{inc}; \mathtt{stk}; \mathtt{from\_candies\_out} = \mathtt{dead}, \mathtt{out} = \mathtt{dead}; 0 >$$
$$\mathtt{r9} =< \mathtt{stk}; \mathtt{to\_candies\_out} = \mathtt{x}; \mathtt{stk}; \mathtt{from\_candies\_out} = \mathtt{dead}, \mathtt{out} = \mathtt{dead}; 0 >$$

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.

2.3 Model Refinement

It is well accepted now that the development of distributed 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 bibliography, 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.

2.3.1 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 nondeterministic 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 sub-network, 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.

2.3.2 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.

2.3.3 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:

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

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

Definition 2.4 *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.

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:

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

The transformation $\mathcal{R}$ of the example is a domain refinement, based on Definition 2.4. If, for example, $\hat{r}4$ is modified to $\hat{\mathtt{r}}4 =< \mathtt{off}; \mathtt{in} = \mathtt{ab}; \mathtt{on}; \mathtt{out} = \mathtt{bb}; 0 >$, $\mathcal{R}$ 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:

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

The transformation $\mathcal{R}$ of the example is a domain refinement, based on Definition 2.4. 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 $hot \notin \mathcal{R}(good)$.

2.3.4 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 2.5 *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* $fs = s \overset{r_i}{\rightsquigarrow} s_x \dots s_y \overset{r_j}{\rightsquigarrow} s'$*, 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 2.6 *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 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.

Example: structure refinement (see Figure 2.4)

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:

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

The transformation $\mathcal{R}$ of the example is a structure refinement, based on Definition 2.6. If, for example, $r_{n_2}2$ is changed to $\mathtt{r_{n_2}2} =< \mathtt{good; int = b; good; out = a; 0} >$, $\mathcal{R}$ is not a structure refinement any more, since the firing sequence of DFN that corresponds r_n2 would map $a \mapsto out$ instead of $b \mapsto out$.

2.3.5 Refinement in the Modeling Approach

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

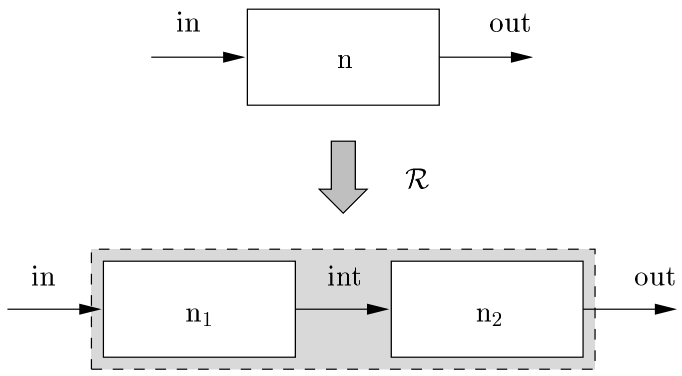

Figure 2.4: Example of Structure Refinement

levels of modeling can be expressed by means of refinement, and how uncertainty modeling can influence the rules of refinement.

Model Changes and Dataflow 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 should 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.

Multi-Level Modeling

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. Refinement within a given level (hierarchical modeling) is already described in previous sections. In the following refinement between 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 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 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, manual interaction is necessary in many cases. 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.

Resource allocation and its effects can be presented in the 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. (For an example of resource allocation refer to the next section.)

These changes of the model effect only the structure of the dataflow network, and leave its behavior unaffected. Therefore it is not a refinement, and can not be described with the rules of refinement, but the 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.

Effects of Uncertainty on 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.

2.3.6 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 2.5, while the complete formal notation is presented in Appendix A.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 2.3) 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.

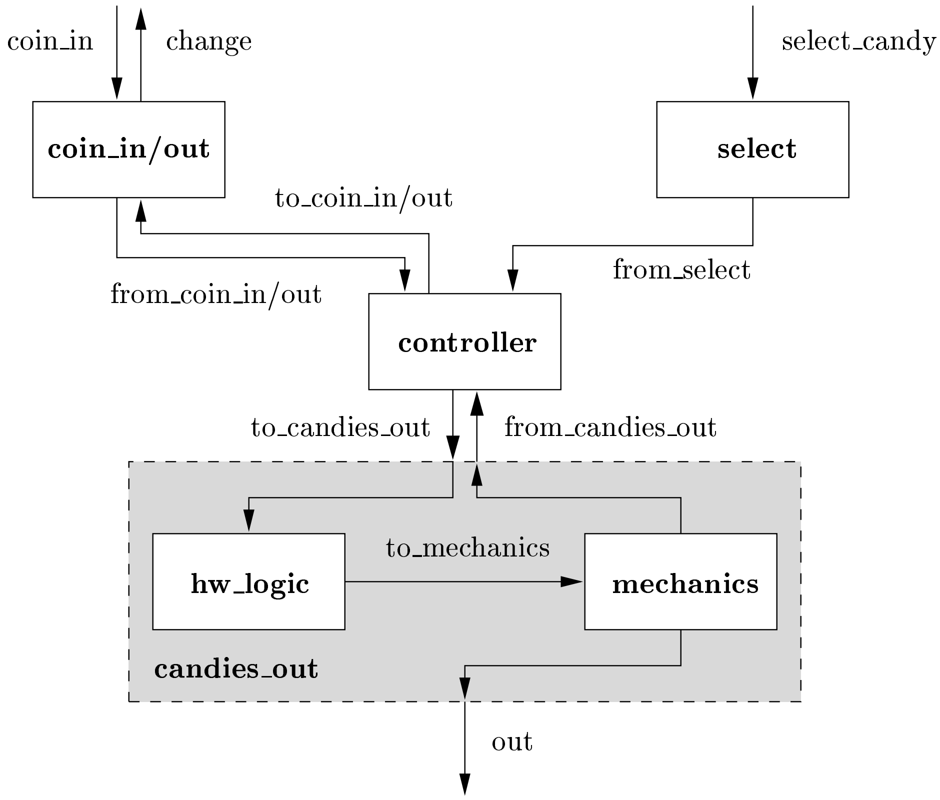

Figure 2.5: DFG of the Refined Candy Automaton

component:	error
hwl	ok ctrl
mechanics	ok stk

Table 2.3: Set of States of the Components After Refinement

mechanics A candy may stuck in the mechanics. It leads to state `stk` where the component can not deliver candies any more.

The fault model of the system is the same as before refinement (refer to Section 2.2). Therefore, the token set of the nodes is unchanged.

Appendix A

A.1 Formal Description of the Candy Automaton

In this section the formal definition of the candy automaton is given. Similarly to Section 2.2, 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 2.3 in case of the original automaton and to Figure 2.5 in case of the refined automaton.

A.1.1 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)}

A.1.2 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>
```

A.1.3 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>
```

A.1.4 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>
```

A.1.5 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>
```

A.1.6 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)}

A.1.7 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>
```

A.1.8 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>
```

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.

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

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

[BBC+] C. Bernardeschi, A. Bondavalli, Gy. Csertán, I. Majzik, and L. Simoncini. Temporal Analysis of Data Flow Control Systems. Accepted to Automatica.

[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.

[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.

[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.

[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.

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

[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.

[Cse95] 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. In Press.

[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.

[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.

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

[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.

[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.

[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.

[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.

[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.

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

[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.

[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.

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

[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.

[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.