ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2006, Vol.35, No.2

FAULT TOLERANT PROCESS NETWORKS

Jonas Čeponis, Egidijus Kazanavičius, Antanas Mikuckas

Department of Computer Engineering, Kaunas University of Technology
Studentu st. 50-213, LT-51368 Kaunas, Lithuania

Abstract. Kahn process network is a kind of data flow process networks. It is a computation model in which many concurrent processes communicate through unbounded FIFO buffer and can be executed simultaneously. In real time digital signal processing applications execution time is infinite. However, failures of implementation hardware can occur. In our work, dynamic run-time reconfiguration is introduced into process network which ensures error handling, avoiding deadlocks, continuous and on-time result delivery. After dynamic reconfiguration, network execution results may become non deterministic, but this helps avoiding critical termination of network execution. In this paper, we present a description of possible failures of network execution and discuss the means for avoiding these failures.

Keywords: Process Networks, Data Flow, Dynamic Reconfiguration, Concurrent Processes, Labelled Transition Systems.

1. Introduction

Real time data flow processing is frequently used in digital signal processing (DSP), multimedia and control applications. Depending on implementation, the application may be control oriented and/or event driven. Such systems respond to changes of internal states or environment. In response to these changes, the system must reconfigure itself. A well-designed system should always have a powerful mechanism for handling error conditions. Reconfiguration can be used in ensuring error proof ness of the system.

Error-proof data flow applications should use general computational model for design, modelling, analysis and implementation in various platforms. Such computational models are Synchronous Data Flow (SDF) [13], Parameterized Synchronous Data Flow (PSDF) [12], Discrete-Event (DE) [12], Kahn Process Network (KPN) [2, 9], etc.

Kahn Process Network is a subset of more general Process Network (PN) model. PN consists of concurrent processes communicating over first-in first out unidirectional queues. PN is useful for modelling and exploiting functional parallelism in streaming data applications. The PN model maps easily onto multiprocessor and/or multi-threaded targets.

Synchronous Data Flow network is useful for modelling simple dataflow systems without complicated flow of control. In SDF, nodes (processes) read and write a fixed number of tokens each time they are executed. The number of tokens and execution time are defined during system design. Process Network differs significantly from SDF, as PN uses completely dynamic execution of nodes. In PN, nodes are executed asynchronously, but the result of network execution is deterministic. Determinism is ensured by blocked reading from channel.

Parameterized Synchronous Data Flow is useful for modelling dataflow systems with reconfiguration. PSDF represents a design point between complete static scheduling in SDF and completely dynamic execution in Process Networks.

Discrete-Event model differs from other discussed models as it supports time-oriented models of systems such as queuing systems, communication networks and digital hardware.

KPN ensures completely dynamic execution of nodes. In KPN, there is no need for describing scheduling during system design, because scheduling does not affect the functional behaviour of the nodes. These features of KPN are very useful in dynamic network reconfiguration. In this work, Error-Proof Process Network (EPPN) model is presented, based on Kahn process network model. Because of dynamic reconfiguration used in EPPN, execution results may become non deterministic, but this helps avoiding critical termination of network execution.

DSP real time systems often require non-standard and costly hardware and software solutions. Modern workstation can represent an alternative to develop real time intensive signal processing applications. Furthermore, the programming model of Process Network corresponds completely to this kind of applications and fits perfectly on multiprocessor systems. Various PNs can be modelled using personal computer. This helps in finding and processing critical network operation points. Performance can be improved

dynamically changing network parameters. On the other hand, hardware systems (embedded systems, programmable logic) dedicated to a particular problem class can be used to minimize processing time. Process network implementation in the hardware system would help in verifying efficiency of the solution in the particular situation [10].

The remainder of this paper is structured as follows. The next section discusses related works. This is followed by the formal definition of Error-Proof Process Network model. Next, implementation issues are discussed. In section 5, we conclude and present suggestions for future.

2. Related work

Process networks and dataflow networks are a good model of computation for streaming multimedia and digital signal processing applications [14]. The most popular models for streaming applications are Synchronous Data Flows and Kahn Process Networks.

Synchronous Data Flow network is applicable to simple dataflow systems without complicated flow of control. In SDF, a node produces and consumes a fixed number of data tokens on each of its outgoing and incoming channels during each activation. For activation of the node it must have at least as many tokens on its input channels as it needs to consume. The number of tokens and execution times must be defined during system design [6].

KPN is a computation model in which many concurrent processes communicating over first-in first-out unidirectional queues can be executed simultaneously. KPN is mostly applicable for parallel processing of streaming data. Many of the existing KPN modelling tools (Ptolemy [12], YAPI [10]) are commonly used for simulation rather than implementation. In analyzed dataflow process networks modelling methods, two important issues are considered: detection of critical execution points and dynamic network reconfiguration.

An important task of the system designer is to determine which parts of the application can be implemented in software and which parts are more critical and, hence, should be performed by dedicated hardware. To help the designer determine the critical parts of the system, in [7, 8] an algorithm for determining the relative criticality of processes in Kahn Process Networks is presented.

Since the PN model is Turing complete, memory requirements cannot be predicted statically. In general, any bounded-memory scheduling algorithm for this model requires run-time deadlock detection. The few PN implementations that perform deadlock detection detect only global deadlocks. Not all local deadlocks, however, will cause a PN system to reach global deadlock. Olson and Evans [17] presented local deadlock detection algorithm for PN models based on the Mitchell and Merritt algorithm. Their algorithm is suitable for both parallel and distributed PN implementations.

In order to capture the interaction between input events and execution units as well as reconfiguration in dynamic stream processing, Reactive process Networks (RPN) are introduced [4]. The foundation for RPN was laid by efforts to integrate dataflow model and its reactive behaviour [5, 11]. Reactive behaviour in these models is commonly specified using hierarchical state machines.

Another means for specifying dynamic network reconfiguration during run-time is parameterizable SDF model [1, 3]. In PSDF, node execution is characterized by iterations that fire subprocesses in a particular order. Node execution can be reconfigured between iterations at run-time.

Yet another approach for dynamic process network reconfiguration is presented in the work of Neuendorffer and Lee [16]. It is concentrated on reconfiguration as a particular kind of event handling. The states of the network, when reconfigurations are allowed, are named quiescent states. The FIFO channel communication is used for sending and receiving events or parameters and for dividing input ports into streaming input ports and parameter input ports. This work focuses primarily on reconfiguration of SDF networks. This work along with the other discussed works on dynamic reconfiguration focuses on using reconfiguration for increasing efficiency.

Our work differs from the above-discussed approaches because it presents another point of view to the purpose of process network dynamic reconfiguration. This point of view is based on the idea that we must specify critical moments in network execution in order to avoid critical termination. These critical moments appear when communication between network nodes is broken or node fails. Error-Proof Process Network proposed in this work can be reconfigured dynamically in order to capture and process critical moments of network execution.

3. An Operational Model of Fault Tolerant Process Network

There are two common approaches for describing dataflow process network: denotational and operational. In PN, every node consumes input tokens and computes a functional result, which is then written to its outputs. The denotational interpretation of the network can be derived from the composition of these functions of the nodes. This approach is commonly used to capture the intended functionality of a process network or to define the functional semantics of a system or programming language implementing process networks, without specifying unnecessary implementation details.

When network is characterized operationally, processes read tokens from channels or write tokens to channels and perform computations in the mean time.

The channels that connect processes store tokens that are in transit from one process to another. The operational semantics has means for describing network implementation details, such as required buffer capacities, priorities of the nodes, deadlock detection.

In our work, the operational semantics is used for Error-Proof Process Network. The specification is presented in the form of Labelled Transition Systems (LTS).

We start with some common definitions and notations for process network. For FIFO channel specification we assume the universal finite set of channels CH and for every channel $c \in CH$ there is a corresponding finite channel alphabet Σ. Each channel $c \in CH$ is described by its length L and pointer $c(p)$ which refers to the last data record in the channel. The actions for data transfer through the channel are $c \to a, c \leftarrow a \mid c \in CH, a \in \Sigma$. The action $c \leftarrow a$ denotes input of data into channel. The action $c \to a$ denotes output of data from channel. These actions form the set of actions for data transfer $Ac = \{c \to a, c \leftarrow a \mid c \in CH, a \in \Sigma\}$. The channel can be in one of states sc_n during network execution. The set $Sc = \{sc_{\perp}, sc_w, sc_{cf}, sc_{ce}, sc_{pop}, sc_{push}, sc_{inc}, sc_{dec}\}$ defines all possible states of the channel. After setting initial parameters in state $sc_{\perp}$, channel transits to waiting state sc_w. In this state channel waits for requests from the nodes. When request from writing node arrives, channel moves to state sc_{cf}, in which it checks the availability of free space in channel memory. If there is a free space in memory, channel moves to the state sc_{push}. During this state the incoming token is written to the channel. When request from reading node arrives, channel moves to state sc_{ce}, in which it checks the availability of data in channel. If there are at least one data token, channel moves to the state sc_{pop} and send first data token to reading node. When the token from the writing node arrives and channel is full, channel transits to the state sc_{inc} in which the length of the channel is increased. The amount of memory used for increasing channel length depends on chosen strategy for memory allocation. Let's say we have a network with K channels and we can allocate M amount of memory in this network. The possible strategies for memory allocation are:

- all channels get M/K amount of memory;
- channel length is estimated according to expected data intensity;
- channel length is alternating according to network state.

When channel length increasing is successful it moves to state sc_{push}, otherwise it moves to state sc_w. The channel can transit to state sc_{dec} when a request from reading node arrives and other channels require to be increased. In state sc_{dec}, the length of the channel is decreased thus freeing memory for other channels.

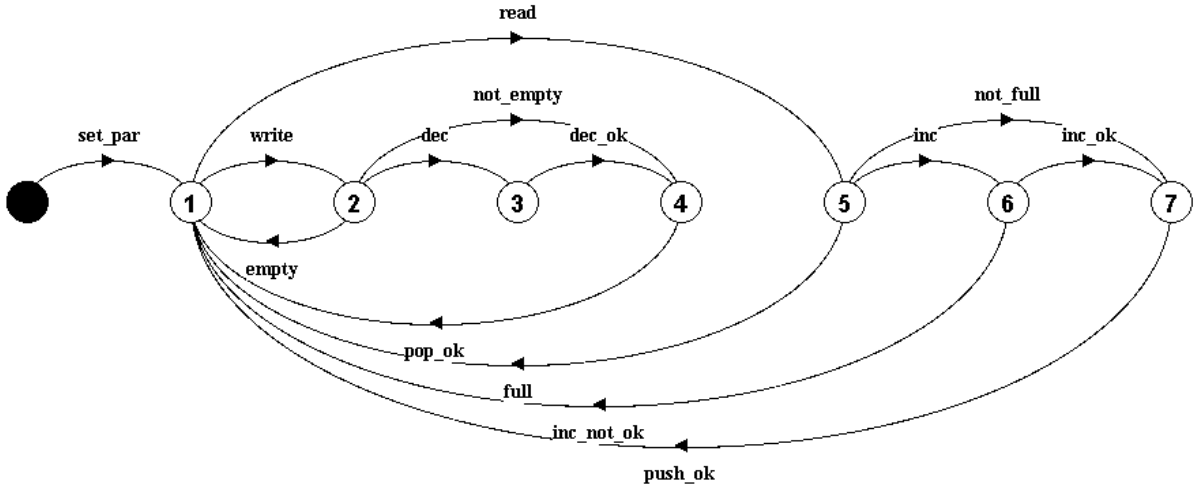

Figure 1. The states of channels in EPPN

Sequences of actions in the states of channel are described textually as finite state processes (FSP) and displayed and analysed by the LTSA analysis tool (presented in Figure 1) [15].

```
Channel = Initial,
Initial = (set_par -> Waiting),
Waiting = (read -> CheckFull | write
-> CheckEmpty),
CheckFull = (full -> Waiting |
not_full -> Push | inc -> Increase),
Push = (push_ok -> Waiting),
Increase = (inc_ok -> Push |
inc_not_ok -> Waiting),
CheckEmpty = (empty -> Waiting |
not_empty -> Pop | dec -> Decrease),
Decrease = (dec_ok -> Pop ),
Pop = ( pop_ok -> Waiting).
```

For network nodes specification, we use a universal finite set of nodes N and for every node of this set $n \in N$ there is a corresponding set of atomic actions Act. All actions of all network nodes are defined by the set A and the actions of every node $Act \subseteq A$. Every node has a set of input and output channels $(c_{in}, c_{out}) \in CH$. The node can be in one of the states ns_n (presented in Figure 2).

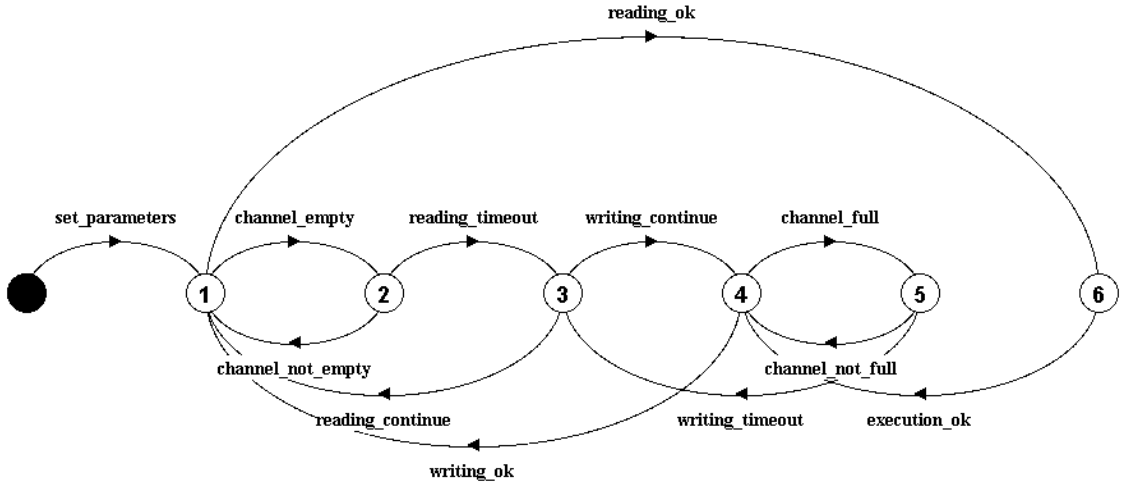

Figure 2. The states of nodes in EPPN

$Ns = \{ns_\perp, ns_r, ns_{br}, ns_w, ns_{bw}, ns_{ex}, ns_c\}$ is the set of states, which defines all possible states of the network node. Initial node state $ns_\perp$ denotes the starting point of node execution. In this state, initial working parameters and values of variables are set for the node. Afterwards, the node moves to reading state ns_r and tries to read data from input channels $c_{in} \in CH$. If at least one $c(p)=null \mid c \in c_{in}$, the node transits to state ns_{br} and waits until data are available in the channel. When the node has successfully read data from input channels, it moves to state ns_{ex} and executes actions $Act \subseteq A$. After finishing execution, the node transits to writing state ns_w and writes results to its output channels $c_{out} \in CH$. If at least one $c(p) \neq null \mid c \in c_{out}$, $p=L$, the node moves to blocked writing state ns_{bw} and waits for available free space in the output channel. When the node has finished writing data to channels, it moves to state ns_r.

The change of network execution parameters may be required in two cases:

- a node is in blocked reading or blocked writing states (nsbr or nsbw) and timeout occurs;
- external request is received.

In any of these two cases the node moves to control state ns_c and changes required parameters. Afterwards, the node transits to reading or writing state (ns_r or ns_w) and continues execution.

Sequences of actions in the states of node are described textually as finite state processes and displayed and analysed by the LTSA analysis tool (presented in Figure 2).

```
Node = Initial,
Initial = (set_parameters ->
Reading),
Reading = (reading_ok -> Execution |
channel_empty -> BlockedReading),
Execution = (execution_ok ->
Writing),
Writing = (writing_ok -> Reading |
channel_full -> BlockedWriting),
BlockedReading = (channel_not_empty
-> Reading | reading_timeout ->
Control),
BlockedWriting = (channel_not_full -
> Writing | writing_timeout ->
Control),
Control = (reading_continue ->
Reading | writing_continue ->
Writing).
```

The operational semantics of EPPN is based on the elements described above and will be presented in the form of labelled transition system (LTS).

DEFINITION. (Labelled Transition System) A labelled transition system is a tuple $(S, \hat{s}, I, O, A, M, \rightarrow)$ consisting of countable set of states S, initial state $\hat{s} \in S$, set of input channels $I \subseteq CH$, set of output channels $O \subseteq CH \setminus I$, set of input $(\{c \leftarrow a \mid c \in CH, a \in \Sigma\} \subseteq Ac)$ and output $(\{c \rightarrow a \mid c \in CH, a \in \Sigma\} \subseteq Ac)$ actions $A \subseteq Ac$, set of network nodes $M \subseteq N$ and labelled transition relation $\rightarrow \subseteq S \times A \times S$.

Labelled transition $\rightarrow$ is used to define atomic actions (reading data from channel, writing data into channel, processing data in the node). A 3-tuple $(s_i, \alpha, s_{i+1}) \in \rightarrow$ is said to be a transition and is usually written as $s_i \xrightarrow{\alpha} s_{i+1}$ which denotes that LTS in state s_i can perform action α which brings it to state s_{i+1} Action α can be atomic action or composite action consisting of several atomic actions. Transition from state s_i caused by action α is possible only if there is some $s_{i+1} \in S$ such that $s_i \xrightarrow{\alpha} s_{i+1}$.

DEFINITION. (Data transmission) Data transmission in LTS is a sequence $\hat{s} \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots s_i \xrightarrow{\alpha_i} s_{i+1}$, where $s_n \in S$, $\alpha_n \in A$ and $i \geq 0$ denotes execution time.

Given a data transmission in LTS with actions $\alpha = \alpha_0 \cdot \alpha_1 \cdot \ldots$, we can find out all input and output

data values in set of channels $C \subseteq CH$. For each channel $c \in C$, $c \leftarrow a$ is a bounded or unbounded (according to execution time) input sequence consisting of elements from the set Σ. These elements are acquired by mapping the actions α to reading actions in channel c.

4. Implementing EPPN

Real time digital signal processing applications implemented using process networks are becoming more dynamic, often requiring run-time reconfigurations of network execution parameters or even its structure. These issues are especially important if network is implemented in a distributed multicomputer system [6]. In our work, we are using a multi-threaded implementation of EPPN but we believe that implementation of EPPN is applicable to multicomputer system made up of several independent computers interconnected by a telecommunications network. An important problem in a distributed system emerges when we have to deal with hardware failures. Any failure in network node or channel causes global deadlock and network execution terminates. Faulty network node does not read data from its input channels and does not write data to its output channels. In such situation, preceding network node is blocked because its output channel (which is also the input channel of the faulty node) gets full. This leads to chain reaction of blocking all network nodes preceding the faulty one. The network node succeeding faulty node is also blocked as it cannot read data from its input channel (which is also the output channel of the faulty node). This also leads to chain reaction of blocking all network nodes succeeding the faulty node. Thus global deadlock occurs and execution of the whole network terminates. Analogical situation can be observed in case of network channel failure. The network nodes connected by faulty channel are blocked which leads to chain reaction of blocking all network nodes. Run-time network reconfiguration can be used for avoiding such situations.

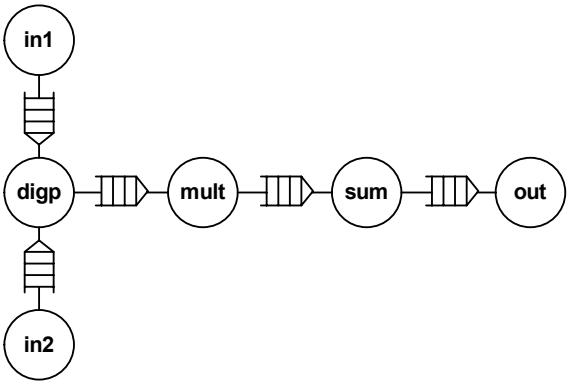

Figure 3. FIR filter Kahn process network model

In our work, we are proposing to use run-time reconfiguration for solving problems with network hardware failures. The further presented process network example of FIR filter is implemented using parallel programming and based on EPPN specification. The initial FIR filter process network model is presented in Figure 3.

In case of network node failure we need to redistribute its actions to other nodes. We also need to redistribute the input and output channels of the faulty node. Data loss occurs each time of network element failure. To solve this problem, we introduce the default value to compensate lost data. This conflicts process network determinism feature but enables further execution and helps to synchronize data.

In order to minimize data loss, the change of network parameters must follow these rules:

1. All nodes connected with faulty node *n* perform their actions until:
   a. input channels of the faulty node become full;
   b. output channels of the faulty node become empty.
2. The nodes connected with faulty node *n* transit to blocked reading or blocked writing states.
3. After timeout the nodes connected with faulty node *n* transit to control state. In this state:
   a. actions of the faulty node n are transferred to node *n1* which first transits to control state and is connected to output channel of *n*;
   b. the channel between n and n1 is destroyed;
   c. output channels of n are connected to n1;
   d. input channels of n are connected to n1;
   e. the data lost during failure are compensated.
4. The nodes which are in control state move to writing or reading states and reconfigured network continues execution.

These rules are applicable to network nodes which have input and output channels $(c_{in}, c_{out}) \subseteq CH$. There are two exceptions when these rules cannot be applied. The first one is when the faulty node does not have input channels ($c_{in}=\varnothing$). The reconfiguration in this situation should follow these rules:

1. All nodes connected with faulty node n perform their actions until output channels of the faulty node become empty.
2. The nodes connected with faulty node n transit to blocked reading states.
3. After timeout the nodes connected with faulty node n transit to control state. In this state:
   a. actions of the faulty node n are transferred to node n1 which first transits to control state and is connected to output channel of n;
   b. the channel between n and n1 is destroyed;
   c. output channels of n are connected to n1;
   d. the data lost during failure are compensated.
4. The nodes which are in control state move to writing or reading states and reconfigured network continues execution.

The second exception is when the faulty node does not have output channels ($c_{out} = \varnothing$). The reconfiguration in such case should follow these rules:

1. All nodes connected with faulty node *n* perform their actions until input channels of the faulty node become full.
2. The nodes connected with faulty node *n* transit to blocked writing states.
3. After timeout the nodes connected with faulty node *n* transit to control state. In this state:
   a. actions of the faulty node n are transferred to node n1 which first transits to control state and is connected to input channel of n;
   b. input channels of n are connected to n1;
   c. the data lost during failure are compensated.
4. The nodes which are in control state move to writing or reading states and reconfigured network continues execution.

In order to demonstrate the actions taken in case of node failure we are going to analyze the FIR filter process network model (Figure 3). Suppose network node *mult* fails. A possible solution for such situation is presented in Figure 4. The actions of faulty node *mult* are transferred to the modified node *multsum*, the input channels of *mult* become input channels of *multsum* and the channel connecting *mult* and *sum* is destroyed.

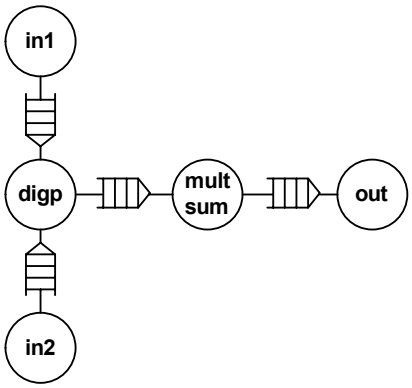

Figure 4. Modified FIR filter Kahn process network model

In distributed process network, channel failure is also critical for network execution and can cause a global deadlock. In case of network channel failure, the change of network parameters must follow these rules:

1. The nodes connected by faulty channel transit to blocked reading and writing states ns_{br} and ns_{bw}.
2. After timeout the nodes connected by faulty channel transit to control state ns_c.
3. Creation of a new channel is initiated by the node which first transits to control state.
4. The new communication channel is connected to the reading node as its input channel.
5. The new communication channel is connected to the writing node as its output channel.
6. The data lost during failure are compensated.
7. The nodes which are in control state move to writing or reading states and reconfigured network continues execution.

In order to analyze the behaviour of process network in case of hardware failure we are using a multi-threaded implementation of EPPN. IIR and FIR filters process network models were chosen for testing implementation. The example process networks were implemented in C# programming language using multiple threads running at the same time and performing different tasks of process network. We used separate thread for each element of the process network (node and channel) and main program for coordination. The failures of network elements were imitated by destroying a thread of node or channel.

5. Conclusion and future work

Process networks are frequently used in streaming multimedia applications. In real time digital signal processing applications, execution time is infinite. However, failures of implementation hardware can occur. In our work, dynamic run-time reconfiguration is introduced into process network which ensures handling of hardware failures, avoiding deadlocks, effective utilization of available resources, continuous and on-time result delivery.

The Error Proof Process Network presented in this paper is modelled as a labelled transition system. Formal specification of EPPN and the states of network elements during execution is presented. This specification was used as a base for describing the rules for network run-time reconfiguration in case of network element failure. These rules minimize data loss and enable further execution of EPPN. Multi-threaded implementation of EPPN was used to analyze the behaviour of process network in case of hardware failure. The failures of network elements were imitated by destroying a thread of a node or channel.

Future works include implementation of EPPN in a distributed multicomputer system. Attention will also be focussed on data loss compensation algorithms.

References

[1] **B. Bhattacharya, S. Bhattacharyya.** Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing,* 49(10), 2001, 2408-2421.

[2] **J. Čeponis, E. Kazanavičius, A. Mikuckas.** Design and analysis of DSP systems using Kahn process networks. *Ultragarsas*, 2002, *Vol*.45, ISSN 1392-2114, 43-46.

[3] **M. Dyer, M. Platzner and L. Thiele.** Efficient Execution of Process Networks on a Reconfigurable Hardware Virtual Machine. *Proceedings of the* 12*th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'04), 2004.

[4] **M. Geilen, T. Basten.** Reactive Process Networks. *EMSOFT'04,* 2004.

[5] **A. Girault, B. Lee, E. Lee.** Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems,* 18(6), 1999, 742-760.

[6] **M. Goel**. Process networks in Ptolemy II. *Technical Memorandum UCB/ERL No.*M98/69, *University of California, EECS Dept.*, 1998.

[7] **D. Hofstee.** Exploring Criticality Numbers for Kahn Process Networks. *MSc Thesis*, 2003.

[8] **D. Hofstee, B.H.H. Juurlink.** Determining the criticality of processes in Kahn process networks for design space exploration, *Proceedings ProRISC* 2002, *Veldhoven, The Netherlands*, 2002, 292-297.

[9] **G. Kahn.** The semantics of a simple language for parallel programming. *Information Processing* 74: *Proc. of the IFIP Congress* 74, 1974, 471-475.

[10] **E. Kock.** YAPI: Application modeling for signal processing systems. *In Proc. of the* 37*th. Design Automation Conference*, *IEEE*, 2000, 402-405.

[11] **B. Lee.** Specification and Design of Reactive Systems. *PhD thesis, Electronics Research Laboratory, University of California, EECS Dept.*, 2000.

[12] **E. Lee.** Overview of the Ptolemy project. *Technical Memorandum UCB/ERL No.*M01/11, *University of California, EECS Dept.*, 2001.

[13] **E. Lee, D. Messerschmitt.** Synchronous data flow. *IEEE Proceedings,* 75(9), 1987, 1235-1245.

[14] **E. Lee, T. M. Parks.** Dataflow process networks. *Proceedings of the IEEE,* 83(5), 1995, 773-798.

[15] **J. Magee, J. Kramer.** Concurrency: State Models & Java Programs. *John Wiley & Sons*, 1999.

[16] **S. Neuendorffer, E.A. Lee.** Hierarchical reconfiguration of dataflow models. *In Proc. Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE* 2004*), IEEE Computer Society Press*, 2004.

[17] **A.G. Olson, B.L. Evans.** Deadlock detection for distributed process networks. *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Proc., Vol.*5, 2005, 73-76.

Received January 2006.

DOI: 10.5755/j01.itc.35.2.12041