PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 39, NO. 1, PP. 3–26 (1995)

SIMULATION-BASED PERFORMABILITY ANALYSIS OF MULTIPROCESSOR SYSTEMS

Axel HEIN and Wolfgang HOHL

Institute for Computer Science III (IMMDIII)
University of Erlangen-Nürnberg
Martensstr. 3
D - 91058 Erlangen
e-mail: alhein@immd3.informatik.uni-erlangen.de

Received: Dec 16, 1994

Abstract

The primary focus in the analysis of multiprocessor systems has traditionally been on their performance. However, their large number of components, their complex network topologies, and sophisticated system software can make them very unreliable. The dependability of a computing system ought to be considered in an early stage of its development in order to take influence on the system architecture and to achieve best performance with high dependability. In this paper a simulation-based method for the combined performance and dependability analysis of fault tolerant multiprocessor systems are presented which provide meaningful results already during the design phase.

Keywords: fault tolerant multiprocessor systems, performability analysis, object-oriented model design, process-oriented simulation.

1. Introduction

The *performability analysis* of multiprocessor systems is a complex and important issue. This class of computing systems designed for the computation of time-consuming numerical algorithms has to provide *fault tolerance mechanisms* in order to tolerate the failure of components and to continue working even with reduced performance. The probability of a component failure increases with increasing number of components and the whole system would be useless if every component failure caused a failure of the complete system.

Performance and dependability can be analysed via various methods. *Measurements* based on benchmarking or monitoring are only possible when the real machine or a prototype already exists. In this late phase of the design cycle the hardware and major parts of the system software are fixed and cannot easily be changed. Therefore, it is difficult to correct weak points and bottlenecks. In order to exert an influence on the system design the performance and dependability analysis has to take place as soon as possible, i.e., already during the design phase. Theoretical methods like *analytical* and *simulation models* facilitate the evaluation of complex systems during the early design phase.

In this contribution, the simulation environment SimPar (HEIN 1994) is described which combines *object-oriented software design* and *process-oriented simulation* allowing a detailed *performability evaluation of massively parallel systems.* The simulation considers specific fault tolerance mechanisms and actual workloads as well as the inter-component dependencies thus capturing the essential characteristics of the actual system.

We address the issue of dependability analysis via *simulation-based fault injection* of massively parallel systems. Faults are injected in a simulation model of the target architecture which also comprises software implemented fault tolerance algorithms and application programs. In this way, the dependability of the multiprocessor system can be analysed under realistic circumstances and the evaluation can be carried out already during the system development. Discovered faults and weak points of the system design can be corrected to avoid their manifestation in the hardware system.

The paper is organized as follows. In Section 2 popular methods and existing tools for performance and dependability evaluation of computing systems resp. related works are briefly described. We present a simulation environment and a concrete simulation model developed for the performability analysis of multiprocessor systems in Section 3. The simulation experiments of Section 4 prove the usefulness and efficiency of the approach; in Section 5 the main issues of the paper are summarized and a brief outlook for future work is given.

2. Modelling Techniques and Related Works

A modelling environment for performability analysis has to provide support for the modelling of the topology and functionality of the target system as well as for the consideration of workload and for fault injection. Performance analysis would be superficial and only of theoretical interest, if realistic workload is not taken into account. For example, the peak performance of a multiprocessor system is often calculated as the product of the peak performance of a single processing node and the number of processing nodes; this artificial scenario does not consider the communication overhead of distributed systems and is not realistic at all. Fault injection is mandatory for dependability analysis and can be conducted dependent on the modelling technique in various ways. To summarize briefly, a modelling environment which fulfills our needs and requirements has to provide the following characteristics:

Our main interest is the evaluation of massively parallel systems including easy scalability. The resulting models are large-scale and ought to be scalable. An object-oriented model design facilitates the specification, implementation, and feasibility of such large and flexible models. An important feature is the simplicity to define the logical and physical interrelationships among the numerous single components.
In order to depict realistic workload, it is desirable to take into consideration real application programs or their time and resource requirements such as computing time, communication requirements, input/output behaviour, etc. within the models.
Modelled system components such as processors or routing switches can be set in faulty states, i.e., their predefined fault-free behaviour is disturbed. This feature of fault injection is considered and interpreted differently from different point of views corresponding to the used modelling technique.

In this section, two of the most important and popular modelling techniques and some modelling tools are discussed which are widely used in the field of performance and dependability analysis of multiprocessor systems.

2.1 Analytical Methods

Analytical methods like Markov processes, queuing systems and various types of Petri nets are popular methods to construct models of the target system, which can be solved by analytical and numerical procedures. These types of models depict the target system on a very high level of abstraction in the very early design phase; sophisticated and well-known algorithms enable the computer architect to specify and analyse a rough model of the target system in a short time. Recently, *Markov reward models* and a specific class of timed Petri nets, the *Generalized Stochastic Petri Nets* (*GSPN*), are frequently used methods and provide useful tools in performability analysis.

Numerous analytical models of massively parallel systems are based on variants of standard *Petri nets*, which are effective means for the description and analysis of concurrency and synchronization in parallel systems. Standard Petri nets are well suited for the description of the logical structure of systems, but they do not include any time concept, i.e., they cannot be used to carry out a quantitative analysis of the system behaviour. The consideration of time in Petri nets allows the modelling of the dynamic behaviour of systems, taking into account both the state evolution and the duration of each action performed by the system. Different approaches of *timed Petri nets* are discussed by MARSAN (1990).

Widely used versions of timed Petri nets are called *Stochastic Petri Nets* (*SPN*) (MARSAN et al., 1984 and MOLLOY 1981). They are obtained by associating with each transition in a Petri net an exponentially distributed random variable which expresses the delay from the enabling to the firing of the transition. MOLLOY (1981) showed that (SPN) are isomorphic to *continuous-time Markov chains* (*CTMC*) which can be handled via well-known analytical and numerical methods.

The so-called *GSPN*s (*Generalized Stochastic Petri Nets*) are introduced by MARSAN et al. (1984) and (1987). They are an extension of the *SPN*s in which timed and immediate transitions both are allowed. Immediate transitions fire in zero time once they are enabled. Timed transitions fire after a random, exponentially distributed enabling time. It has been proven that *GSPN*s are isomorphic to *CTMC*s, i.e., every *GSPN* model can automatically be converted to a *CTMC*.

A formal definition and comprehensive description of *SPN*s and *GSPN*s including numerous examples can be found at (MARSAN et al., 1986). In recent years, numerous tools have been developed and presented supporting the analysis of *SPN* and *GSPN* models such as PENPET (LEPOLD, 1992), SHARPE (TRIVEDI et al., 1987), *SPNP* (TRIVEDI et al., 1990) and others. Because of the isomorphism of *GSPN*s and *CTMC*s, the models can be analysed via procedures which are implemented in these tools. The user has to construct and parametrize the *GSPN* model; the tools generate the underlying *CTMC* and analyse it via analytical and numerical methods.

The so-called *stochastic activity networks* (*SAN*) implemented in the tool UltraSAN are an extension of *GSPN*s (SANDERS et al., 1991 and 1993). Impulse-based rewards, which are associated with the completion of an activity, and rate-based rewards are allowed; the latter are associated with markings. A specification method for combining various submodels in a hierarchical manner is provided.

Models based on *GSPN*s and *CTMC*s can represent massively parallel systems only on a very abstract level because of their fast growing complexity and the huge state-space. A support of hierarchical model design which has only been found in SHARPE facilitates the realization of large-scale models using this modelling technique. Workload is considered implicitly, i.e., the model designer defines working, non-working, and other states of the system as well as transitions between the states. In *GSPN*s and *CTMC*s, fault injection is carried out by defining transitions from *fault-free system states* to *faulty system states*, and vice versa, and putting a probability or distribution function on them. Rewards can be added to the model states in order to measure performance and – in case of faults – performance degradation.

2.2 Simulation Models

Simulation models of the target architecture are powerful means delivering accurate results and avoiding some constraints of analytical modelling such as state space explosion and simplifying assumptions. In comparison to the *GSPN* or *CTMC* based analytical tools, simulation-based tools offer greater flexibility. Different levels of detail can easily be simulated, from the gate to the system level, and complex relationships and dependencies among the system components can be modelled. Furthermore, more complicated time behaviours can be considered, for instance deterministic time as well as Weibull or normal distributed time.

Simulation tools provide different types of model construction, for instance the abstract *Petri net method* and the *object-oriented method*, where each component of the basic system is represented by an object in the simulation model. Simulation tools provide the user with a simulation engine which handles the administration of the event list and the scheduling of the pseudoparallel processes running in the simulation environment. Furthermore, the tools contain algorithms for the generation of random numbers and the recording of information which arise during the runs of the simulation experiments. The user only has to define the model, the experiments, and the characteristics to be measured.

Several tools developed in the last decade facilitate the detailed analysis of parallel system via *simulation-based methods*, such as *C++SIM*, *CSIM*, *DEPEND*, *SIMPACK++*, which are briefly described in this section. Typically, a *process-oriented approach* is chosen which is more convenient and feasible for the construction of large-scale models and for the modelling of complex intercomponent dependencies than the pure event-driven approach (Schwetman 1986). The system behaviour is described by a collection of asynchronous *processes* (or *co-routines*) that interact with one another. Processes represent the functionalities of the system components, e.g. of the processors or routing switches. Only one process executes in any instance of real runtime, but many processes may execute at any instance of simulation time. The processes which are currently inactive are placed on a simulation queue, which is arranged in increasing order of simulation time. The execution of these processes is coordinated by a simulation scheduler.

Besides, the process-oriented simulation environment allows actual user-defined distributed programs to be run within the simulation environment to analyse the complete system considering the multiprocessor hardware, operating system, and workload. The workload consists of user-defined algorithms modelling the structure and the dynamic behaviour of application programs which are common from the point of view of computation and communication.

2.2.1 C++ SIM

C++ SIM (LITTLE et al., 1991) is a simulation package written in *C++* which provides discrete process-based simulation similar to SIMULA's simulation class and libraries. The simulation environment provides active objects, which are instances of *C++* classes, as the units of simulation. For its implementation, use has been made of Sun Microsystems lightweight process (thread) package which can be replaced by other lightweight process packages.

The processes are the basic components of this tool and the user has to use the type inheritance facilities of *C++* to generate his models, i. e., the user can define his own classes of processes representing the desired functionalities. The paradigm of object orientation facilitates the construction and maintenance of hierarchical and large-scale models; nevertheless, the definition of the complex relationships between the numerous components is not sufficiently supported and a suitable modelling environment has to provide such easy-to-handle means. Software written in *C* and *C++* can be executed within the simulation environment representing realistic workload, whereas fault injection is not possible

2.2.2 CSIM

CSIM (SCHWETMAN, 1986) is written in *C* and runs on various computers and operating systems. It is a process-oriented simulation package for use with *C* or *C++* programs. It is developed for the performance evaluation of computer and communication systems. *CSIM* is implemented as a library of routines to create simulation programs. It can be used as an execution and test environment for parallel programs. A simulation model is specified by writing a *C* or *C++* program and by including the library and simulation environment offered by *CSIM*. The statistics gathering is partially automated and is easy to extend. Facilities representing basic servers and storages can be declared and allocated using several different service disciplines (FCFS, Round-Robin, etc.). *CSIM* offers more powerful basic objects than *C++SIM*, but basically, the same points concerning the usefulness and drawbacks of *C++SIM* already mentioned in Section 2.2.1 are valid for *CSIM*.

2.2.3 DEPEND

DEPEND has been developed by Goswami and Iyer (1992 and 1993). Like *CSIM* it is a process-oriented simulation environment, but it supports dependability and performance analysis by allowing the simulated injection of faults in the simulated components. *DEPEND* is written in *C++*; it was designed to handle functional fault models which simulate the *system level manifestation* of low-level faults, such as stuck-at faults.

For model construction the user has to write a *C++* or *C* control program using the pseudoparallel simulation environment and the object library provided by *DEPEND* (*Fig. 2.1*). After compiling and linking the experiment can be carried out. Faults are injected into the components as well as faulty components can be repaired. Besides, data are automatically gathered in the fault statistics.

In the following part, we only speak about objects, but more exactly and following the terminology of object-oriented programming, *DEPEND* provides classes and the user has to instantiate the classes in order to create objects. There are basic objects for the synchronization of processes (*events*), for the exchange of messages between processes (*mailboxes*), and for the statistics gathering. Complex objects simulate the behaviour of a self-checking processor, of a TMR-system, or of a fault tolerant link connection. All objects are designed with four criteria (GOSWAMI AND IYER, 1992):

The objects make very few assumptions and should be of general-purpose.

The objects should easily interface with other objects.

The objects should be easy to customize and easy to use in creating new objects.

The object should provide default functionality to reduce the work done by a user. It should be possible to override these defaults.

Another fundamental *DEPEND* object, the fault injector, is used to inject faults in any components to disturb their predefined behaviour. To activate the injector, the user specifies the time to fault distribution for each component and the fault subroutine which specifies the fault model. For instance, as soon as a fault is injected into a link, the messages are corrupted or lost by a process modifying or destroying the messages. Various distributions of fault injection time, such as exponential, Weibull, or load dependent can be used. A fault report including mean time between failures, mean time to repair, etc. is automatically created and updated during the simulation runs.

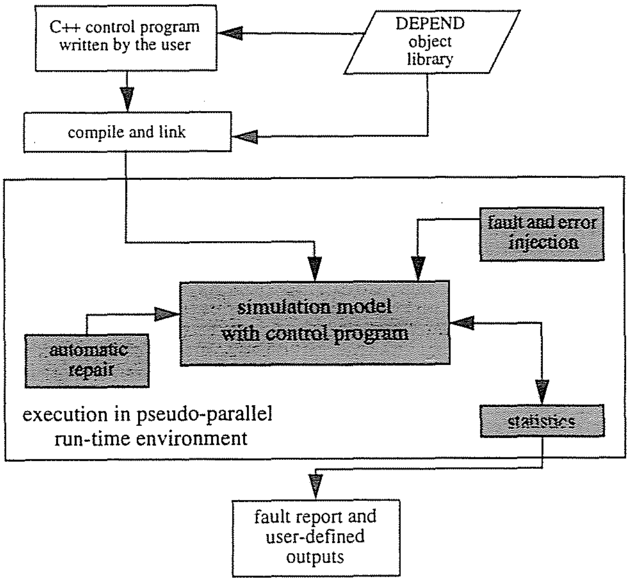
C++ control program written by the user
DEPEND object library
compile and link
fault and error injection
simulation model with control program
automatic repair
execution in pseudo-parallel run-time environment
statistics
fault report and user-defined outputs

Fig. 2.1. Steps in developing and simulating a model with *DEPEND*

Software written in *C* or *C++* for the target system can be executed within the simulation environment to examine their runtime behaviour or to test implemented fault tolerance algorithms. *DEPEND* has successfully been used for the analysis of a TMR-based system and for the simulation of software behaviour under hardware faults (GOSWAMI, 1993).

DEPEND provides powerful and flexible objects for dependability analysis of small models like TMR systems, but it does not sufficiently support the development of large-scale models of massively parallel systems. There is a lack of methods facilitating the definition of the complex relationships and intercomponent dependencies which are typical for multiprocessor systems.

2.2.4 SIMPACK++

SIMPACK (Fishwick) is a collection of *C* and *C++* libraries and executable programs for computer simulation. Several different simulation algorithms are supported including discrete event simulation, continuous simulation and combined simulation. The following modelling types are available:

> *Declarative models*: models with explicit state transitions as finite state machines and Markov models.
> *Functional models*: queuing networks, block models, pulse processes and stochastic Petri nets.
> *Constraint models*: defined by differential and difference equations.
> *Multimodels*: hybrid conglomerates of other models connected in a graph or network to solve combined simulation problems at multiple abstraction levels.

SIMPACK has been used in simulation classes at the University of Florida. It was originally coded in *C* and many parts have been ported to *C++*. For abstract models such as queuing systems and Markov models *SIMPACK* is an interesting and useful tool, but it is not sufficiently suited for a detailed simulation of fault tolerant massively parallel systems considering the possibility of fault injection.

3. *SimPar* - an Environment for the Simulation-Based Performability Analysis of Multiprocessor Systems

The tool *DEPEND* (Goswami and Iyer 1992) has been taken as the basic simulation engine and has been extended to facilitate the modelling of multiprocessor systems. *DEPEND* provides a pseudo-parallel runtime environment which schedules the light-weight processes and a library of elementary objects such as mailboxes, links and fault tolerant servers (described in Section 2.2.3). In addition, dependability experiments can be performed by injecting simulated faults into the components. *DEPEND* does not sufficiently provide means for the design and implementation of large-scale models resulting from massively parallel systems. The simulation environment *SimPar* is developed to enhance the modelling power of *DEPEND* and its main issue is the evaluation of massively parallel systems which provide elementary fault tolerance mechanisms such as spare processors and redundant communication paths.

SimPar is an easy-to-handle extension of *DEPEND*; it provides procedures for the development of models of massively parallel systems and, thus, facilitates the performability analysis of these systems. Various classes representing complex components like interconnection networks and classes for specific components such as routing switches and processors have been implemented. The system architecture is modelled as well as its functional behaviour. Methods are provided which enable the user to depict the hardware topology of the target multiprocessor system, to define his simulation experiments, and to start application programs running within the simulation environment.

For instance, a distributed numerical algorithm is running concurrently to fault-diagnosis programs and a Weibull distribution defines the failure time of the routing switches. After the occurrence and detection of a routing switch failure the simulated multiprocessor system has to be reconfigured and restarted; furthermore, another routing mechanism must be considered to bypass the faulty routing switch. The reconfiguration policy causes a loss of performance due to the routing switch failure which can be measured via the simulation model.

Like *DEPEND*, *SimPar* combines process-oriented simulation and object-oriented model design in order to facilitate an efficient model development and a feasible handling of large-scale models.

3.1 Object-Oriented Model Development

Object-oriented design is the construction of a software system as a structured collection of abstract data type implementations. Data abstraction and inheritance are central features of the object-oriented software design which are ideal for the design, reusability, and scalability of large-scale models of massively parallel systems. Inheritance allows a new class to be derived from an existing class and to inherit its features in order to become an extension or a specialization. In several phases of inheritance and derivation very powerful and specialized classes can be provided without reimplementing every class from scratch. Additionally, the class hierarchy facilitates the reusability and maintainability of complex simulation models.

In accordance with the object-oriented approach classes are implemented in the object-oriented language *C++* representing the processors, routing switches, and links which encapsulate the component-specific functionalities.

When the model of the target multiprocessor system is initialized, objects of these classes are created and dependencies between the objects are defined in a manner that the essential characteristics of the real system are simulated. The physical connections between the processors, routing switches, and links are simulated via logical dependencies between corresponding objects. The object-oriented design facilitates the hierarchical model construction, the encapsulation of the component-specific functionality, and the reusability of the models. For instance, the model can easily be modified by replacing the class of the processor by another class, which simulates the real processor in more details or represents a completely different type of the processor. The classes representing the links and routing switches of the target system are combined in macro classes which model the interconnection network of the system and can be replaced by classes representing other networks with minimal effort.

The user has only to define the topology and the size of his simulation model, for instance, the number of processors (an example for the scalability of a simulation model is given in Section 3.3). An appropriate number of objects of the processors, routing switches and links is initialized and the interrelationships between the objects are defined without any additional lines of user code or an additional effort during the preparation of a simulation run. Thus, easy scalability of complex and sophisticated simulation models is ensured.

3.2 Target System

In this section the topology of a concrete multiprocessor system is presented. The target hardware is a Parsytec GC multiprocessor system which has been the hardware platform of the *FTMPS*[1] project (Parsytec, 1991). This massively parallel machine is designed in order to run scientific and technical application programs requiring huge computing power. This is achieved by the large number of processing elements and their connection via communication paths providing high bandwidth. In addition to this, the Parsytec GC system takes into account the increasing probability of the failure of system components, when their number becomes very large; redundant processors replace faulty processors and the communication paths are redundant to reduce communication delays as well as to tolerate the failure of links and routing switches.

The architecture consists of a data network on which the user application runs and a control network which performs supervising and monitoring functions. The topology of the data network is a three-dimensional grid in which each node corresponds to a *cluster* containing 16 processors plus 1 spare processor; these 17 processors are fully and redundantly connected via 4 crossbar-like routing switches (*Fig. 3.1*).

The control network also forms a three-dimensional grid; each control unit consists of a control processor and a control routing switch and supervises 4 clusters of the data network (*Fig. 3.2*). An entity of 4 clusters and a control unit forms a *cube*. Within the clusters the routing switches

[1]ESPRIT project 6731: FTMPS (a practical approach to Fault-Tolerant Massively Parallel Systems)

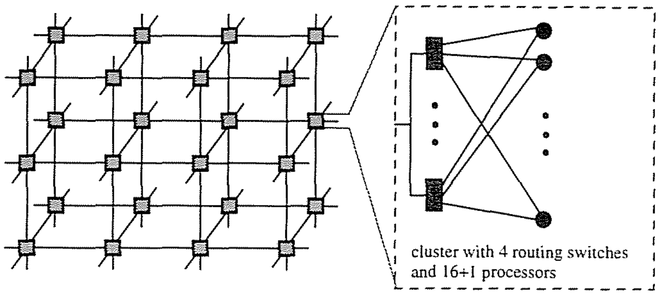

Fig. 3.1. Grid topology of the data network

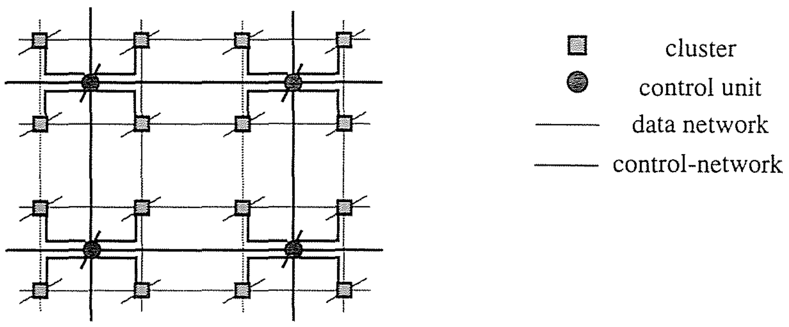

Fig. 3.2. Grid topology of the control network

and processors are lined up in 4 daisy chains via 2 additional control links, whereas the spare processor is directly connected to the control unit (*Fig. 3.3*). The control network is the hardware basis of the fault tolerance software developed for the Parsytec GC. The processors and routing switches of the data network send error messages to the control units when their on-chip control hardware detects an error. The control processors can stop the application programs running on the processors of the data network by sending halt messages via the control network. Furthermore, the control processors supervise their neighboring processors in the three-dimensional grid of the control network.

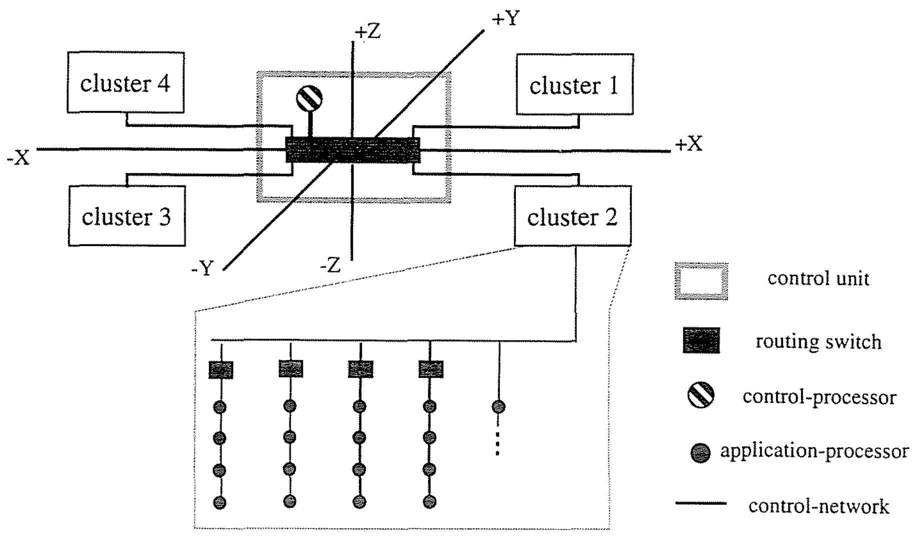

Fig. 3.3. Control network within a cube

3.3 Simulation Model of the Target System

SimPar has been used to construct the simulation model of the Parsytec GC in accordance with the object-oriented approach in order to facilitate the scalability and reusability of the simulation model. In the following, $SimPar_{GC}$ stands for the simulation model of the Parsytec GC (HEIN, 1994). Each object in the model represents a group of physical components performing some system function. Classes of objects are implemented which simulate the functionalities of the processors, links and routing switches in the target system (*Fig. 3.4*). These objects are hierarchically connected together to form a complete system. In this section a simulation model of the concrete architecture is described, but the concepts are generally usable and a simulation model of any multiprocessor topology can be developed with SimPar following the same approach.

The *processor object* is supposed to perform processes, to schedule them, and it has to receive and send messages. As soon as a fault is injected or a latent fault is activated, the processor object ejects any jobs in progress and stops handling messages. The on-chip control hardware of the processor is modelled to simulate fault injection and fault tolerance experiments in an accurate manner, i. e., if an error is detected by the on-chip control hardware a corresponding error message is sent to the control processor of its cube.

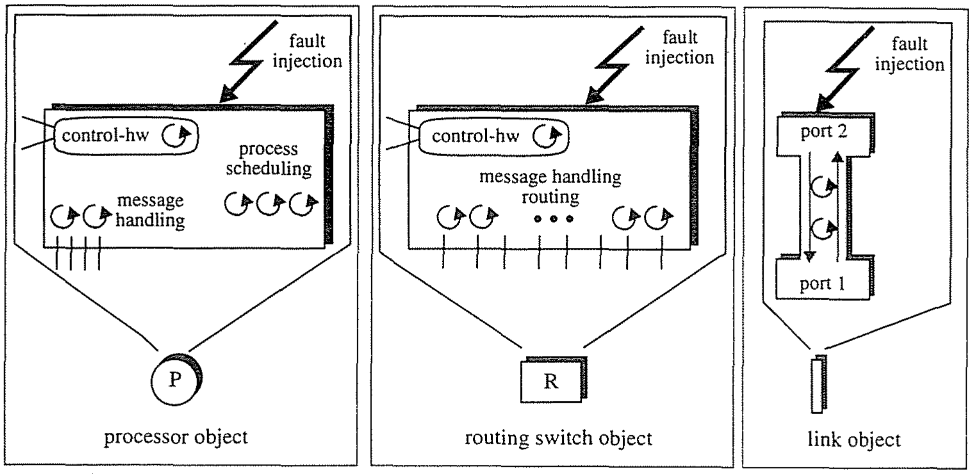

Fig. 3.4. Basic components

The *routing switch object* receives and forwards messages; the on-chip control hardware fulfills the same task as the on-chip control hardware of the processor object. A faulty routing switch communicates no longer with its links, and all messages buffered in the routing switch are lost.

The *link object* receives messages at a port and forwards it to the other port. When a link object is injected, it stops forwarding messages from the incoming to the outgoing port or it corrupts messages. The relationships and dependencies among the basic components of the simulation environment have to be defined in a way, that the topology and functionality of the target system are represented. More complex objects are created which model parts of the multiprocessor system, such as clusters and cubes of the Parsytec GC (*Fig. 3.5*). A *cluster object* contains several processor objects and routing switch objects which are connected by an appropriate number of link objects. The cube object comprises a processor object, a routing switch object and several link objects modelling the control net of a cube. The cluster and cube objects model the topology and functionality of a cluster and a cube of the Parsytec GC, respectively, which are described in Section 3.2.

After the user has defined the number of clusters in the three spatial dimensions of the data network, the required number of clusters and control unit objects are automatically initialized, and the model of the target architecture is constructed in two steps. Firstly, the interdependencies between the cluster and cube objects are defined to model the topology of a single cube of the target system; secondly, additional link objects are positioned

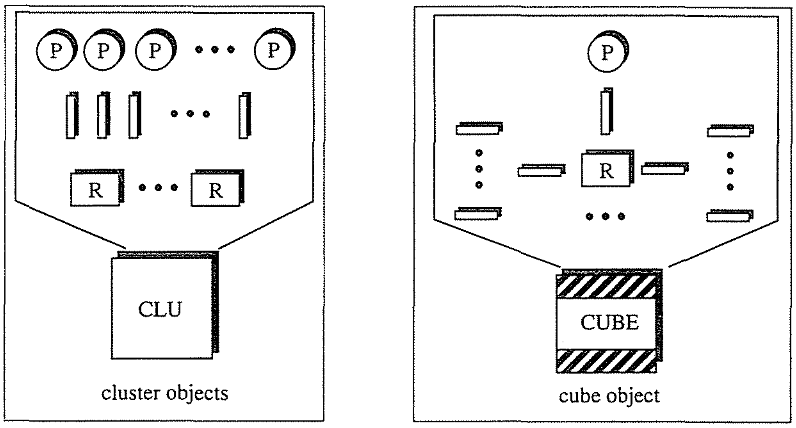

Fig. 3.5. Objects for clusters and cubes

to enable communication via the data network between the cluster objects as well as communication via the control network between the control unit objects. The final object is a model of the overall target multiprocessor system (*Fig. 3.6, object* MP). The hierarchical model structure supported by the object-oriented design facilitates the scalability and reusability of the simulation model. For instance, the processor object (*Fig. 3.4*) can easily be replaced by another object, which models the real processor in more detail.

The multiprocessor object MP is totally scalable by varying the number of clusters in the three dimensional grid of the data network. It is the interface to the user-written control program, i.e., the user has to call methods of the MP object to load user-defined processes on the processors of the data and control network. These processes can be real distributed application programs as well as operating system routines. For instance, fault tolerance programs to perform fault diagnosis and reconfiguration of the simulated multiprocessor system can run on the simulated multiprocessor system as well as distributed number crunching algorithms modelling realistic workload. Messages are sent from the sending processor object through the intermediate link and routing switch objects of the data or control network to the receiving processor object. Like in the target system, successfully received messages are acknowledged. Additionally, the MP object provides methods to define the fault injection, to get information about the current system state, and to output fault reports. The system analyser prepares his experiments by simply calling methods of the MP object.

To summarize, the following steps are required to prepare and run a simulation experiment with *SimPar*:

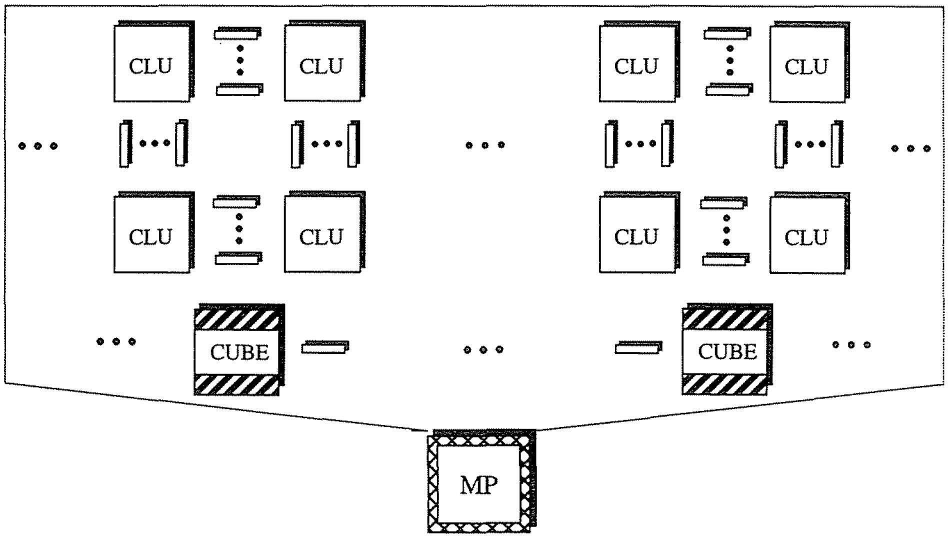

Fig. 3.6. Object MP: model of the complete multiprocessor system

1. Definition of the multiprocessor architecture. If $SimPar_{GC}$ is used, the basic topology is predefined and it is sufficient to input the size of the system. Furthermore, the global random number generator used for fault injection has to be initialized with a seed.
2. Definition of the dependability experiment, for instance fault injection into various components based on distribution functions. Fault tolerance algorithms can be loaded and executed on the simulated multiprocessor system.
3. Definition of the workload. User-defined application processes are mapped and loaded on the model in order to simulate a realistic workload scenario.
4. Start of the experiment.
5. End of the experiment; output and analysis of the simulation results.

4. Examples

In this section we present two problems which have to be examined within the *FTMPS* project and which are analysed with the simulation model $SimPar_{GC}$ described above. The first example described in Section 4.1 deals with fault diagnosis which is based on heartbeat messages; the fault diagnosis algorithm specified for the Parsytec GC multiprocessor system uses heartbeat messages to detect and localize faulty processors. Spare processors are available in the target system which replace faulty ones; different reconfiguration policies are possible and their efficiency depends strongly on the actual distributed application programs as we show in Section 4.2.

4.1 Fault Diagnosis based on Heartbeat Messages

Fault tolerance methods are added to $SimPar_{GC}$ in the form of an error detection and localization mechanism based on heartbeat messages and on the fail-silent assumption of the processing nodes.

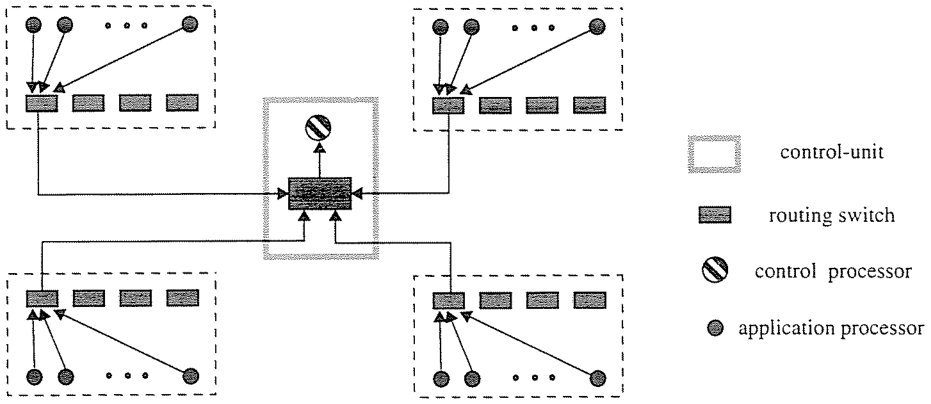

Fig. 4.1. Heartbeat messages

One of the tasks of the control processors is the detection and localization of faulty application processors. For this purpose we implement heartbeat messages sent by the application processors to the associated control processor. A specific connection is used between one routing switch per cluster of the data network and the routing switch of the control unit, which is the only hardware connection between data and control network. Each application processor performs an additional high-priority process that periodically sends a heartbeat message. Furthermore, on each control processor a supervisor process periodically checks the time stamps of the associated application processors of the 4 clusters. The paths of the heartbeat messages within a cube are shown in *Fig. 4.1.*

The messages are sent periodically in predefined time intervals and the control processors perform timeout checks. Every control processor updates the last time stamp of an application processor as soon as it receives a heartbeat message of this processor. The frequency of the heartbeat messages has a large impact on the efficiency of this error detection mechanism. On the one hand, errors of subcomponents as processors in a multiprocessor environment have to be detected as fast as possible to avoid error propagation; in order to achieve a short latency time, the heartbeat messages have to be sent and checked very frequently. On the other hand, processes sending heartbeat messages can influence the runtime of the concurrent application programs.

When implementing heartbeat messages in distributed systems the three parameters T_{AL}, T_{SV} and T_{DM} have to be defined. T_{AL} is the time interval after that the application processor has to send a heartbeat message. A high-priority process is started on every application processor sending the messages in an infinite loop. Since we use a priority-driven process-scheduling strategy and since there are no other high-priority processes running on the application processors, the process sending heartbeat messages accesses the processor as soon as it calls for it.

After every T_{SV} time units a high-priority process running on every control processor checks the last time stamps of its associated application processors and assumes that a processor is faulty if its last time stamp is older than T_{DM} time units.

The maximum allowed delay time T_{DM} between two successive time stamps of an application processor has to consider the time interval T_{AL}, but also the possible time delay of the message caused by high message traffic and contention in the network. If T_{DM} is too small, the control processor would falsely assume that a non-faulty application processor is faulty, whereas its heartbeat message is blocked in the network and does not arrive in time.

It is an important and complex issue to tune this mechanism to achieve small error detection times without false alarms and to influence the flow and runtime of application programs during the fault-free case as little as possible. We implement and execute the described mechanism on $SimPar_{GC}$, the simulation model of the Parsytec GC, and vary the parameters T_{AL}, T_{SV}, and T_{DM}, to find out their influence on the latency times. The heartbeat messages use both the data network and the control network, because they are sent via the connection between them.

In order to simulate a realistic situation, we start an application program concurrently to the processes of the heartbeat mechanism. The application program models the workload and performs a grid-like communication pattern, i.e., application process (i, j) exchanges data with application processes $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. This communication pattern is typical for a large class of numerical algorithms such as the red-black relaxation or multigrid methods for the analysis of partial differential equations, in which the data are partitioned among the processing nodes. The simulation model is initialized with 4 clusters and a control unit forming a cube.

The results of the simulation experiments do not show any measurable difference of latency times between the experiments with and without the concurrently running application program performing frequent interprocess communication. This is due to the fact that the processes sending heartbeat messages are the only high-priority processes running on the application processors; therefore, it is guaranteed that it is executed by the processor as soon as it calls for the processor. Another reason is the high redundancy in the data network of the clusters. Every processor has 4 links which can be used for interprocessor communication and the application processes have at most 4 logical neighbouring processes, i.e., the links are not loaded intensively. The process sending heartbeat messages can only use the link connected to the routing switch of the data network which has a link to the control unit. The links between the control processor and the routing switches of the data network are not used by application processes; therefore, running application processes including their communication have almost no impact on the contention situation in this part of the network.

In *Fig. 4.2* the curves represent the mean latency times of the heartbeat-based error detection mechanism with concurrent application processes. Heartbeat messages are sent in intervals of $T_{AL} = 1.0$ sec; the time intervals T_{SV} and T_{DM} are shown in the figure. It can be seen that the latency times increase almost linearly with T_{SV}. An interesting fact is that the latency time is not reduced if T_{DM} is chosen smaller than 1.1 sec (not shown in the figure). For comparison purposes curve ctrl shows the mean latency time, if the error is detected by the on-chip control hardware of the injected application processor and an error message is automatically sent to the control processor; the mean latency time in this case is 4.404e-6 sec. The values are based on the performance of the target system and are measured with a relative error less than 5.0 percent.

4.2 Impact of Reconfiguration Policies on the Run Time of Distributed Application Programs

In this example we inject permanent faults in the processors of the data network. After error detection and fault diagnosis the system is reconfigured and the application processes of faulty processors are restarted on the spare processors. If the initial mapping was optimal, a loss of performance has to be accepted which will be measured. The loss of performance causes

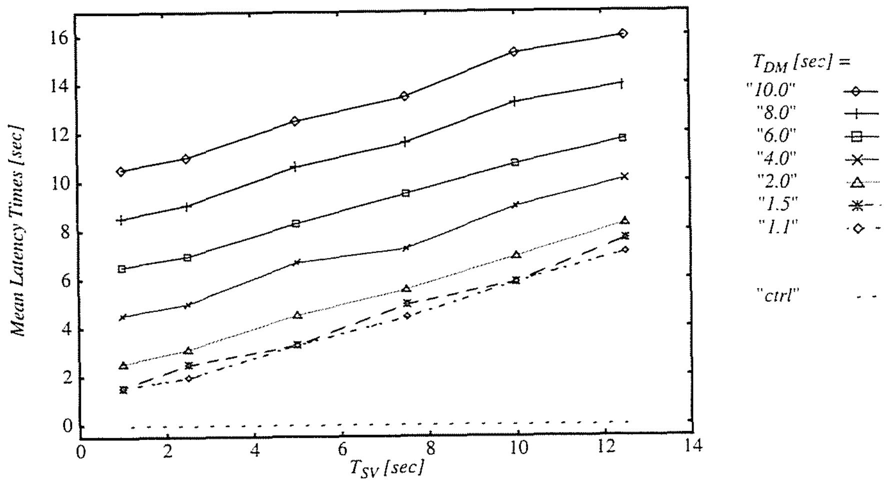

Fig. 4.2. Mean latency times

longer run times of the distributed application programs and is due to the fact that the communication between processes takes more time. The reconfiguration policy can assign processes which have run on processors of the same cluster to processors of different clusters possibly resulting in longer communication times.

Only 16 of the 17 processors in a cluster are used for application programs. The 17th processor of each cluster is a spare processor, which can replace a faulty one. Simulating a system with 4 (8) clusters, we map the logically neighboring processes on the processors following an optimal mapping scheme and run the application on the 64 (128) processing nodes. We measure the mean communication times without processor failures. In the optimal case, at most one processor per cluster fails and is replaced by the spare processor of the same cluster. A primitive reconfiguration algorithm which does not consider this policy increases the communication overhead. Additionally, if more than one processor per cluster fails, spare processors of other clusters, which are selected in accordance with a predefined rule, have to take over the processes.

A numerical algorithm is started on the simulated multiprocessor, of which the communication pattern is a simple line, i.e., process i exchanges data with process $i+1$ and process $i-1$. For this communication pattern it is quite easy to construct an optimal mapping scheme allowing shortest communication time and therefore shortest run time of the distributed algorithm on the underlying architecture.

The efficiency of the following reconfiguration policies is measured in terms of run time of the distributed application.

$reconf_a$: If the spare processor of cluster 1 is still intact and free, it replaces the faulty processor; otherwise, if the spare processor of cluster 2 is still intact and free, it is selected; otherwise the spare processor of cluster 3 is checked, etc.

$reconf_b$: A processor of cluster fcl fails. The spare processor of cluster fcl takes over the processes of the faulty processor, if it is fault-free and unused. Otherwise, the processors of cluster $fcl + 1$, $fcl + 2$, $fcl + 3$ and so on until cluster $fcl - 1$ are checked.

$reconf_c$: A cluster is randomly chosen until a cluster with an unused and intact spare processor is found which replaces the faulty processor.

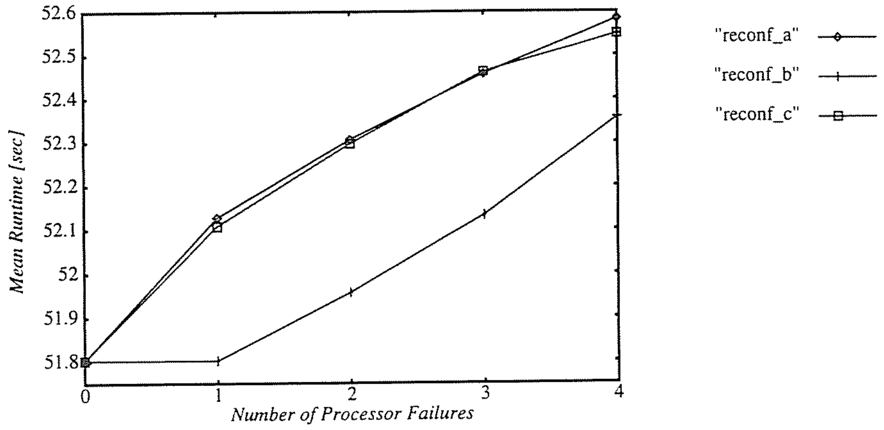

Fig. 4.3. Run time of a system with 4 clusters (1 cube)

A single processor failure has no impact on the run time of the algorithm if reconfiguration policy $reconf_b$ is selected (*Fig. 4.3* and *Fig. 4.4*). This is due to the facts that the spare processor belongs to the same cluster as the faulty processor and within a cluster the interprocessor connections are symmetric and redundant (*Fig. 3.1*). In a system of 4 clusters as well as in a system of 8 clusters the mean run time of strategy $reconf_b$ is clearly shorter than the mean run times of $reconf_a$ and $reconf_c$ which have only minor differences for any number of processor failures.

The statistical results in Section 4.2 are measured with a relative error of less than 1.0 percent.

This experiment will be extended allowing a larger number of processor failures than spare processors available. Processes of additionally failed processors have to be scheduled to processors which already run application processes; it is assumed that the run time of the overall application will heavily increase since not only the communication but also the computation will take additional time. Furthermore, other communication patterns will be considered such as a mesh-like or cluster-oriented communication pattern considering communication which is more intensive within the clusters than between processors of different clusters.

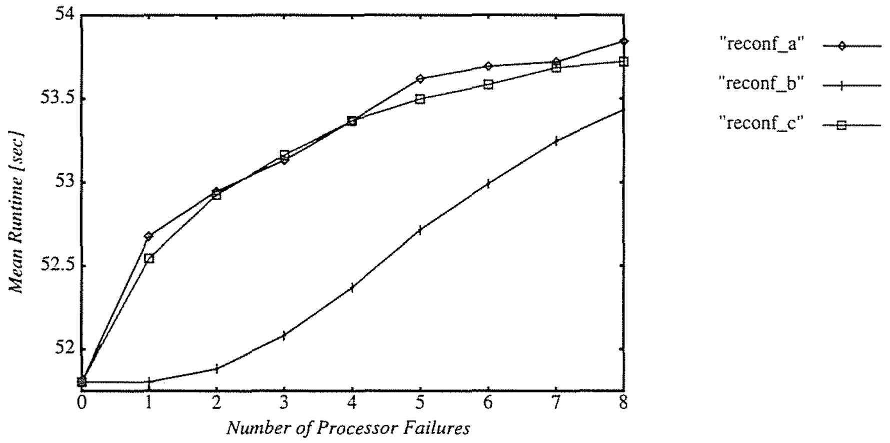

Fig. 4.4. Run time of a system with 8 clusters (2 cubes)

5. Summary and Future Works

In this paper, we presented a method for an efficient performability analysis of fault tolerant massively parallel systems. The modelling environment *SimPar* has been implemented based on process-oriented simulation and object-oriented model development which are well suited for the design of large-scale models and for the modelling of complex inter-component dependencies. Real distributed programs can run on the simulated multiprocessor systems representing realistic workloads and valid scenarios for performance and dependability evaluation. In future, we will use *SimPar* to investigate the impact of sophisticated fault tolerance algorithms on the run time behaviour of various distributed application programs and benchmarks on different multiprocessor topologies.

Currently new features are being added to *SimPar* to offer an easy-to-handle and powerful tool for the modelling of user-defined multiprocessor systems. Simulation objects of I/O components such as hard disks will be provided to take the I/O behaviour of the simulated multiprocessor system into consideration. This feature will allow accurate investigations of fault tolerance algorithms including checkpointing, diagnosis, reconfiguration, and recovery.

Furthermore, within the *FTMPS* project a simulation model representing the *Parsytec GC/PP* architecture, Parsytec's multiprocessor system using the PowerPC processor as computing nodes and transputer as communication processors (Parsytec, 1994), is being developed based on *SimPar*. In a future version of *SimPar* various process scheduling policies, such as pre-emptive resume, pre-emptive round robin, etc., will be available. Additionally, an interface to a portable parallel programming environment such as *PVM* (*Parallel Virtual Machine*) (GEIST et al., 1993) or P4 (BUTLER et al., 1994) will be provided in order to run portable parallel programs without changes on the simulated multiprocessor systems.

References

BUTLER, R. M. – LUSK, E.L. (1994): User's Guide to the p4 Parallel Programming System. Argonne National Laboratory, Mathematics and Computer Science Division. Argonne, IL (USA), 1994.

FISHWICK, P. A. (): *SIMPACK*: Getting Started with Simulation Programming in *C* and *C* + +. Department of Computer Information Science, University of Florida.

GEIST, A. ET AL. (1993): PVM 3 User's Guide and Reference Manual. Oak Ridge National Laboratory, Oak Ridge, Tennessee. Tennessee, May 1993.

GOSWAMI, K. K. – IYER, R. K. (1992): DEPEND: A Simulation-Based Environment for System-Level Dependability Analysis. Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1992.

GOSWAMI, K. K. (1993): Design for Dependability: A Simulation-Based Approach. Ph.D. Thesis, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1993.

HEIN, A. (1994): $SimPar_{GC}$ - Ein Simulator zur Leistungs- und Zuverlssigkeits-Analyse des Multiprozessorsystems Parsytec GC, Version 1.0. Internal Report 2/94, IMMD III, University of Erlangen-Nürnberg, 1994.

LEPOLD, R. (1992): PENPET: A New Approach to Performability Modelling Using Stochastic Petri Nets. in B. R. Haverkort, I. G. Niemegeers, and N. M. van Dijk (eds), Proc. of the First Int. Workshop on Performability Modelling of Computer and Communication Systems, 1992.

LITTLE, M. C. – MCCUE, D. L. (1991): Construction and Use of a Simulation Package in *C++*. Department of Computing Science, University of Newcastle upon Tyne, 1991.

MARSAN, M. A., – BALBO, G. – CONTE, G. (1984): A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. Transactions on Computer Systems (acm), Vol. 2, No. 2, May 1984, pp. 93-122.

MARSAN, M. A., – BALBO, G. – CONTE, G. (1986): Performance Models of Multiprocessor Systems. Cambridge, London: the MIT Press, 1986.

MARSAN, M. A., – BALBO, G., – CHIOLA, G. – CONTE, G. (1987): Generalized Stochastic Petri Nets Revisited: Random Switches and Priorities. Proc. of the International Workshop on Petri Nets and Performance Models, Madison, WI, USA, August 1987. IEEE Computer Society Press, 1987.

MARSAN, M. A. (1990): Stochastic Petri Nets: An Elementary Introduction. Advances in Petri Nets 1989, LNCS 424, Springer, 1990.

MOLLOY, M. K. (1981): On the Integration of Delay and Throughput Measures in Distributed Processing Systems. Ph.D. Thesis, UCLA, Los Angeles, CA, 1981.

PARSYTEC (1991): The Parsytec GC Technical Summary, Version 1.0. Parsytec Computer GmbH, Aachen (Germany), 1991.

PARSYTEC (1994): Parsytec GC/PP System Information, Report 1.1.4, Esprit Project 6731, Parsytec Computer GmbH, Aachen (Germany), 1994.

SCHWETMAN, H. (1986): *CSIM*: A C-Based, Process-Oriented Simulation Language. Proc. of the 1986 Winter Simulation Conference (WSC' 86), Washington, D.C., 1986.

SCHWETMAN, H. (1992): *CSIM* User's Guide, Rev. 2. MCC Technical Report, Microelectronics and Computer Technology Corporation, July 1992.

SANDERS, W. H. ET AL. (1991): Performability Modelling with UltraSAN. *IEEE Software*, 8, September 1991.

SANDERS, W. H. – OBAL II, W. D. (1993): Dependability Evaluation using UltraSAN. Proc. of the FTCS-23, Toulouse, France, 1993, IEEE Computer Society Press, 1993.

TRIVEDI, K. S. – SAHNER, R. A. (1987): Reliability Modeling Using SHARPE. *IEEE Transactions on Reliability*, Vol. R-36, No. 2, June 1987.

TRIVEDI, K. S., – CIARDO, G. – MUPPALA, J. (1990): SPNP: Stochastic Petri Net Package. *Proc. of the Third International Workshop on Petri Nets and Performance Models*, Kyoto, 1989. IEEE Computer Society Press, 1990.