Conjoint Simulation - a Technique for the Combined Performance and Dependability Analysis of Large-Scale Computer Systems

Axel Hein[†] and Kumar K. Goswami[‡]

[†]Institute for Computer Science III (IMMD III)
University of Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
e-mail: alhein@informatik.uni-erlangen.de

[‡]Tandem Computers
19333 Vallco Parkway, Loc 3-22
Cupertino, CA 95014, USA
email: kumar@loc3.tandem.com

Abstract

In this paper we propose an approach which seamlessly integrates two modeling techniques to facilitate the simulation and analysis of performance and dependability of large-scale parallel computer systems. The approach, called Conjoint Simulation, combines object-oriented, process-based simulation with Petri net modeling. The approach splits a system into a performance model and a dependability model. The models are developed independently but interact at run time to provide both performance and dependability metrics. This separation facilitates model representation, development, and maintenance.[1]

1 Introduction

The complexity of modern computer architectures requires sophisticated design tools which explicitly support the evaluation and optimization of system attributes such as performance and dependability. As computers become more complex, the probability of system failure increases. This is especially true for *massively parallel systems* which contain hundreds or thousands of computing nodes interconnected via elaborate communication networks. For such systems, it is important to not only study their performance characteristics but also their dependability and how it impacts system performance, i.e. performability.

However, joint performance and dependability evaluation of large systems is difficult because one has to design and incorporate a failure-repair model into an already complex architecture model. If users want to evaluate different architectures and failure-repair mechanisms they must either create separate models for each combination or create one, large, complex model to evaluate all combinations. This can be a daunting task.

This paper proposes an approach called *Conjoint Simulation* to address this issue. *Conjoint Simulation* combines object-oriented, process-based simulation with Petri nets. The object-oriented, process-based simulation paradigm is used to represent the architecture-workload model (*AWM*) of the system being evaluated. A separate Petri net is used to represent the failure-repair model (*FRM*). The two models are tied together by defining key events (e.g. component failure, component reintegration) in the simulation model. Thus the models are specified independently but their run-time behavior is inter-dependent. This approach has several advantages:

1. Two submodels are easier to develop than one large model that incorporates both functions.
2. Each model is developed with the best suited modeling paradigm. The authors feel that detailed, complex architectural models are best represented using object-oriented, process-based simulation whereas most failure-repair mechanisms are most easily represented with Petri nets.
3. Each submodel can be developed, modified and maintained independently of each other. Only the interface defining their interactions must remain the same or modified concurrently.
4. Various combinations of architectures, workload scenarios, and failure-repair mechanisms can be easily evaluated to provide both performance and dependability metrics.

Conjoint Simulation is incorporated in the modeling environment *SimPar* (*Sim*ulation of *Par*allel systems) which supports both object-oriented, process-based simulation and Petri net modeling paradigms. *SimPar* is a tool that has been explicitly designed to facilitate the development, and performance and dependability analysis of massively parallel fault-tolerant systems [5]. *SimPar* uses *Depend* which was developed at the University of Illinois at Urbana-Champaign [2]. *Depend* provides the underlying simulation engine and the basic components such as fault-tolerant servers and busses. *Depend* has been used for the analysis of a triple-modular-redundancy system and for the simulation of software behavior under hardware faults [3], [4]. An overview of *SimPar*, *Depend*, and further related modeling tools is given in [6].

The following chapter provides background on related research. This is followed by two chapters descibing the two submodels. Chapter 5 discusses the properties of *Conjoint Simulation* and how the two submodels interact.

1. This paper was published in Proceedings of the IPDS '96, *International Performance and Dependability Symposium*, Urbana-Champaign (IL), U.S.A., September 1996

An example to illustrate the approach is presented in Chapter 6, and Chapter 7 summarizes the paper.

2 Related work

We refer to the technique of combining object-oriented, process-based simulation models with Petri net models as *Conjoint Simulation* to distinguish it from other *hybrid modeling* approaches. Hybrid approaches typically simplify a detailed model and combine simulation and analytical techniques to reduce the time needed to evaluate a model; these techniques provide a close approximation of the results obtained from a detailed model.

For instance, the modeling tool *HARP* (*Hybrid Automated Reliability Predictor*) combines different modeling techniques and provides an approach called *behavioral decomposition* which is based on the observation that the fault-occurrence/repair behavior consists of relatively rare events, while the fault/error-handling behavior is composed of events occurring in rapid succession, once a fault has happened [13]. The dependability model is decomposed into a fault-occurrence/repair (*FORM*) and a fault/error-handling (*FEHM*) submodel. *FORM* and *FEHM* are separately constructed and their results are automatically combined. The *FORM* submodel describes the structure of the hardware redundancy, the fault arrival process, and manual repair; it is represented as a Markov chain or a fault tree which is internally converted to a Markov chain. The occurrence of permanent, intermittent, and transient faults as well as the on-line recovery procedure are modeled in the *FEHM* using various modeling techniques. Simulation is used for the analysis of the *FEHM* if it is represented as an *ESPN* (extended stochastic Petri net), a specific type of timed Petri nets.

Another hybrid approach for performance analysis was presented in [11]. *Discrete event simulation* and *analytical queuing network models* are combined to achieve good approximation and high speed-up in comparison with a pure simulation model. To analyze a central server queuing model, the arrival and activation of jobs is represented using discrete event simulation, whereas queuing networks, which model the use of processors, are evaluated analytically in order to reduce the overall evaluation time. Both of these approaches strive to approximate the results via model simplification and abstraction.

The primary objective of *Conjoint Simulation*, however, is to facilitate the representation of a system in detail so as not to abstract away system characteristics that are crucial to the performability of the system. It should be stated that our approach to *Conjoint Simulation* does not preclude and in fact can also facilitate hybrid simulation; however, that is not the focus of this paper.

3 *AWM* - the architecture-workload model

The *AWM* comprises the architecture and the workload of the target system. The architecture and the topology are represented using *object-oriented* and *process-based paradigms*. A system-level notation is used to define the key entities and their physical connections. These entities are essentially basic classes implemented in C++ for the modeling of processors, switches, memories, input/output components, and link connections; they provide predefined, but user-replaceable functions and error injection mechanisms to alter their specified behavior. *SimPar* provides high-level commands that automatically create regular topologies such as *grid* and *tree* as well as other modeler-defined topologies based on the basic classes [5]. The construction and maintenance of the architecture model is facilitated by the hierarchical definition of the topology and of the routing tables which is presented in [1]; furthermore, an object-oriented interface for the workload representation encourages the modeling of complex workload scenarios [12].

The dynamic flow of the *AWM* is realized by a set of asynchronously interacting *simulation processes* which are assigned to the basic classes to depict their temporal and functional behavior such as the routing and forwarding of messages. Additionally, the process-based paradigm is a well-suited technique to represent workload common for distributed systems. Each workload process of a user-defined distributed application program written in C or C++ is scheduled like an ordinary simulation process by the simulation engine; these processes are mapped onto the architecture model and executed under control of *SimPar*.

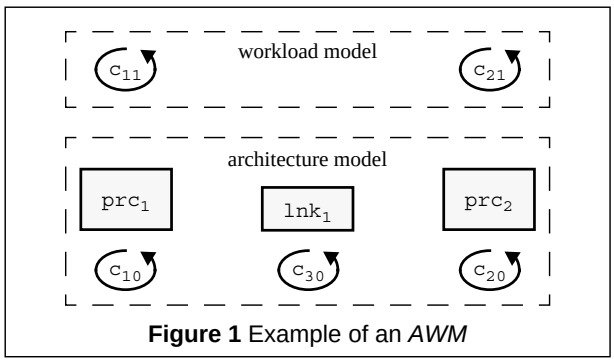

Figure 1 Example of an *AWM*

A simple model of a computer system is shown in Figure 1 illustrating the basic modeling concepts of the *AWM*. The model consists of two processor objects prc_1 and prc_2 and a link object lnk_1. The processor objects simulate a typical processor upon which *simulation processes* execute. The link object simulates a generic link through which data is passed between a source and a destination. A simulation process is assigned to each processor object to perform the scheduling of workload processes (c_{10} and c_{20} in Figure 1). The simulation process c_{30} of the link object models the forwarding of messages sent between prc_1 and prc_2. The workload is represented by the two workload processes c_{11} and c_{21} which are assigned to the processor objects prc_1 and prc_2, respectively; c_{11} and c_{21} may model data processing such as

numerical algorithms as well as the sending and receiving of messages. If the fail-stop failure mode is chosen, injecting an error into processor prc_1 will prevent the process c_{10} from scheduling further workload processes such as c_{11}.

4 *FRM* - the failure-repair model

The *failure-repair model* (*FRM*) depicting the failure modes and the repair-maintenance mechanisms of the target system is represented with a *timed-transition Petri net* (*TTPN*). Petri nets are user-friendly description methods and are well-suited for graphically representing asynchronously interacting sequences of events with complex dependencies. This makes them ideal for modeling failure, repair, and maintenance behavior where how a system fails or whether it is in proper order depends not only on the particular entity in question but also on the activities and states of other entities on which it relies.

According to the definition of *generalized stochastic Petri nets* (*GSPN*), time behavior is assigned to the transitions of the *TTPN* [7], [8]. Since we are using simulation instead of analytical or numerical analysis methods, various distribution functions, constant values, or user-defined random variates are valid to define the time delay between enabling and firing of a transition. Because the underlying stochastic process is not necessarily Markovian, we take into consideration the three memory policies *age memory* (*R-A*), *enabling memory* (*R-E*), and *resampling* (*R-R*) presented in [9] to model the memory dependencies of the transitions. The formal notation of a *TTPN* model is given in Def. 1.

Def. 1 *Timed-Transition Petri Net TTPN*

A timed-transition Petri net *TTPN* = (P, T, A, M_0, H, R, G, E) contains a place-transition net *PN* = (P, T, A, M_0) with a set of n places P = (p_1, p_2, p_3, ..., p_n), a set of m transitions T = (t_1, t_2, t_3, ..., t_m), a set of directed arcs $A \subseteq (P \times T) \cup (T \times P)$, and an initial marking $M_0 = (m_{01}, m_{02}, m_{03}, ..., m_{0n})$ with m_{0i} denoting the initial number of tokens in place i. Additionally, a *TTPN* comprises the following components and characteristics:

- a set of inhibitor arcs $H \subseteq (P \times T)$,
- a set of priorities $R = (r_1, r_2, ..., r_m)$ which are non-negative integer values and are assigned to the transitions; the lowest priority 0 is assigned to timed transitions, and priorities r_i equal to or larger than 1 can be assigned only to immediate transitions,
- a set of weight and distribution functions $G = (g_1, g_2, ..., g_m)$; g_i defines either the weight w_i used for the computation of firing probabilities if t_i is an immediate transition, or g_i defines the distribution function f_i of the transition firing delay if t_i is a timed transition,
- a set of memory policies $E = (e_1, e_2, ..., e_m)$ assigned to the transitions with $e_i \in \{R\text{-}A, R\text{-}E, R\text{-}R, \perp\}$; if t_i is a timed transition, e_i defines its memory policy; otherwise e_i is undefined, i.e., $e_i = \perp$.

In the graphical representation timed transitions are drawn as filled boxes, while immediate (timeless) transitions are drawn as thin bars.

5 *Conjoint Simulation*

The interaction between *AWM* and *FRM* is based on the events of the *TTPN* model, that means on the enabling or firing event of a transition in the *TTPN*. Thus, we extend Def. 1 and call this extended type of Petri nets *interpreted timed-transition Petri nets* (*ITTPN*).

Def. 2 *Interpreted Timed-Transition Petri Net ITTPN*

An interpreted timed-transition Petri net *ITTPN* = (P, T, A, M_0, H, R, G, E, C^A, C^F) is a *TTPN* = (P, T, A, M_0, H, R, G, E) which is extended by the two sets C^A and C^F. $C^A = (C^A_1, C^A_2, ..., C^A_m)$ comprises the sets of operations which are performed in the *AWM* when a transition is enabled. $C^A_i = \langle c^A_{i,1}, c^A_{i,2}, ... \rangle$ is a sequence of operations $c^A_{i,j}$ to perform when transition t_i is enabled. Equivalently, $C^F = (C^F_1, C^F_2, ..., C^F_m)$ consists of the sequences $C^F_i = \langle c^F_{i,1}, c^F_{i,2}, ... \rangle$ of operations $c^F_{i,j}$ which are executed as soon as transition t_i fires.

It should be noted that the operations $c^A_{i,j}$ and $c^F_{i,j}$ performed in the *AWM* can not only start activities to influence and alter the behavior of the *AWM*, but they can also query the status of the *AWM* in order to control and modify the *FRM*. In the following sections we describe possible interactions between *FRM* and *AWM*. For the sake of simplicity, we are using self-explaining names as indices of the *FRM* components instead of numbers; for instance, t_{inj} denotes a transition which triggers an error injection.

5.1 Active role of the *FRM*

The role of the *FRM* in *Conjoint Simulation* is twofold. First, the *FRM* is an active counterpart of the *AWM* exerting an influence on the dynamic and functional behavior of the *AWM*. Since the *FRM* controls and triggers the error injections into components of the *AWM* as well as the fault-tolerance mechanisms performed within the *AWM*, it can be regarded as a control unit which launchs operations in the *AWM*. At the same time, the *FRM* reflects the state of the *AWM* concerning the failure of components and possible ongoing diagnosis, recovery, or repair activities. The markings of the *ITTPN* display the state of failure, the state of recovery, or other corresponding states of the *AWM*. Abstracting from the numerous detailed states of the *AWM* such as the forwarding of a message or the computation phase of a workload process, the states - or markings - of the *ITTPN* represent the superstates of the overall system model.

The example in Figure 2 shows an *ITTPN* whose transitions control and trigger activities of the *AWM*. As soon as the timed transition t_{inj} is enabled the applications of the workload model are started or, if the applications have been interrupted by an error or recovery mechanism, the applica-

tions are rolled back and are restarted from the last valid checkpoint. After a period of time defined by the distribution function f_{inj} an error is injected into one of the processor objects of the *AWM*. After the firing of t_{inj}, the timed transition $t_{recover}$ is enabled and the applications of the workload part of the *AWM* are stopped. The distribution function $f_{recover}$ assigned to $t_{recover}$ defines the time between enabling and firing of $t_{recover}$; when $t_{recover}$ fires the faulty component of the *AWM* is replaced and the *AWM* is reconfigured. Finally, t_{inj} is enabled again.

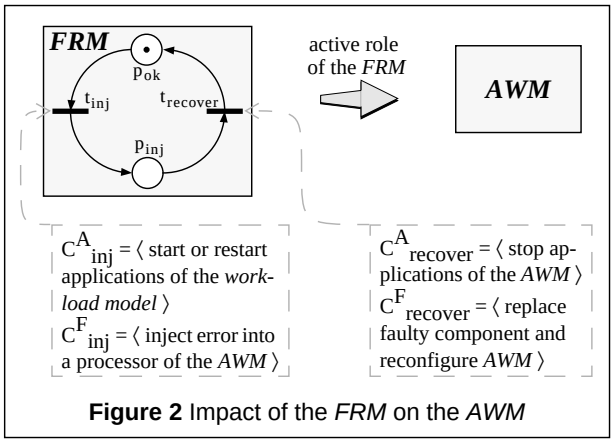

Figure 2 Impact of the *FRM* on the *AWM*

In this example, the *FRM* plays the active part, since it has activities performed within the *AWM*; there is no feedback from the *AWM* to the *FRM* and, thus, the states and the dynamic flow of the *FRM* do not depend on the *AWM*. Furthermore, the *FRM* of Figure 2 mirrors the failure state of the *AWM*. A token in p_{ok} represents the non-faulty and correctly working *AWM*; when p_{inj} holds a token, an error has occurred and the *AWM* is in the recovery stage.

5.2 Impact of the *AWM* on the temporal behavior of the *FRM*

The temporal behavior and the dynamic flow of the *FRM* may depend on the actual state and on ongoing activities of the *AWM*. In this way, the time to elapse between the enabling and firing of a transition, which is usually defined as a random variable via a distribution function, can be obtained by measuring the span between specific events of the *AWM*.

A typical example is the ascertainment of the *error latency* which describes the duration between the injection and the detection of an error. We extend the example of Figure 2 by taking into consideration the required latency time modeled by transition t_{det} in Figure 3. Futhermore, a software-based fault tolerance mechanism is loaded onto the processor objects of the *AWM* as an additional workload which runs concurrently to other applications of the workload model and which detects the processor failures in the architecture model. Thus, the error latency, which has to be considered in the *FRM*, depends on inherent parameters and on the actual characteristics of the *AWM* such as the fault tolerance mechanisms as well as on the message transfer times, the routing characteristics, and possible communication bottlenecks caused by intense message traffic. Transition t_{det} is pictured as a filled box combined with a thin bar symbolizing the fact that the time between its enabling and firing is determined by the *AWM*.

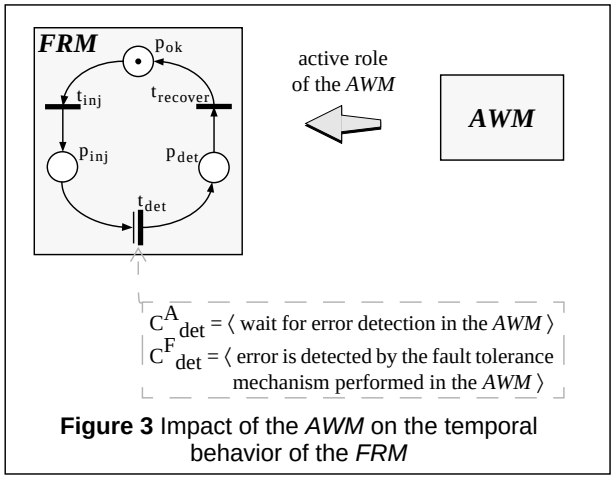

Figure 3 Impact of the *AWM* on the temporal behavior of the *FRM*

The time delay of the timed transition t_{det}, which depicts the error latency in Figure 3, is not only subject to the parameters of the *FRM* and, for this reason, cannot be decided exclusively within the *FRM*, but requires a feedback from the fault tolerance mechanism conducted in the *AWM*. As soon as t_{inj} fires, an error is injected and t_{det} is enabled; the execution of the *FRM* simulation stops and the control is given to the *AWM* simulation until the predefined event in the *AWM* occurs, i.e., until the processor failure is detected by the fault tolerance mechanism. Now, the simulation time of the *FRM* advances to the detection time and, resuming the *FRM* simulation, transition t_{det} fires. Thus, the time between enabling and firing of t_{det} is determined by operations which take place within the *AWM*. This example illustrates the impact of the *AWM* on the temporal behavior of the *FRM*.

5.3 Impact of the *AWM* on the sequence of events of the *FRM*

Furthermore, the sequence of events in the *FRM* can be directed and controlled by the actual state of the *AWM*. That means, the state of the *AWM* affects the selection of the transition to fire if several transitions are concurrently enabled. By default, a weight w_i and a priority r_i (see Def. 1) are assigned to each immediate transition t_i and are initialized before the simulation of the *FRM* is started. We allow these weights and priorities to be modified during the execution of the simulation experiments. In this way, the firing of a transition of the *FRM* can be made more (or less if desired) likely, or transitions can even be prevented from firing by setting their weights to zero (similarly, the distribution functions f_i of timed transitions and their parameters can be dynamically altered depending on the state of the

AWM to model for instance workload-dependent error injection rates).

We demonstrate this issue with the following example. We take the *FRM* of Figure 3 and extend it by allowing two different recovery policies. After error detection one of the two policies is chosen depending on the existence of spare components. Figure 4 shows the *TTPN* representation of the *FRM*. The immediate transitions t_{ic} and t_{ip} are drawn as double bars to show that their priorities and weights depend on the state of the *AWM*.

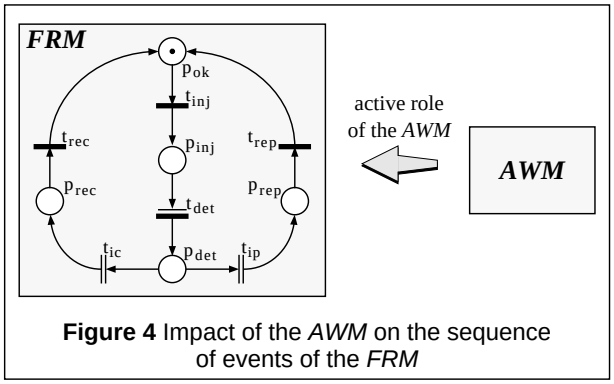

Figure 4 Impact of the *AWM* on the sequence of events of the *FRM*

We presume that the workload applications and the software-based detection mechanism, which is part of the workload model, are already running in the initial state of the *FRM* shown in Figure 4. After error detection, a token is deposited in place p_{det}. Afterwards, the immediate transition t_{ic} is enabled and fires if spare components are available. Otherwise, t_{ip} is enabled and fires. Here, we are presupposing that t_{ic} and t_{ip} have the same priority (r_{ic} is equal to r_{ip}); furthermore, the weights of the immediate transitions t_{ic} and t_{ip} are modified when t_{det} fires, that means when the error is detected. The transitions t_{ic}, t_{ip}, t_{rec}, and t_{rep} replace $t_{recover}$ in Figure 3, and model the decision for reconfiguration or repair as well as the time required for system reconfiguration and repair. t_{rec} is enabled after the firing of t_{ic}; its enabling initiates the reconfiguration of the *AWM*, and its firing depicts the accomplishment of the reconfiguration. t_{rep} is enabled after the firing of t_{ip} starting the repair of the architecture model; the repair is successfully completed when t_{rep} fires.

6 Example

In this chapter, we use *Conjoint Simulation* to model a massively parallel computer system and evaluate the impact of various fault tolerance mechanisms on the system's availability. We define *system availability* to be the probability that the target system is available for application programs at any point in time. We present the main characteristics of the *architecture-workload model* (*AWM*) as well as the structure of the *failure-repair model* (*FRM*).

6.1 *AWM* representation

6.1.1 Architecture model

The target architecture is a message-passing multiprocessor system which is similar to the Parsytec GC multiprocessor architecture presented in [10]. This massively parallel machine is designed to run scientific and technical application programs requiring huge computing power. High performance is achieved by the large number of processing elements and their interprocessor connections providing high bandwidth.

The architecture consists of a *data network*, called the *dnet*, which forms a three-dimensional grid and on which the user applications execute. Each node of the *dnet* corresponds to a *cluster* containing q processors which are fully and redundantly connected via p crossbar-like routing switches (Figure 5). Several processors per cluster can serve as spare components to replace failed processors.

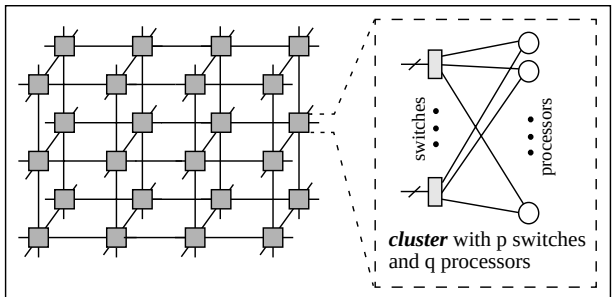

Figure 5 *Data network* of the architecture model

In addition to spare processors and redundant communication links, the target architecture contains a separate *control network* (*cnet*) which is used for supervising, monitoring, and fault recovery purposes. The smallest unit of the *cnet* is a so-called *cube* which comprises a control processor monitoring and supervising 4 clusters (Figure 6). The switch of the cube connects the control processor with a switch of each of its 4 clusters and with the neighboring cubes. Like the *dnet*, the cubes are connected together in a three-dimensional grid. Application programs do not have access to the *cnet*, however, application programs are loaded onto the processors of the *dnet* through the *cnet*. Fault tolerance algorithms for detection of component failures, reconfiguration, and recovery run on the *cnet*. In the remainder of this paper, we use the term *cnet* to denote the control processors, the link connections between the control processors, and the connections between control processors and clusters; the clusters are not part of the *cnet*.

6.1.2 Workload model

The workload model is comprised of a software-based technique to detect and localize failed components. This technique, referred to as the *heartbeat mechanism*, consists of a set of processes which are assigned to supervised processors and send messages to a predetermined control processor at specific time intervals. Processors that fail to

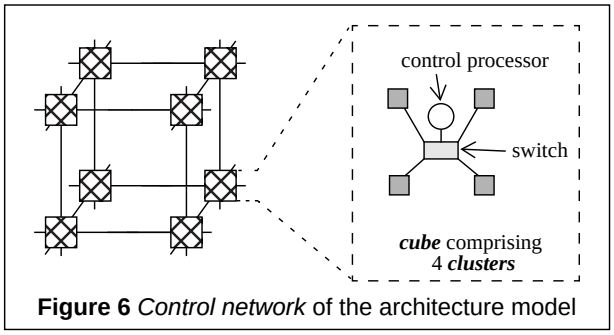

Figure 6 *Control network* of the architecture model

send a status message over a specified time period are considered to be faulty by the control processor.

In the target architecture, heartbeat messages are sent by application processors (*dnet*) to their control processors (*cnet*) every T_{AL} time interval. A control processor updates the last time stamp of a processor as soon as it receives a heartbeat message from the processor. Every T_{SV} time units a process running on every control processor checks the last time stamps of its associated *dnet* processors and assumes that a processor is faulty if its last time stamp is older than T_{DM} time units. The frequency of the heartbeat messages, $1/T_{AL}$, has a large impact on the efficiency of this error detection mechanism. On the one hand, frequent heartbeat messages reduce the time to detect component failures and may avoid error propagation. On the other hand, the overhead of processing these messages can adversely impact the execution time of the concurrent application programs.

Failures of *cnet* processors are detected by another heartbeat mechanism. Each control processor periodically sends a heartbeat message to the control processors of its (at most 6) surrounding cubes. Thus, each control processor receives heartbeat messages of its neighboring control processors and is able to detect their failures.

6.2 *FRM* representation

The *FRM* is represented as a *TTPN* model which is organized in three parts shown in Figure 7. The failure-repair behavior of the *dnet* processors is represented by $TTPN_{dnet}$. As long as the processors of the *dnet* are fault-free, a token is positioned in p_{dok}; a token in p_{dok} indicates an error-free and available *dnet*. When transition t_{dinj} fires, an error is injected into a processor object of the *AWM*; furthermore, the token is removed from p_{dok} and a token is placed in p_{dinj}. Transition t_{ddet} fires when the failure of the processor is detected by the heartbeat mechanism executed in the *AWM*. After the detection of the failed processor, both immediate transitions t_{dic} and t_{dip} are enabled. If the architecture model has available spare processors in the *dnet*, t_{dic} fires and a token is placed in p_{drec}; otherwise, t_{dip} fires and p_{drep} obtains a token. Transition t_{drec} models the time required for the reconfiguration of the *AWM*, i.e., for possible modifications of the routing tables, for changes of the logical addresses, for the remapping of the workload model onto the architecture model, and for the roll back and restart of the workload applications. The *dnet* is repaired, that means all failed processors are set to the fault-free state, as soon as t_{drep} fires; the enabling time of t_{drep} models the time needed for repair.

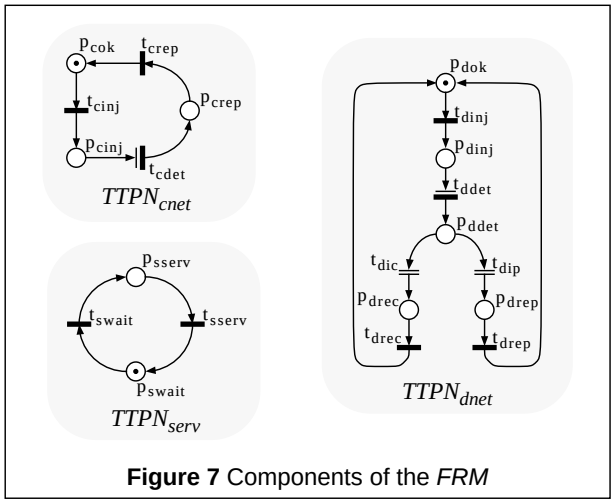

Figure 7 Components of the *FRM*

$TTPN_{cnet}$ models the failure-repair behavior of the *cnet* and is similar to $TTPN_{dnet}$. The *AWM*, however, cannot be reconfigured in the case of the failure of a *cnet* processor, because the *cnet* does not provide spare control processors. In the initial fault-free state of the *cnet*, a token is positioned in p_{cok}. When t_{cinj} fires, an error is injected into a *cnet* processor and, after the removal of the token in p_{cok}, a token is put into place p_{cinj}. The time for detection of the processor failure in the *cnet* is symbolized by t_{cdet} and is determined by the heartbeat mechanism. When a token is located in p_{crep}, the *cnet* is just being repaired; all the *cnet* processors are fault-free after the firing of t_{crep}.

The third part of the *FRM*, $TTPN_{serv}$, represents a periodically performed maintenance service comprising repair and upgrade of the architecture model. Transition t_{swait} models the time between two successive maintenance services called the *service interval*, while t_{sserv} depicts the duration of a service session. It is obvious that there is only one token in each of the three parts of the *FRM*; the respective marking shows the state of the *dnet*, the *cnet*, or the maintenance service.

The accomplished *FRM* is sketched in Figure 8. Some auxiliary places, transitions, and arcs are inserted to connect the three parts. In order to link $TTPN_{dnet}$ and $TTPN_{cnet}$ the place p_{c1} and the transitions t_{c1}, t_{c2}, t_{c3}, and t_{c4} are added. A token in p_{c1} enables the four immediate transitions t_{c1}, t_{c2}, t_{c3}, and t_{c4}; when one of these transitions fires, the token from one of the places of $TTPN_{dnet}$ is removed. The interpretation is as follows. As soon as a component failure is detected in the *cnet* of the architecture model, every ongoing operation of the *dnet* is stopped; ongoing workload applications are interrupted, and reconfiguration or repair procedures of the *dnet* are stopped. The repair of the *cnet* includes the repair of the *dnet*, and after

the firing of t_{crep} a token is positioned in both places p_{cok} and p_{dok}.

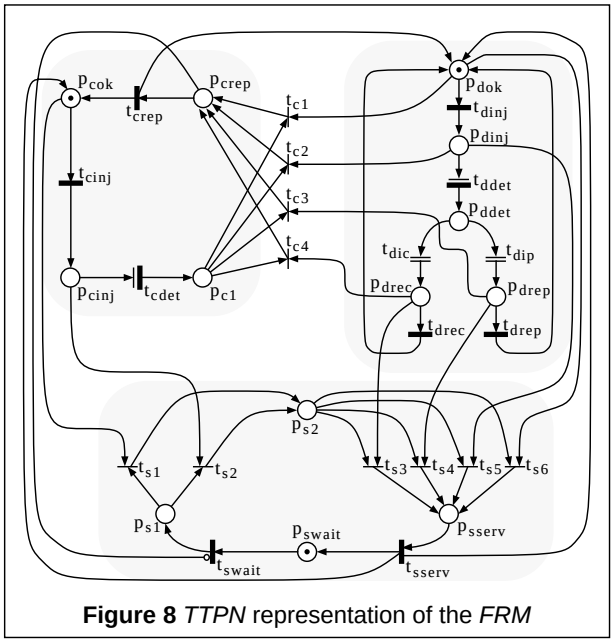

Figure 8 *TTPN* representation of the *FRM*

The connection of $TTPN_{dnet}$ and $TTPN_{cnet}$ with $TTPN_{serv}$ requires the insertion of two additional places p_{s1} and p_{s2} and of six new immediate transitions t_{s1}, t_{s2}, t_{s3}, t_{s4}, t_{s5}, and t_{s6}. The token is withdrawn from $TTPN_{cnet}$ by the firing of t_{s1} or t_{s2}. Furthermore, the token is removed from $TTPN_{dnet}$ when one of the four transitions t_{s3}, t_{s4}, t_{s5}, or t_{s6} fires. The inhibitor arc between p_{crep} and t_{swait} prevents t_{swait} from being enabled as long as p_{crep} holds a token; that means the maintenance service is not started if the *cnet* and the *dnet* are just being repaired. Periodical repair and upgrade tasks can be included in the ongoing repair stage, and there is no need to perform an additional maintenance service. This fact is modeled by the inhibitor arc and by the *R-E* memory policy (enabling memory) of transition t_{swait} (Table 1); when p_{crep} holds a token, t_{swait} cannot be enabled and the time to wait for the successive maintenance service is restarted from scratch after the firing of t_{crep}. Figure 8 shows the initial marking of the whole *TTPN*.

The priorities, weights, and distribution functions as well as the memory policies of the *FRM* are listed in Table 1. The interactions between *AWM* and *FRM* are presented in Table 2. The weights and enabling times of the transitions t_{dic}, t_{dip}, t_{ddet}, and t_{cdet} are determined in the *AWM*. The weights w_{dic} and w_{dip} of the immediate transitions t_{dic} and t_{dip} are set in accordance with the state of the *AWM* when transition t_{ddet} fires. If suitable spare processors are available in the *dnet* t_{dic} has to fire; in this case, w_{dic} is set to 1.0 and w_{dip} gets the value 0.0. If there are no suitable spare processors t_{dip} fires, i.e., t_{dic} obtains the value 0.0 and t_{dip} is 1.0.

Transition t_i	Priority r_i, Weight w_i / Distribution Function f_i	Memory Policy e_i
t_{dinj}	exponential distribution	*R-E*
t_{ddet}	determined in *AWM*	*R-E*
t_{dic}, t_{dip}	priority: 1.0; weight: determined in *AWM*	⊥
t_{drec}	normal distribution	*R-E*
t_{drep}	normal distribution (dependent on the number of failed *dnet* processors)	*R-E*
t_{cinj}	exponential distribution	*R-E*
t_{cdet}	time between enabling and firing is determined in *AWM*	*R-E*
t_{crep}	normal distribution (dependent on the number of failed *cnet* processors)	*R-A*
t_{swait}	constant value (see Section 6.3)	*R-E*
t_{sserv}	normal distribution (dependent on the number of failed *dnet* and *cnet* processors)	*R-A*
t_{ci}, t_{si}	priority: 1.0; weight: 1.0	⊥

Table 1 Temporal and probabilistic characteristics of the *FRM* transitions

Transition t_i	Enabling Operations C^A_i	Firing Operations C^F_i
t_{dinj}	start heartbeat mechanism on *dnet*	inject error in *dnet* processor
t_{ddet}	determine latency time in the *AWM*	stop heartbeat mechanism on *dnet*; determine weights of t_{dic} and t_{dip}
t_{dic}, t_{dip}	∅	∅
t_{drec}	∅	reconfigure *dnet*
t_{drep}	∅	repair *dnet*
t_{cinj}	start heartbeat mechanism on *cnet*	inject error in *cnet* processor
t_{cdet}	determine latency time in the *AWM*	stop heartbeat mechanism on *cnet* and *dnet*
t_{crep}	∅	repair *cnet* and *dnet*
t_{swait}	∅	∅
t_{sserv}	stop heartbeat mechanisms on *dnet* and on *cnet*	perform maintenance service and repair *cnet* and *dnet*
t_{ci}, t_{si}	∅	∅

Table 2 Interactions of *FRM* and *AWM*

The latency times symbolized by the transitions t_{ddet} and t_{cdet} depend on the detection mechanisms performed in the *AWM*. When one of these two transitions or both are enabled, the *FRM* simulation is blocked until the corresponding processor failure is detected by the heartbeat mechanism of the *AWM* simulation, or until t_{swait} fires starting the maintenance service.

6.3 Failure, repair, and maintenance scenarios

The following experiments examine two different topologies of the architecture. Both provide the same number of available processors of the *dnet*. The first architecture ARCH_A consists of 4 cubes, 2 in X- and 2 in Y-direction, and each cluster comprises 8 processors which can be used by application programs. The second architecture ARCH_B consists of 2 cubes in X-direction, and each cluster contains 16 accessible *dnet* processors. Thus, 128 processors are at the users' disposal in both architectures. In

order to tolerate permanent processor failures, each cluster has additional 1, 2, or 3 spare processors which replace faulty *dnet* processors; in the following the respective architectures are termed ARCH_A_1, ARCH_A_2, or ARCH_A_3, and analogously ARCH_B_1, ARCH_B_2, or ARCH_B_3.

Four experiments are conducted to determine the impact of the number of spares, service time intervals and different detection schemes on system availability. We define *system availability* as the probability that application programs can be executed on the target system and the target system is fault-free. The target system is available when the *dnet* is available, and system availability corresponds to the probability that place p_{dok} of the *FRM* shown in Figure 8 holds a token.

6.3.1 *Experiment 1*

The main structure for *Experiment 1* is depicted in the *TTPN* representation of Figure 8. The heartbeat mechanism is initialized and mapped on the architecture model in order to detect failures of *dnet* and *cnet* processors. The heartbeat messages are sent once per minute (T_{AL} = 1 min.); the supervising process, which is executed by each control processor, checks the time stamps every 3 minutes (T_{SV}), and reports the failure of a processor if its last time stamp is older than 10 minutes (T_{DM}). Errors are injected into the processors of the *cnet* and of the *dnet*; spare processors are not affected by error injections. Furthermore, we rely on the *fail-stop* assumption, i.e., a faulty processor does not execute application processes and does not send or receive messages.

After the heartbeat mechanism has detected the failure of a *dnet* processor, the *dnet* is reconfigured, i.e., a spare processor replaces the faulty component if spare processors are still available. Note that in this experiment, we assume that the spare processor can *come from any cluster in the system and not just the local cluster*. Otherwise, the *dnet* has to be repaired; a repair is more time-consuming than a reconfiguration, and all the *dnet* processors are considered fault-free after repair.

Failure detection in the *cnet* is based on a heartbeat mechanism between the control processors. Since there are no redundant or spare *cnet* processors, the overall *cnet* has to be repaired when a *cnet* processor fails and the failure is detected. A *cnet* repair includes the repair of the *dnet* and, thus, the *cnet* as well as the *dnet* are fault-free after a *cnet* repair. A periodically executed maintenance service checks the overall system, i.e. the *cnet* and the *dnet*, and replaces faulty and decrepit components. The architecture is fault-free after the maintenance service.

The results of *Experiment 1* are shown in Figure 9. The last item of the x-axis called *inf* indicates that no maintenance service is performed (the length of the service interval modeled by transition t_{swait} is infinite). If the service interval is small (1 or 3 days), the number of spare

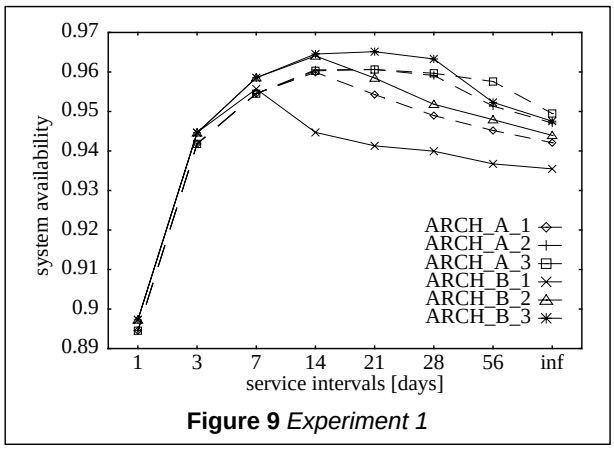

Figure 9 *Experiment 1*

processors does not have any impact on the system availability. For such short service intervals, ARCH_A has a slightly lower system availability than ARCH_B; both architectures have the same total number of available *dnet* processors which can be affected by errors occurring in the *dnet*, but ARCH_A contains 4 control processors, while ARCH_B has only 2 control processors. Under the assumption that *cnet* processor failures are independent and identically distributed, the probability that a *cnet* processor fails in ARCH_A is twice as large as in ARCH_B.

The architecture with the smallest total number of spare processors, namely ARCH_B_1, has its highest availability if the service interval is 7 days; ARCH_A_1 has twice as many spare processors and reaches its largest availability if the service interval is 14 days. ARCH_B_2 is most available if the service interval is 14 days; if the service intervals are longer, the availability of ARCH_A_2 is higher and has its maximum at 21 days. Regarding the architectures with 3 spare processors per cluster, ARCH_A_3 and ARCH_B_3, the availability of ARCH_A_3 is higher only for large service intervals (56 days) or if no maintenance service is performed at all. For smaller service intervals, the larger number of *cnet* components renders ARCH_A_3 more vulnerable, and this illustrates the recursive character of fault tolerance. The best system availability (0.965) is achieved by ARCH_B_3 with a service interval of 21 days.

To sum up, we can say that spare processors are required and this redundancy gains more and more importance as the service intervals become longer. Very short service intervals of 1 or 3 days unnecessarily decrease the system availability because a lot of time is spent in an idle state during the maintenance services.

For the sake of comparison, we have simulated ARCH_A and ARCH_B without any spare processors; the availability values are in the range of 0.794 to 0.817. They show a quite low system availability and motivate the need of at least one spare processor per cluster.

6.3.2 *Experiment 2*

In *Experiment 2*, we study the impact on availability when *dnet* processor reconfigurations can take place *only if*

there is a spare in the same cluster. This is a more realistic limitation than the one assumed in the first experiment because the process of locating and reconfiguring with spares located in other clusters is significantly more complex. It is not a coincidence that this simpler scheme is the one originally designed for the Parsytec GC system.

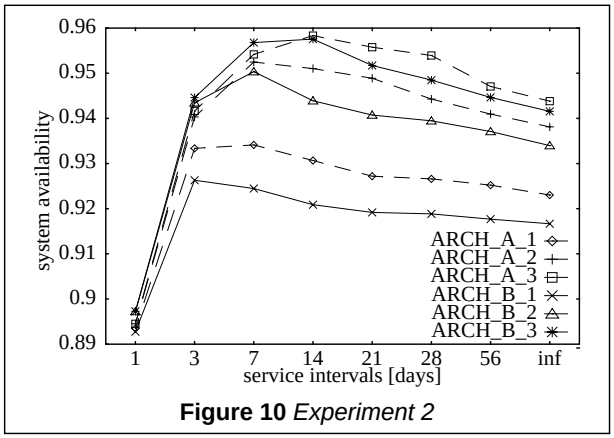

Figure 10 *Experiment 2*

Clearly for *Experiment 2* (Figure 10), the number of spare processors per cluster is much more important than it was for *Experiment 1*. Furthermore, the larger total number of spare processors of ARCH_A results in a larger system availability than ARCH_B even in the case of small service intervals and despite the larger vulnerability because of the larger number of *cnet* processors. ARCH_A_1 and ARCH_B_1 require short service intervals of 3 or 7 days to achieve their highest system availability. ARCH_A_3 provides the best system availability if the time between service intervals is at least 14 days or if there is no service interval at all. Just as in *Experiment 1*, a service interval of 1 day is too short and decreases the system availability below 90 percent. Comparing the results of *Experiment 2*, the best system availability (0.958) is obtained if ARCH_A_3 and a service interval of 14 days are chosen.

6.3.3 *Experiment 3*

In *Experiment 3*, we assume that if a *cnet* processor fails or if a *dnet* processor fails and there are no spare processors (local or global), the system is not repaired until the next service interval. To model this behavior, only some slight modifications of the *TTPN* representation of the *FRM* shown in Figure 8 are necessary. Note that this further illustrates the advantage of a logical separation of the entire system model into its *AWM* and *FRM* submodels.

The $TTPN_{Experiment\ 3}$ (Figure 11) represents the new *FRM*. Having taken the *TTPN* of Figure 8, we have removed the transitions t_{drep} and t_{crep}, which have represented immediate repair, as well as input and output arcs linked to these transitions. Furthermore, the inhibitor arc between p_{crep} and t_{swait} has become superfluous and has been taken off. A new immediate transition t_{s7} is inserted to remove the token from p_{crep} after t_{swait} has fired; this transition models the start of a maintenance service given that a

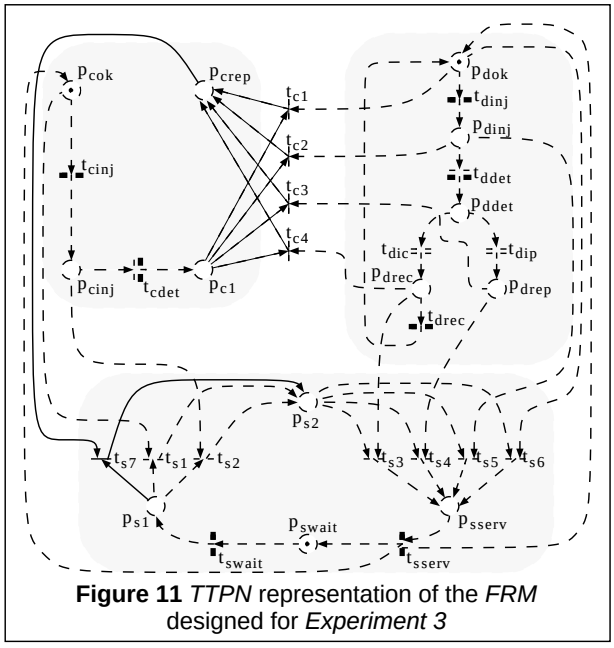

Figure 11 *TTPN* representation of the *FRM* designed for *Experiment 3*

cnet processor is failed. Transition t_{s7} has an input arc from p_{crep} and p_{s1} as well as an output arc to p_{s2}; it has the same temporal and probabilistic properties as transition t_{s1} and it has no interaction with the *AWM*, i.e., $C^A_{s7} = C^F_{s7} = \varnothing$. In Figure 11, the parts of the *TTPN* which have been taken over from Figure 8 are drawn with dashed borders and lines; the new components are drawn using solid lines.

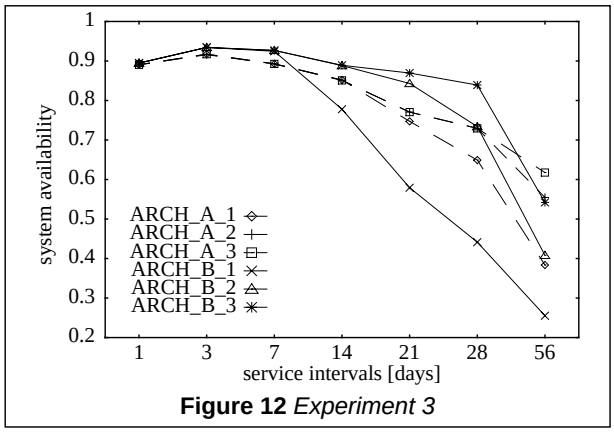

Figure 12 *Experiment 3*

It is obvious that the system availability deteriorates substantially when immediate repair is not provided. All the architectures have their highest system availability for the short service interval of 3 days (Figure 12). The increase in the length of the service intervals decreases the system availability rapidly; for instance, only architectures ARCH_B_2 and ARCH_B_3 have an availability larger than 80 percent for a service interval of 21 days. Interestingly, ARCH_A_2 has a higher availability than ARCH_B_3 when the service interval is quite large (56 days). This is due to the larger total number of spare processors (32 vs. 24 spare processors) in ARCH_A_2; this fact was not evident in the results from the previous experi-

ments. Summarizing, it can be said that only very short service intervals provide acceptable system availability when immediate repair is not possible.

6.3.4 *Experiment 4*

Experiment 4 studies the impact of a low overhead detection scheme which replaces the heartbeat mechanism. In this scheme a program is started periodically to check the state of the *dnet* and *cnet* processors. The test program takes 10 minutes to check the overall architecture, and 2 hours pass between the end of the test program and the start of the successive one. The test program is considered perfect, i.e., if there is a failed processor it will be detected. It is assumed to run concurrently with other application programs. Apart from the mechanism for the detection of failed processors, the same assumptions are made as in *Experiment 1*.

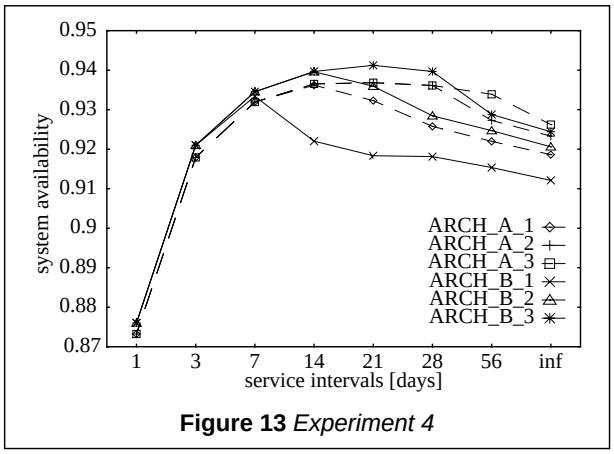

Figure 13 *Experiment 4*

Because of the nearly identical assumptions, the shape of the curves of *Experiment 1* (Figure 9) and *Experiment 4* (Figure 13) are similar. However, the availability values of all the architectures are noticeably higher in *Experiment 1* than in *Experiment 4* since the latency time, the time span between occurrence and detection of processor failures, is much shorter in *Experiment 1*. The frequently sent heartbeat messages guarantee rapid detection of processor failures, while the test program of *Experiment 4* is performed less frequently. The highest system availability (0.941) of *Experiment 4* is attained by ARCH_B_3 at a service interval of 21 days.

7 Conclusion

We have presented the concept and a case study of *Conjoint Simulation*, a powerful and flexible approach for the combined performance and dependability analysis of large-scale computer systems. *Conjoint Simulation* facilitates the representation, development, and maintenance of sophisticated simulation models considering the target architecture, actual workload, and various failure-repair modes. Model evaluation is enhanced by the independent development and combined analysis of *AWM* and *FRM* as well as by their intuitive combination. We have illustrated, especially in *Experiment 3*, the benefits of such a logical separation where only minor, local modifications to a submodel are necessary to change the system behavior. Intensive work is going on to analyze more complex models depicting sophisticated multiprocessor architectures, workload scenarios, and fault tolerance mechanisms which could not be discussed within this paper. Furthermore, we are investigating the use of enhanced versions of fault trees and series-parallel diagrams in order to further facilitate the representation of failure-repair dependencies for large-scale fault-tolerant computer systems.

Literature

[1] Ellermeier, M. *Hierarchische Modellerstellung mit der Simulationsumgebung SimPar*. Studienarbeit (Practical Work), IMMD III, University of Erlangen-Nürnberg, 1995, Advisor: A. Hein.

[2] Goswami, K. K. and R. K. Iyer. *DEPEND: A Simulation-Based Environment for System-Level Dependability Analysis.* Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1992.

[3] Goswami, K. K. *Design for Dependability: A Simulation-Based Approach.* Ph.D. Thesis, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1993.

[4] Goswami, K. K and R. K. Iyer. *Simulation of Software Behavior under Hardware Fault* in *Proceedings of the 23rd Intern. Symp. on Fault-Tolerant Computing,* Tolouse, France, 1993.

[5] Hein, A. and K. Bänsch. *SimPar - A Simulation-Based Environment for Performance and Dependability Amalysis of User-Defined Fault-Tolerant Parallel Systems*. Internal Report 1/95, IMMD III, University of Erlangen-Nürnberg.

[6] Hein, A. and W. Hohl. *Simulation-Based Performability Analysis of Multiprocessor Systems.* To appear in *Periodica Polytechnica*, University of Budapest, Hungary, 1995.

[7] Marsan, M. A., G. Balbo, and G. Conte. *A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems* in *Transactions on Computer Systems (acm)*, Vol. 2, No. 2, May 1984.

[8] Marsan, M. A., et al. *Generalized Stochastic Petri Nets Revisited: Random Switches and Priorities* in *Proceedings of the International Workshop on Petri Nets and Performance Models*, Madison, WI, USA, August 1987. IEEE Computer Society Press, 1987.

[9] Marsan, M. A. et al. *The Effect of Execution Policies on the Semantics and Analysis of Stochastic Petri Nets* in *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, July 1989.

[10] Parsytec. *The Parsytec GC Technical Summary*, Version 1.0. Aachen (Germany), Parsytec Computer GmbH, 1991.

[11] Schwetman, H. *Hybrid Simulation Models of Computer Systems* in *Communications of the ACM*, Vol. 21, No. 9, September 1978.

[12] Simon, G. *Workload Modeling with SimPar*. Diplomarbeit (Diploma Work), IMMD III, University of Erlangen-Nürnberg, 1995, Advisors: S. Dalibor and A. Hein.

[13] Trivedi, K. S. et. al. *Analysis of Typical Fault-Tolerant Architectures using HARP* in *IEEE Transactions on Reliability*, Vol. R-36, No. 2, June 1987.