Annals of Software Engineering
Discontinued

Article Abstract

Export Citation: Text RIS
doi:10.1023/A:1018927430556

Annals of Software Engineering
8 (1-4): 239-287, 1999
Copyright © 1999 Kluwer Academic Publishers
All rights reserved

Composing, analyzing and validating software models to assess the performability of competing design candidates

Frederick T. Sheldon
School of Electrical Engineering and Computer Science, Washington State University, PO Box 642752, Pullman, WA 99164-2752, USA E-mail: Sheldon@eecs.wsu.edu

Stefan Greiner
Performance Modeling & Process Control Research Group, Department of Computer Science IMMD IV, The University of Erlangen-Nürnberg, Martensstrasse 1, 91058 Erlangen, Germany E-mail: Stefan.Greiner@daimlerchrysler.com

In a perfect world, verification and validation of a software design specification would be possible before any code was generated. Indeed, in a perfect world we would know that the implementation was correct because we could trust the class libraries, the development tools, verification tools and simulations, etc. These features would provide the confidence needed to know that all aspects (complexity, logical and timing correctness) of the design were fully satisfied (i.e., everything was right). Right in the sense that we built it right (it is correct with respect to its specification) and it solves the right problem. Unfortunately, it is not a perfect world, and therefore we must strive to continue to refine, develop and validate useful methods and tools for the creation of safe and correct software. This paper considers the analysis of systems expressed using formal notations. We introduce our framework, the modeling cycle, and motivate the need for tool supported rigorous methods. Our framework is about using systematic formal techniques for the creation and composition of software models that can further enable reasoning about high-assurance systems. We describe several formal modeling techniques within this context (i.e., reliability and availability models, performance and functional models, performability models, etc.). This discussion includes a more precise discourse on stochastic methods (i.e., DTMC and CTMC) and their formulation. In addition, we briefly review the underlying theories and assumptions that are used to solve these models for the measure of interest (i.e., simulation, numerical and analytical techniques). Finally, we present a simple example that employs generalized stochastic Petri nets and illustrates the usefulness of such analysis methods.

Article ID: 326228

Subscription Info | Customer Service | Sales Kit | Kluwer Alert | Feedback | Help
Copyright | Terms and Conditions | Privacy Policy

©2004 Kluwer. All rights reserved.

To Appear: Annals of Software Engineering 1999: Special Volume on Software Reliability, Testing and Maturity

COMPOSING, ANALYZING AND VALIDATING SOFTWARE MODELS TO ASSESS THE PERFORMABILITY OF COMPETING DESIGN CANDIDATES

Frederick T. Sheldon[1]
School of Electrical Engineering and Computer Science
Washington State University
PO Box 642752
Pullman, Washington 99164-2752, USA
E-mail: Sheldon@eecs.wsu.edu

Stefan Greiner
Performance Modeling & Process Control Research Group
Department of Computer Science IMMD IV
The University of Erlangen-Nürnberg
Martensstrasse 1, 91058 Erlangen, Germany
E-mail: Stefan.Greiner@daimlerchrysler.com

[1] This work was partially supported by NASA ARC / Stanford ASEE faculty program fellowship (Automated S/E group) and the University of Erlangen Visiting Scholar program.

ABSTRACT

In a perfect world, verification and validation of a software design specification would be possible before any code was generated. Indeed, in a perfect world we would know that the implementation was correct because we could trust the class libraries, the development tools, verification tools and simulations etc. These features would provide the confidence needed to know that all aspects (complexity, logical and timing correctness) of the design were fully satisfied (i.e., everything was right). Right in the sense that we built it right (its correct with respect to its specification) and it solves the right problem. Unfortunately, it's not a perfect world, and therefore we must strive to continue to refine, develop and validate useful methods and tools for the creation of safe and correct software.

This paper considers the analysis of systems expressed using formal notations. We introduce our framework, the *modeling cycle*, and motivate the need for tool supported rigorous methods. Our framework is about using systematic formal techniques for the creation and composition of software models that can further enable reasoning about high-assurance systems. We describe several formal modeling techniques within this context (i.e., reliability and availability models, performance and functional models, performability models etc.). This discussion includes a more precise discourse on stochastic methods (i.e., DTMC and CTMC) and their formulation. In addition, we briefly review the underlying theories and assumptions that are used to solve these models for the measure of interest (i.e., simulation, numerical and analytical techniques). Finally, we present a simple example that employees generalized stochastic Petri nets and illustrates the usefulness of such analysis methods.

1. Introduction

Modern high-assurance systems share five key attributes: (1) reliable, meaning they are correct, (2) available, meaning they remain operational, (3) safe, meaning they are impervious to catastrophe (fail-safe), (4) secure, meaning they will never enter a hazardous state, and (5) timely, meaning their results will be produced in time to satisfy deadlines (i.e., timing correctness). The correctness, safety and robustness of a critical system specification are generally assessed through a combination of rigorous specification capture and inspection; formal modeling and analysis of the specification; and execution and/or simulation of the model/specification.

In our view of the world, we constrain the classic notions of verification and validation to first confirming that at least all of the possibly bad things we could think of can not happen or at least the chances of happening are well bounded. Second, that the performability of the models (or blueprints) of the systems (i.e., system models) that we *plan* or propose building are determined to be adequate with respect to function, structure and behavior and with respect to the assumptions of the operating environment and potential hazards. Moreover, our goal is to refine our approach as a framework for the formal *representation* and *analysis* of software components and architectures that relate specifications to programs and programs to behavior. Thus, in our view of the world the focus is tool-supported rigorous methods for reasoning about software and systems as a constructive approach to software design and evolution.

Table 1. Modeling techniques described in Section 3.

Modeling Approach	Model types	Para
Reliability block diagrams (RBD)	Structuring to show how the reliability of individual components affect the overall system reliability.	3.1.1
Reliability graphs	Provides a generic ordering of components from source to sink useful in reliability analysis.	3.1.2
Fault trees	Safety / hazard analysis.	3.1.3
Stochastic process algebras	Performance using stochastic annotations (advantage: composability of parallel systems).	3.2.1
Precedence graphs	Dependency among processing entities showing task partitioning and processor allocation.	3.2.2
Markov - Stationary DTMC / CTMC - Transient CTMC	Availability, reliability and performance (both steady state and transient behavior).	3.2.3
Queuing networks Local - global balance Product form solutions	Performance measures independent of time (i.e., statistical equilibrium or steady state); transient solutions of *only* simple queuing systems are available in closed form: measures include number of jobs in the system, queue length, utilization, throughput, response and wait time.	3.2.4
Markov reward models	Availability.	3.3.1
Petri Nets (PNs)	Mathematical models of systems whose dynamics are characterized by concurrency, synchronization, mutual exclusion and conflict.	3.4.1
Generalized Stochastic PNs (GSPN)	Performance analysis of distributed systems.	3.4.2
Extensions to GSPNs	Constructs designed to reduce the complexity of the GSPN representation form while maintaining the fidelity of the solution results.	3.4.3
Process algebras and SPN combined	Combined functional and stochastic analysis.	3.4.4

This article first presents some important issues as they relate to today's software engineering challenges. Second, we introduce a framework for the modeling and analysis of systems and software called the modeling cycle.

Third, inside the context of this framework, we review a set of known computational modeling techniques which are organized by their measures of interest (i.e., performance, reliability, availability etc.) as summarized in Table 1. This review discusses the rationale, approach, challenges and caveats associated with their solution methods. Finally, a fairly simple example is given to illustrate the usefulness of such techniques.

1.1. The challenges that lie ahead for software and systems

In a recent collection of articles [Yen 1998], some important issues associated with practical high assurance design strategies are reported and the challenges that lay ahead. These articles highlight key applications such as ensuring passive safety (i.e., not requiring a system to initiate any action to arrive at a safe state) for a missile warhead; developing *highly reliable* miniaturized avionics for long-life deep-space applications that exhibit continuous system operation for up to 15 years *and* is highly evolvable (e.g., subject to preventive/corrective on-board maintenance); using fault tolerance to ensure continuously available systems because the cost of downtime is extremely high; avoiding unplanned (or even planned) outages by using predictive techniques which are based on behavioral models of such systems; capturing and creating realistic (i.e., faithful) models of all relevant environmental and system requirements and verifying the various attributes of the system with respect to the model(s); and closing the gap from specifications and models to synthesized programs using some combination of domain theory and reuse library of high integrity components (i.e., providing formal traces extracted from the proofs that relate specifications to programs and programs to execution).

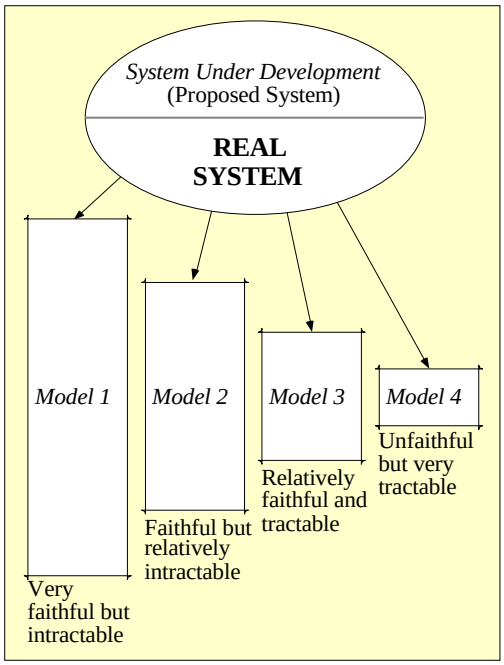


Figure 1. A model is always a compromise between faithfulness and simplicity.

The common denominator among all these challenges is complexity. Complexity in such systems is one of, if not the most important properties that make the design and implementation of high assurance systems so difficult. Furthermore, as the complexity of future systems increase, the more important it becomes to evaluate and predict their behavior. To better understand the complexity, it is common practice to create a model. Hence the term *model-based specification*, used widely in software engineering, came into being [Sommerville 1996].

1.2. The compromise between realism and simplicity

The eventual customer of a new (or enhanced) system desires one which is reliable, responsive and has a reasonable cost. If any single one of these qualities is missing, it could mean the eventual ruin of the customer and perhaps the supplier. If a legacy or prototype system exists, then it is possible to answer many questions with regards to performance and dependability by direct measure. If not, then we must defer to techniques that allow us to predict the behavior of a system during the conception and specification, and to modify the design before actual implementation. Figure 1 shows the relation between the realism (or faithfulness) and simplicity (tractability) of a model [Marie 1993]. How closely a model mirrors its originator or the vision of the new system is in direct conflict

with how easily and efficiently the model can be analyzed (i.e., solved with respect to its predicted behavior). One must consider only those facets of the system that are important to the behavior of interest.

1.3. Conquering complexity using parallel computers

Parallel computers with large amounts of main memory gain more and more importance when analyzing models. For the model, it does not matter if we solve it on a sequential or parallel computer. However, the problem that often occurs is that many of the underlying analysis algorithms are designed for sequential machines. Since main memory is not as big a problem in today's machines, the focus has shifted to how can the algorithms (problem to be solved) be partitioned over some number of n processors. If this is not done properly, the communication overhead between processors will kill the advantage of having multiple processors solving the same problem simultaneously. Much research is underway to extend these algorithms to parallel machines.

2. Creation and composition of software models

The basic modeling philosophy begins by identifying the essential components of the system and the different ways they interact. This means that many assumptions will be introduced. In making these assumptions we must keep in mind the following three basic modeling considerations: (1) simplicity, (2) ease of evaluation, and (3) adequacy of measurements [Lazowska 1984]. The key to a successful modeling examination is the skill to introduce assumptions. The principles of the modeling cycle are shown in Figure 2a.

As was mentioned at the beginning, today's real life systems are typically *very* complex. This fact prevents us from making a direct analysis. Therefore a series of abstraction steps (two basic steps described here) are necessary to obtain system measures from the real system. All these steps together give the modeling cycle. Initially, the system model is created at an abstract level (i.e., abstract model) that specifies the system at the five-hundred foot level.[2] To do an analysis of the system we need measures of the system using some type of measurement technique (i.e., we start at the Real System place, top of Figure 2a). The Proposed System place is detailed later, but without a real system to compare to, a great deal of work is needed to explore the parameter space.

The data collected from system measurements are used to parameterize the abstract model. However, usually the system model will still contain too many details that prevent an efficient system analysis. In a second abstraction step the computational model is created (one-thousand foot level) which allows an easier and more efficient system analysis. The computational model can be considered to be the highest level of model abstraction. The process of refining the computational model is a matter of building confidence in the model. Thus, in Figure 2a the process of operational validation is performed which results in a modified system and with modified system input parameters/data. This step can be repeated until the computed performance measures fulfill the requirements.

If there exists no way to gather system measurements for the purpose of parameterizing the system model then there may be a great deal of more work to do. In addition, the stopping rule for performing operational validation is now based on relative comparisons from one iteration to the next. Different model parameterizations are used to compare the different design (or implementation) candidates with the goal of making architectural design decisions.

[2] Five hundred-foot level is a terminology used here to assign a number to the degree of abstraction employed. It's a relativistic term, so if you think of ground level (or zero feet), that is at a very low (e.g., the bare machine) system level. The thousand-foot level is more abstract than say the five hundred-foot level. The thousand-foot level here roughly corresponds to employing additional simplifying assumptions.

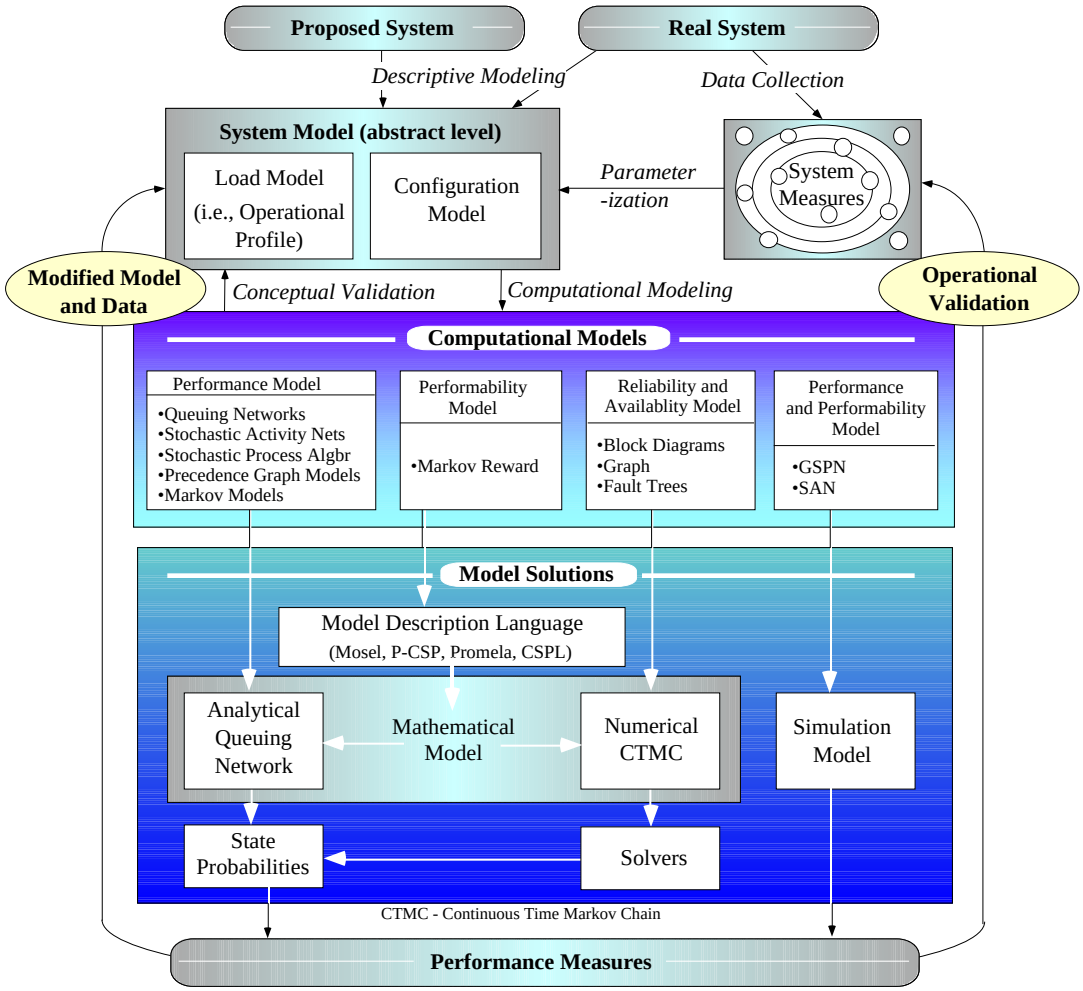


Figure 2a. The modeling cycle shows the principle steps of the modeling process.

In simple terms, using a technique commonly known as sensitivity analysis (see paragraph 1.1.5) to optimize the model's structure toward achieving critical functional and non-functional requirements [Blake 1988; Heidelberger 1988; Choi 1993; Mainkar 1993].

2.1. Model parameterization out of data collection

During the elaboration of a model, the distinction between discrete and continuous variables is made. Introducing a continuous variable facilitates the model solution (i.e., some problems are easier to solve in the continuous domain compared to the discrete case). The only restriction comes if the simplification gives unhappy consequences with regards to the validity of the result.

Another simplification is made if the elaboration of a very faithful model introduces a large number of deterministic variables (family(s) thereof) and would result in an intractable model. A common simplification consists of replacing a family of n deterministic variables $x_1, \ldots, x_n$ with a unique random variable X the distribution of which depends on the characteristics of the variables of $x_1, \ldots, x_n$. This simplification pertains to the model and it is of course the model that is random and not the real system. It is also important not to go too far, for example replacing the X (random variable) with $\bar{x}$ = E[X]. When there is a lack of information on the variables

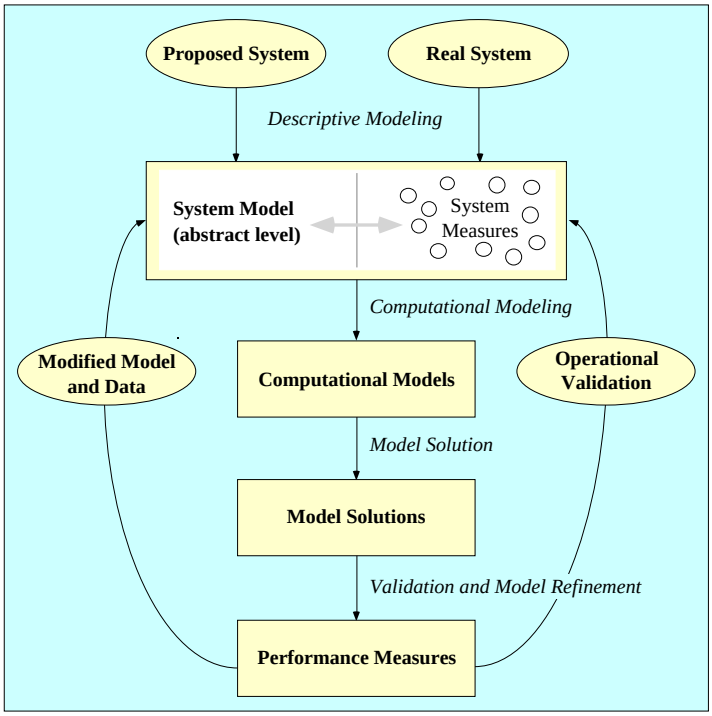


Figure 2b. Modeling-cycle emphasizing the principle steps.

("parameters" in Figure 2a) using a probabilistic universe is a place often visited which then leads to the use of random variables.

If we are lucky, measured data can be the major source for parameterization. System measurement can help in the process of deciding which components of the system are important in regards to the measure of interest. It is useless to measure for instance the average response time of a computer system if what we are interested in is the mean time to failure (naturally, that also depends on how we define failure). We must stand back and think about the modeling purpose before the commencing the measurement process.[3]

2.2. Descriptive modeling

The result of descriptive modeling is a formal specification. In this step, we abstract from all the system details those, which are not necessary to the purpose(s), defined for the model (e.g., the color or weight of the system). The system model can be divided into two parts, the load and the configuration model. The configuration model gives the system component level structure while the load model gives a typical load (operational) profile. The load profile has a great deal to do with the quality of the system results and it is possible to have different load models for a system to differentiate the system results. There are various tools available that support the load modeling process [Calzarossa 1991].

The design of the configuration can be either hierarchical (vertically organized) or structural (horizontally organized, a so-called flat model) [Malhotra 1993]. It is possible, in the case of hierarchical modeling, to step-by-step aggregate and disaggregate various parts of the model and to use different description formalisms and solution techniques for different aggregates. At each aggregate level the results serve as input for the next higher level aggregate. In the case of the structural model, the modules and components are all in one layer, no aggregates are used, and the configuration can only be dismantled into its basic components.

2.3. Computational modeling

The abstract model being too complex for immediate application of mathematical techniques is transformed into a computational model. Depending on the questions that may be studied, different description techniques are used:

- Queuing network models for performance analysis [Lazowska 1984; Bolch 1989]
- Stochastic activity networks for performance analysis [Sanders 1991; Sanders 1995]
- Stochastic process algebras for performance analysis [Horton 1994; Siegle 1995]

[3] Standard reliability measures and input data can also be obtained from the MIL-Handbook or F. D. Power.

- Precedence graph models for performance analysis [Fleischmann 1989]
- Markov and Markov reward models for performance and performability analysis [Schweitzer 1991; Trivedi 1993; Stewart 1994; Sahner 1996]
- Generalized stochastic Petri nets for performance and performability analysis [Molloy 1982; Ciardo 1989; Ciardo 1994; Muppala 1994]
- Reliability graphs, block diagrams and fault trees for reliability and availability analysis [Sahner 1986; Reibman 1991; Sahner 1996]

Using a particular description language does not usually mean a particular solution method is applicable (and visa versa). The step from the system "abstract" model to the computational model gives a simpler model (or representation form) that is more readily and efficiently analyzed (i.e., solved).

2.4. Model solution

There are two basic methods used to solve the system model: mathematical and system simulation. The mathematical solution method may be further classified into analytical (non-state-space-based) and numerical (state-space-based). The mathematical method works by solving a system (or set) of linear or differential equations while a simulation is differentiated into discrete event simulation and continuous simulation. Combining mathematical techniques with simulation is called hybrid simulation and when possible, independent components are solved separately using either technique and is then combined in a stepwise fashion (aggregation/ disaggregation). One very well known problem when using the numerical method is the high computational cost due to huge state space. Taking a pragmatic view, there are two ways to cope with this. First we may somehow tolerate or modify the large state spaces, or second, we may prevent the origination of such large state spaces. Depending on what type of analysis is desired, either transient or steady state, different efficient solution techniques are available [Greiner 1999]. Section 5 gives an example of both transient and steady state analysis types.

2.5. Validation and model refinement

The process of validation demonstrates that the model accurately represents the desired system behavior with enough detail [Allen 1978; Allen 1990]. In general, there are three important steps:

- Verify the model assumptions against the real system (reality check),
- Analyze the model structure and the logical relations among its components,
- Compare the behavior of the model under different experimental conditions (e.g., different load models).

We distinguish between *operational* and *conceptual* validation because, as shown in Figure 2a and 2b, there may not be a way to parameterize the model (i.e., extracting system measures from an existing system or its analog). In the <u>conceptual</u> <u>validation</u> phase the computational model is compared to the system model (i.e., its more detailed and less abstract predecessor). This comparison must be done to test the assumptions of the higher level abstraction to determine if the computational model's assumptions are correct or at least reasonable. In fact, this may be a process of adjusting things down to a tolerable level of simplification. Conversely, during <u>operational</u> <u>validation</u>, the computed performance results (i.e., predictions and/or estimates) are compared to the system measures. This is a very important step because it determines how the system model (and input parameters) may be modified to more

closely describe the actual system behavior. This process of refinement gives a computational model that more accurately predicts the real system behavior under different (or new) conditions.

The validation phase will result in a validated final model that may have been modified with respect to its structure and/or its parameterization (i.e., relationships among variables, initialization and initial state etc.). For example, modifications to the model can be carried out with the goal of predicting the behavior for the system under study (SUS). When used in this sense, parts of the model are removed or changed in an effort to investigate the cause and effects of proposed enhancements or adaptations. Furthermore, once a model is validated it may be used to perform sensitivity analysis which can be used to support or discredit the modeling assumptions and analysis conclusion(s) [Choi 1993]. The two most common forms of sensitivity analysis include:

- Testing the robustness of the computational results against the model assumptions. This requires that the model be analyzed some number of times to allow comparing the results from one run to the next.
- Obtaining bounds on the expected performance measures by evaluating the model under worst and/or best case assumptions.

In effect, the modeling cycle yields many insights into the SUS. These insights result from the different steps in the modeling cycle and are used to improve the system in some desired aspect. Thus, given a formal model, its external constraints we can explore what mechanisms are available to optimize the system behavior. For example, consider such factors as the SUS topology, fault tolerance, timeliness, resource allocation, communications etc. How do such mechanisms impact the behavioral aspects such as reliability and performability? Refining the system model can reveal trade-offs in design alternatives such as deciding what features of the system should be changed to improve the system's reliability or validating certain assumptions with respect to various performance goals.

3. Modeling techniques

As discussed, different types of models are available to obtain the computational model from the abstract model. Generally, the choice of model type depends on the answers we wish to obtain from the model. A short list of available techniques was introduced earlier. Lets now explain these description techniques in more detail.

3.1 Reliability and availability models

These models are used to predict the reliability and, depending on the structure, the presence of failure(s) and repair of component(s). When considering the measures of both, these terms are often denoted *dependability* [Muppala 1994]. There are three model types commonly used for reliability and availability analysis: reliability block diagrams, reliability graphs and fault trees. Before explaining each of these model types, lets first define what we mean by reliability and availability:

- The reliability R(t) of a component is the ability of the component to work correctly over a period of time t. If S denotes the system, then R(t) is given by: R(t) = P(S is working correctly in [0,t]). Thus, if X denotes a random variable representing the time to failure, then the system reliability at time t is:

  $$R(t) = P(X > t).$$

  This assumes the system cannot work indefinitely with out failure (i.e., $\lim_{t \to \infty} R(t) = 0$) and the system is operational at time t=0 (i.e., R(0)=1).

- The availability A(t) is defined as the probability that the system is operational at time t, regardless of the number of failures that have occurred up to now: A(t) = P(S is operational at time t). The availability of the system at t=0 is assumed to be 1, and there exists a steady state availability (Figure 3 shows the typical behavior of an availability function):

$$\lim_{t \to \infty} = A.$$

It is important to note that if the system is not repairable, the definition of R(t) and A(t) are equivalent. Given these definitions, lets explain the concept and semantics of the descriptive models which give the important relationship(s) among components that are assessed to determine the measure (i.e., a prediction) of interest.

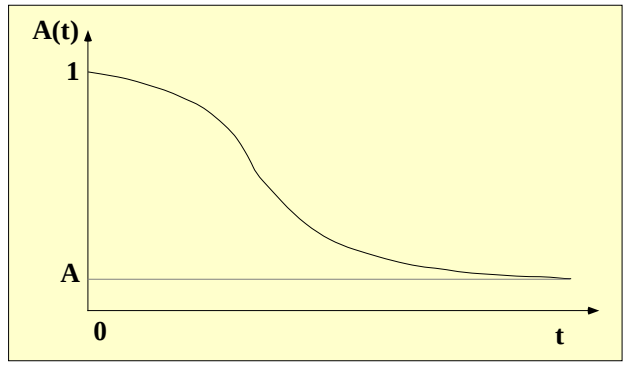


Figure 3. Typical availability function behavior.

3.1.1 Reliability block diagrams

Reliability block diagrams describe the logical structure of the system with regard to how the reliability of individual components affects the overall system reliability. A block in the diagram represents each component. The arcs represent the logical structure between

$$F(t) = \begin{cases} 1 - \prod_{i=1}^{N} (1 - F_i(t)) & \textit{for a series structure,} \\ \prod_{i=1}^{N} F_i(t) & \textit{for a parallel structure.} \end{cases}$$

these components. If we have for example, a set of components in series, the system will run if every component is operational. A parallel structure runs if at least one of the parallel components is operational. The third possibility is to have components connected in a k out of n structure. These basic structures can be combined with each in any desired way. In a system with N components the distribution function for the failure time is:

Consider the model of a bridge used in interconnection networks as shown in Figure 4a. The bridge consists of the five components C_1, C_2, C_3, C_4 and C_5. The system is operational as long as there is one of the following paths available: C_1C_2, C_4C_5, $C_1C_3C_5$ and $C_4C_3C_2$.

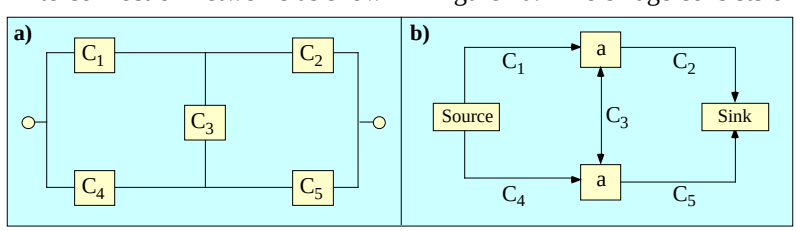


Figure 4. Reliability (a) block diagram, (b) graph of a bridge.

3.1.2 Reliability graphs

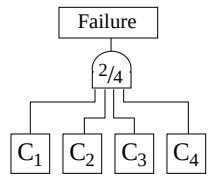
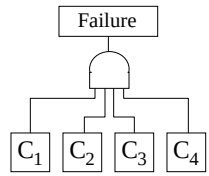
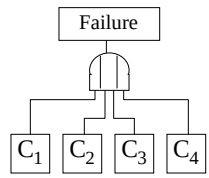
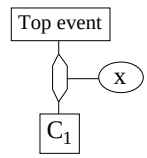
A reliability graph is an ordered graph consisting of edges and nodes. In contrast to many other ordered graphs, the structural relationships between the components that can fail are not described by the nodes but by the edges. The source node contains no incoming edges while the sink node contains no outgoing edges. For the purpose of analysis the edges are assigned failure rates, failure probabilities or unavailability probabilities. If there is no path in the graph from source to sink, the system has failed. The distribution function is the same as for a reliability block diagram. The equivalent reliability graph for the bridge example is shown in Figure 4b. In this case, the system will

fail if the edges labeled C_1,C_2 and C_4,C_5 both fail. Note that as long as a path, for example C_4, C_3, C_2 or C_1, C_3, C_5 exists from source to sink, the system is operational.

3.1.3 Fault trees

Fault trees have their roots in the hardware world (no pun intended) but they are equally applicable to software and especially when considered together as a system. A fault tree can be considered as an elaborated depiction of an undesired event (i.e., failure) to the system and is usually critical from a cost or safety standpoint. This event constitutes the top level happening in the tree and generally consists of the complete or catastrophic failure. It is very important to choose this event carefully. In the case that the top event is too general, further development of and solution of the fault tree may become unmanageable. On the other side, if it's too specific, the analysis may not provide a sufficiently broad view of the system.

Table 2. Fault tree extensions.

Symbol	Name	Meaning
	k out of n combina-tion gate (here 2/4)	The top event appears if k out of n input events return a logic 1
	AND gate	The top event appears if all input events return a logic 1
	OR gate	The top event appears if at least one input gate returns a logic 1
	IF gate	The top event appears if the input event occurs and condition x is fulfilled

Failures happen when an (undesirable) event (or chain of events) occur(s). Multiple events may occur in parallel or sequentially. These undesirable events may happen because a fault (or defect) has been encountered (or invoked). They may be associated with hardware or software and/or some anomaly or user error such as an unexpected environmental occurrence or input. A fault tree thus depicts the logical interrelationships of basic events that lead to the undesired top event. The tree is tailored from the top event down to all of the basic events that could possibly lead to causing the top event to occur. The tree cannot cover all possible faults, but only those that may lead to the top. Moreover, these faults are not complete in the sense that the tree covers all possible events that lead to the top.

The structure of the tree can be considered as a complex of articles known as gates which permit or inhibit the passage of the fault logic up the tree (logic 1 at the node if the event occurs, otherwise 0). The input to a gate is either a basic event or the output of another gate. Basic events are the leaves of the tree (that have no inputs) and the top event is the root (has no output). Furthermore, we assume the basic events are mutually independent with known probability (or rate) of occurrence. Apart from the basic AND <u>and</u> OR gates many other gate types have been introduced [Villemeur 1992]. The most important extensions are shown in Table 2. If we assume the fault tree does not contain repeated events and the number of inputs at a gate is equal to n, the distribution function for the failure time of the system is:

$$F_{FaultTree}(t) = \begin{cases} 1 - \prod_{i=1}^{n} (1 - F_i(t)) & \textit{OR gate}, \\ \prod_{i=1}^{n} F_i(t) & \textit{AND gate}, \\ \sum_{i=k}^{n} \binom{n}{i} F(t)^i (1 - F(t))^{n-i} & \textit{k out of n gate} \end{cases}$$

3.2 Performance and functional models

Performance models enable us to represent the probabilistic nature of the work the system is exposed to (i.e., subject to operational conditions). Such methods permit us to predict the ability of the system to carry out the intended function (work) under the assumption that no components fail.

3.2.1 Stochastic process algebras

A process algebra (PA) is an abstract description language that provides a general mathematical model for representing real systems by the terms or expressions of the model, and manipulating these terms in order to analyze the behavior of the systems [Milner 1989]. Thus, one of the main advantages provided by PAs are their composability. PAs have been used to specify and design computer system. The basic idea is that these systems can be readily decomposed into subsystems, which operate concurrently and interact with each other as well as their common environment [Hoare 1985]. The process behavior is described within an algebra, which provides a set of operations to structure and refine such systems. It avoids many of the traditional problems associated with parallelism in programming (i.e., interference, mutual exclusion, interrupts, multithreading, semaphores, etc.) and it includes many advanced structuring ideas (e.g., monitor class, module, package, critical region). PAs give a secure mathematical foundation for avoidance of errors such as divergence, deadlock and non-termination and for achievement of provable correctness in the design and implementation of computer systems. Thus, PAs permit us to build structured models, meaning that a model can be built using the components of a finer model. It is from the characteristics of the finer model that the characteristics of the coarser model (or the overall model) are derived. Process descriptions, based on these algebraic theories, are for example CSP (communicating sequential processes) and CCS (calculus of communication).

To predict the performance of a system specified using a PA some extensions were introduced and are thus called stochastic process algebras (SPA). In a SPA, process activities are assigned stochastically distributed times. This extension enables us to examine the order of events as well as their performance (timeliness). Most of the approaches are limited to exponentially distributed times which permit them to be mapped to a continuous time Markov chain (CTMC). The great advantage to a SPA is its formalized underpinnings as a description language and the availability of a very powerful calculus. The calculus can be used to simplify the model or to prove the equivalence between processes [Siegle 1995].

An example of a SPA is TIPP (TImed Processes and Performance evaluation) which is one of a good number of Timed Process Algebra (e.g., Algebra of Timed Processes [ATP], CCS Shared Resources [CCSR] based on CCS, Timed CSP [TCSP], U-LOTOS or Urgent LOTOS, etc.). When specifying a system, each process term is assigned

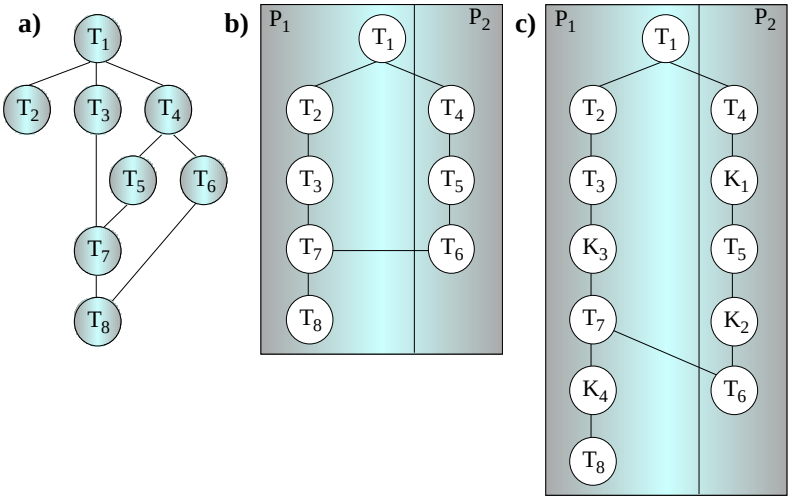


Figure 5. Precedence graph (a) task partitioning, (b) processor allocation, and (c) overall model.

a transition system. Let ACT denote the set of action names and PDF a set of probability density functions. The core of the description language is then given by [Rettelbach 1996]:

- a ACT,
- F_a PDF,
- S proper subset ACT, and
- A set of process variables VAR with X VAR.

The syntax of TIPP can then be given by:

$$\mathbf{P := 0 \mid X \mid (a, F_a).P \mid P + Q \mid P \|_s Q \mid recX : P}$$

- 0: is the empty process that can perform no actions,
- $(a, F_a).P$: the process performs action a and behaves then like process P. The time to perform the action is given by the function F_a
- The process P + Q behaves like P or Q depending on which partial process performs its action first,
- $P \|_s Q$: the partial process P and Q work concurrently; only actions in the set s are executed together, and
- recX: P describes a recursive term with infinite behavior.

TIPP has been used to model multiprocessor systems by mapping them onto a load and a configuration model. The TIPP project focuses on a basic framework that supports functional specification as well as performance evaluation in a single process algebraic formalism.

3.2.2 Precedence graphs

Another description technique for performance modeling is precedence graph models [Fleischmann 1989]. Precedence graphs are also used to model parallel algorithms as shown in Figure 5. The structure is represented by an acyclic graph AG = (T, E) where T denotes the set of nodes and E represents the set of edges:

- The nodes correspond to the different tasks, and
- The edges (t_i, T_j) E represent the dependencies between the tasks. A task T_i can be processed as soon as all its predecessor tasks have been processed.

A precedence graph is designed in three basic steps. In the first step, the partition of the parallel algorithm into tasks is decided (Figure 5a). Next, the tasks are assigned to a processor (P_1 or P_2 in Figure 5b). In this example, the tasks T_1, T_2, T_3, T_7 and T_8 are assigned to processor P_1 and T_4, T_5 and T_6 are assigned to P_2. In the last step, the communication structure is considered (Figure 5c). By assigning a processing time (or rate) to each of the nodes, the overall processing time of the parallel structure may be calculated. Precedence graph models with conjoint (or shared) connections of the tasks have the disadvantage that only static process assignments can be modeled and analyzed, while for dynamic strategies (processor assignment and communication) the so called event traces may be considered and analyzed [Fleischmann 1989].

3.2.3 Markov models[4]

Real life models are usually very complex where many components interact with each other. The models presented so far are limited to the extent that it is either impossible or very difficult to capture all of the complex interactions. Markov models may well be the best choice when it comes to a great deal of complexity [Trivedi 1982; Reibman 1989; Stewart 1994]. Hence, we discuss Markov chains, which are a very important and special case of stochastic processes.

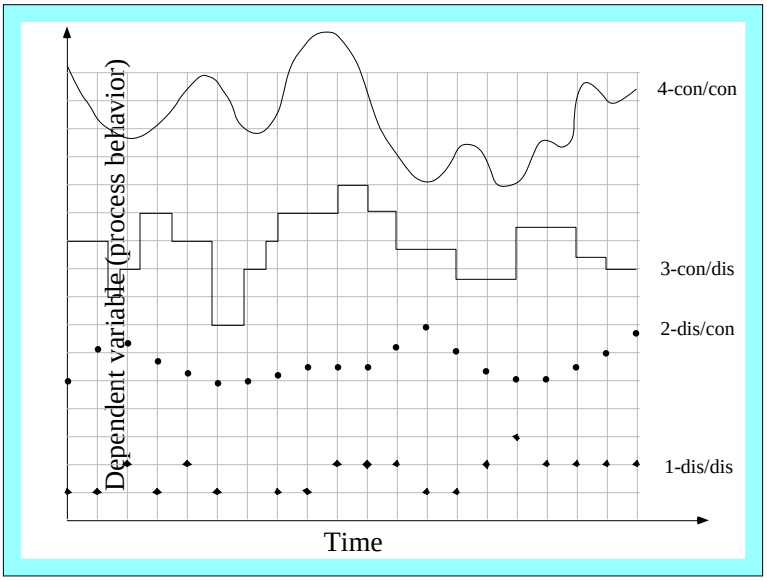


Figure 6. Possible types of stochastic processes.

Consider the set X = (X_t; t T) of random variables defined on the probability space (Ω, A, P) with X_t : Ω → S. The set X is called a stochastic process and is denoted X(t) in the following. The elements x S are called the states and S is called the state space of the stochastic process. The sample space t as well as the state space S of the stochastic process X(t) can be either discrete or continuous. This results in four different types of stochastic processes as shown in Figure 6.

In Figure 6, possibility 1 and 3 are called chains to show clearly the discreteness of the state space and to distinguish them from their continuous counterpart, possibility 2 and 4. If the future behavior of the stochastic process depends only on the current state then we may call the stochastic process a Markov process. Otherwise, the stochastic process is called a non-Markov process (Semi-Markov process). Depending on the state space of a Markov process, we can differentiate:

- Discrete state Markov processes
- Discrete time: DTMC (possibility 1, in Figure 6), and
- Continuous time, CTMC (possibility 3, in Figure 6).
- Continuous state Markov processes (possibility 2 and 4, Figure 6).

The following discussion considers only discrete state Markov processes (i.e., DTMC Sections 3.2.3.1-3), while CTMCs are discussed in Sections 3.2.3.4-6.

3.2.3.1. Discrete Time Markov Chains (DTMC)

The state of the process for DTMCs is observed at a discrete set of times and the future behavior of the process depends only on the current state. This important property is called the Markov property (or memoryless property). The discrete parameter space T is represented by the set of natural numbers {0, 1, 2, …}. Successive observations define the random variables X_0, X_1, …, X_n, … at time steps 0, 1, 2, 3, … n, … Thus, this important property can be satisfied:

[4] The material presented here is of fundamental importance to stochastic analysis and it distinguishes the theory from which most of the modeling techniques presented here are based.

$$P\{X_{n+1} = x_{n+1} \mid X_0 = x_0, X_1 = x_1, \ldots, X_n = x_n\}$$

$$= P\{X_{n+1} = x_{n+1} \mid X_n = x_n\}.$$

The conditional probabilities here are known as the single step transition probabilities of the Markov chain. They give the conditional probability of making a transition from state x_n to state x_{n+1} when the time parameter increases from n to n+1. They are denoted by $p^{(n)}_{ij} = P\{X_{n+1} = j \mid X_n = i\}$. In a homogeneous DTMC these probabilities are independent of n and can be written as $p_{ij} = P\{X_{n+1} = j \mid X_n = i\}$, for all n = 0, 1, … The transition probability matrix P is formed by placing all possible p_{ij} in row i and column j, for all i and j. The elements of the P matrix satisfy the following two properties: $0 \le p_{ij} \le 1$, and for all i,

$$\sum_j p_{ij} = 1$$

When the Markov chain is nonhomogeneous, the elements p_{ij} are replaced with $p_{ij}(n)$ and the matrix P with P(n). The Chapman-Kolmogorov equations give us the method for determining the n-step transition probabilities. If we generalize the single step transition probability matrix to one that is n-step, we get a matrix whose elements are $p^{(n)}_{ij} = P\{X_{m+n} = j \mid X_m = i\}$. These elements may be obtained from the single step transition probabilities.

$$P(n)(m,m+1, \ldots, m+n) = P(m)P(m+1)\cdots P(m+n).$$

For a homogenous DTMC we may write

$$p_{ij}^{(n)} = P\{X_{m+n} = j \mid X_m = i\}, \qquad {}_m = 0, 1, 2, \ldots$$

Thus, we can see that $p_{ij} = p_{ij}^{(1)}$. Using the Markov property, the following recursive formula may be used for calculating the $p^{(n)}_{ij}$:

$$p_{ij}^{(n)} = \sum_k p_{ik}^{(h)} p_{kj}^{(n-h)}, \quad o < h < n.$$

In matrix notation, the Chapman-Kolmogorov equations are written as,

$$P^{(n)} = P^{(h)}P^{(n-h)}.$$

This association states that it is possible to write an n-step homogeneous transition probability as the sum of products of an h-step and (n-h) step transition probabilities. To go from i to j in n steps, it is necessary to go from i to an *intermediate* state k in h steps, and then from k to j in the remaining n-h steps. By summing over all possible intermediate states k, we consider all possible distinct paths leading from i to j in n steps:

$$P^{(n)} = PP^{(n-1)} = P^{(n-1)}P.$$

From this we can see that the matrix of n-step transition probabilities is obtained by multiplying the matrix of one-step transition probabilities by itself (n-1) times (i.e., $P^{(n)} = P^n$).

3.2.3.2. Attributes of DTMC

The states in a DTMC may be characterized based on their long run behavior. A state is transient if during an infinite interval of time the state is entered only a finite number of times. Otherwise, it is called recurrent. For

recurrent states, the mean recurrence time can be used to differentiate the states into positive recurrent (mean recurrence time is finite) and null recurrent (mean recurrence time is infinite). Null recurrent states only exist in infinite state spaces. An absorbing state is said to be absorbing if once entered, it is never left (i.e., only incoming arcs and no outgoing arcs). State k is called periodic if there is an integer m > 1 with p^m_{kk} =0. If such an m exists, then it is called the period of the state k, otherwise that state is said to be aperiodic.

3.2.3.3. Stationary distribution of a DTMC

If we run a number of DTMCs at the same time, then u^t_k denotes the percentage of DTMCs that are in state k at time t. Once the DTMCs have run long enough, we find u^t u^{t+1}, or in other words $\lim_{t \to \infty} u_t = u$. Thus, the long run behavior of the DTMC is not dependent on the initial "starting" state any more. When considering only one DTMC, the stochastic vector u is called the stationary distribution which means that at *any* time t, the DTMC will be in state k with probability u_k. Stationary distributions can be used to compute steady state performance measures such as node utilization. There are DTMCs with none, one or several stationary distributions. A unique stationary distribution exists only for an ergodic DTMC. A DTMC is ergodic if it is irreducible and all states are positive recurrent and aperiodic. Irreducible means that if every state can be reached from every other state (i.e., if there exists an integer m for which $p^{(m)}_{ij} > 0$ for every pair of states i and j). Let S be the set of all states in a Markov chain, and let S_1 and S_2 be two subsets of states that partition S. S_1 is said to be closed if no one-step transition is possible from any state S_1 to any state in S_2. In general, any nonempty subset S_1 of S is said to be closed if no state in S_1 leads to any state outside S_1 in any number of steps (i.e., $p^{(n)}_{ij} = 0$, for i S_1, j S_1, n ≥ 1) [Stewart 1994]. If S_1 consists of a single state, then that state is called absorbing state (i.e., $p_{ii} = 1$, an absorbing state can be considered to be a failed state). If the set of all states is closed and does not contain any proper subset that is closed, then the Markov chain is known to be irreducible. On the other hand, if S contains proper subsets that are closed, the chain is known as reducible.

3.2.3.4. Continuous Time Markov Chains (CTMC)

If the states of a Markov process are *discrete* and state transitions may occur at *any* point in time, we say the process is a CTMC. Let the random variable X = {X(t), t ≥ 0} denote the stochastic process with discrete states space S. This stochastic process forms a CTMC if for all integers n, and for any sequence $t_0, t_1, \ldots, t_n, t_{n+1}$ such that $0 \le t_0 < t_1 < \ldots < t_n < t_{n+1}$. Thus, for any set of time points t_i, random variables X and states x_i the following relationship holds:

$$P\{X(t_{n+1}) = x_{n+1} \mid X(t_0) = x_0, X(t_1) = x_1, \ldots, X(t_n) = x_n\}$$

$$= P\{X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n\}.$$

$$= p_{x0x1}(t_0,t_1) \cdot p_{x1x2}(t_1,t_2) \ldots p_{xn-1xn}(t_{n-1},t_n).$$

This last equation corresponds to the probability of a path with fixed initial starting state, and the prior equation denotes the conditional probability of moving from state x_n to x_{n+1} in the interval $[t_n, t_{n+1}]$ under the condition that we are in state x_n at time t_n.

3.2.3.5. Stationary distribution of a CTMC

The notation of a stationary distribution for a DTMC was introduced above. In the same way a stationary distribution for CTMCs is defined. The CTMC is stationary (or has a stationary distribution) if the vector $\pi(t)$ reaches a statistical balance. This is exactly the case if there exists a time t_0 with $\pi(t) = \pi \quad t \geq t_0$. In this case the state probabilities are constant ($\pi'(t) = 0 \quad t \geq t0$) and simplifies to:

$$0 = \sum_{j \neq k} \pi_j q_{jk} - \sum_{j \neq k} \pi_k q_{kj}$$

$$0 = \pi \mathbf{Q}.$$

Instead of a differential system of equations we now get a system of linear equations to solve (various solution methods are available). In addition, the solution now depends on the condition that the sum of all state probabilities is equal to one. Together, with this condition we get the solution vector π, which is called the stationary distribution. This solution fulfills the steady state property $\pi = \pi P \quad t > 0$. Once a CTMC reaches steady state, it will remain there for all time points $t \geq t_0$, which follows from $\pi(t) = \pi(0)\mathbf{P}(t) = \pi(0)\mathbf{P}(t_0)\mathbf{P}(t-t_0) = \pi(t_0)\mathbf{P}(t-t_0) = \pi\mathbf{P}(t-t_0) = \pi$. Up until the time t_0, the CTMC is said to be in a transient status. The similar transient distribution is now explained.

3.2.3.6. Transient distribution of a CTMC

Sometimes it's important to understand how performance measures evolve over time. This is accomplished by computing the $\pi(t)$ vector of state probabilities repeatedly over the desired interval of time, known as transient analysis. To perform this analysis, the CTMC is not permitted to be acyclic. Acyclic means that once a state has been visited [entered] it will never again be visited. To determine the state probabilities $\pi(t)$ the Kolmogorov system of differential equations must be solved: $\mathbf{P}'(t) = \mathbf{P}(t)\cdot\mathbf{Q}$, (where **P** is the CTMC stochastic matrix which gives the various state to state transition rates and **Q** is known as the generator matrix (or infinitesimal generator matrix **Q**). Once $\mathbf{P}(t)$ is determined the solution vector $\pi(t)$ is given by solving:

$$\pi(t) = \pi(0)\mathbf{P}(t).$$

In addition, we can determine the mean time a Markov process spends in state **i** during the interval [0, t]. This measure is given by the following integral:

$$L_i(t) = \int_0^t \pi_i(x)dx$$

In practice, we often find systems that are not repairable (i.e., once it fails, it will never recover). For a CTMC this means once this state is reached, there is no way out. Such states are called absorbing, and an important measure is, for example, the <u>m</u>ean <u>t</u>ime <u>t</u>o <u>a</u>bsorption (MTTA). Let $\lim_{t \to \infty} L_i(t) = \pi_i$ denote the mean time that the Markov process spends in the state i before entering an absorbing state. To determine τ_i we can partition the state space S of the CTMC into absorbing states S_A and transient states S_T. By reordering the generator matrix **Q**, we get a sub-matrix $\mathbf{Q}_T$ that contains only the transient states. By taking the limits $t \rightarrow$ infinity we obtain a linear system of equations for determining the vector τ:

$$0 = \tau \mathbf{Q}_T + \pi_T(0).$$

By using the vector τ, the following simple expression for the mean time to absorption can be given:

$$MTTA = \sum_{i \in S_T}^{\infty} \tau_i$$

Along our way, we have always assumed there exits methods to determine a solution of the state vector $\pi(t)$. See [Greiner 1999] for the transient and steady state solution of DTMCs and CTMCs.

3.2.4. Queuing network models

Queuing network (QN) models are a well-known modeling paradigm. This is primarily due to their compactness and simplicity. A QN consists of at least two service stations that are connected to each other. Sources in the real system (e.g., machines, CPUs, printers, airplanes...) are mapped to a node in the QN. The node itself can be further divided into the service unit and the queue that stores customers (i.e., jobs). For the system to work we need jobs circulating in the system. Once a job is served at one node it moves to the next node or returns to the same node.

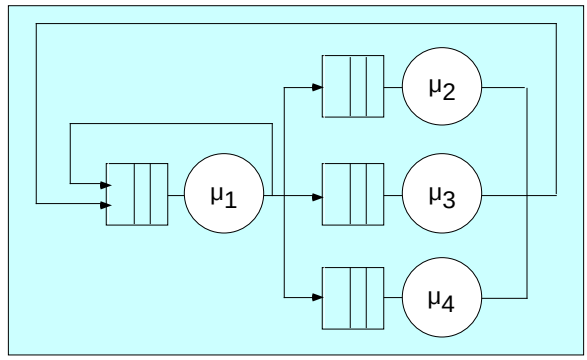


Figure 7. The central server queuing network model.

There are several different types of QNs, open, closed and mixed networks. Open QNs are characterized by the fact that jobs arrive from the outside at the network and leave it once they have received their service. This means especially, that at any time the number of jobs is not constant. The opposite of open QNs are closed QNs where the number of jobs in the system is always constant (i.e., once a job leaves the system another new job immediately replaces it). The third class is mixed QNs where some jobs belong to the open job class and some jobs belong to the closed job class. A simple example of a QN is shown in Figure 7. This is the very well known central server model proposed by [Buzen 1971]. Node 1 represents the CPU that processes jobs with rate μ_1. The other nodes model peripheral devices such as disk, printer, or backup devices etc.

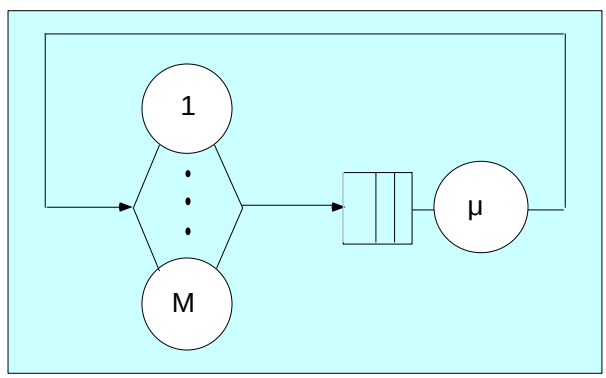


Figure 8. The machine repairman queuing model.

Another well-known QN is the machine-repairman model, shown in Figure 8. The strength of this model comes from the fact that it can be used to model many different types of systems. If, for example, the M machines correspond to terminals and the repairman represents the computers, then we have the model of a terminal system.

In Kendall's notation, a node in a QN can have many different characteristics as for example the node type, the service strategy or type of service processes, etc. Kendall's notation is used to describe elementary queuing systems [Kleinrock 1975; Bolch 1989]: **A/B/m/C** – queuing discipline (CDF is cumulative distribution function),

where A denotes the CDF of the arrival process: $\mu = \frac{1}{\bar{T}_S}$,

and B denotes the CDF of the service process: $\lambda = \frac{1}{\bar{T}_A}$.

Since the arrival and service processes are stochastic, their CDF has a great influence on the system behavior and the mathematical tractability. From a mathematical standpoint, exponentially distributed service times can be handled best. For A and B the abbreviations shown in Table 3 are used.

Furthermore, m denotes the number of servers (m ≥ 1), and C denotes the capacity of the node (number of jobs in the queue + number of jobs in the server). Usually, when C is not specified, an infinite capacity is assumed. Finally, the queuing discipline (also called the service strategy) determines the policy for picking jobs from the queue. The most important disciplines include the ones listed in Table 4. A simple example of Kendalls notation is for example the expression:

M / G / 1 / 5 – RR

Which means that we have an exponentially distributed arrival process, general distributed service process, one server, a finite queue with capacity five, and the round robin service strategy (respectively).

Table 3. Abbreviations for CDFs used in Kendall's notation.

Abbr.	Cumulative Distribution Function
M	Exponential distribution
E_k	Erlang distribution with k places
H_k	Hyperexponential distribution with k phases
C_k	Cox distribution with k phases
G_k	General distribution
GI	General independent distribution
D	Deterministic distribution

Table 4. Important queuing disciplines.

Abbr.	Description of Strategy
FCFS	First-Come-First-Served: jobs are served in the order they arrive.
LCFS	Last-Come-First-Served: the last job in the queue is served first.
SIRO	Service-In-Random-Order: the next job to be served is chosen in random order.
RR	Round Robin: the system uses a periodic time sliced method. Unfinished jobs may be pre-empted and placed at the end of the queue but will eventually finish.
PS	Processor-Sharing:
Priorities	Job selection depends on its priority, which may be assigned to a job's remaining processing time or to the importance of a job. Priorities may be permanent or change while time passes (dynamic priorities).
Preemption	If the queuing discipline is LCFS / priority, the current job may be preempted. If a higher priority job arrives and the current job is preempted then a preemptive resume strategy is defined. Otherwise, if the job is allowed to finish before the higher priority job is given access it's called a Head Of Line strategy.

3.2.4.1. Local balance – Global balance

As we have discussed for Markov chains, the behavior of a system can be described using CTMCs. The steady state probabilities π_i are computed from $\pi\mathbf{Q}=0$. This equation says that for each queuing network in steady state, the flux into a state is equal to the flux out of that state and is called conservation of flow, as shown here:

$$\sum_{j \in S} \pi_j \, q_{ji} = \sum_{i \in S} q_{ij}.$$

This is the global balance equation (GBE). Since the number of equations can be extremely large, an alternate technique is needed. The GBE can be split into local balance equations to describe system behavior. But this is possible only if the network can fulfil certain characteristics concerning inter-arrival, service times, and queuing disciplines. The local balance equations are much easier to solve. Queuing networks that have a unique solution of the local balance equations are called product form networks (the departure and arrival rates to a state i are equal). Not every network can be solved using the local balance equations. However, there always exists a solution of the GBE. Therefore, global balance can be considered as a sufficient, but not necessary condition for local balance. Networks with a solution to the local balance equations are said to have the local balance property (which, by the way is a unique solution to the GBE). If every node in the network fulfil the local balance property then: (1) the network has a product form solution and (2) the overall network has the local balance property. The importance of these results are derived from the fact that whenever the overall network has the local balance property, the separate nodes behave as if they were single queuing systems and can therefore be considered in isolation.

3.2.4.2. Product form solutions

The term product-form was introduced by [Jackson 1963; Gordon 1967] which considered open and closed QNs with exponentially distributed arrival and service rates. Three results, namely the theorem of Jackson, the theorem of Gordon/Newell and the theorem of BCMP are very important because they enable the use of analytical techniques for the solution of product form queuing networks.

3.3. Performability models

Performability is a fabricated word that combines the two terms performance and reliability. The performability discipline tries to merge these two modeling paradigms. The systems under consideration are so-called degradable systems, meaning the system may be able to survive the failure of one or more system components. Once a system component fails, the system may continue to operate with a reduced performance. In such cases, it is necessary to consider both performance and reliability together. A common technique for modeling degradable systems is the Markov reward model (MRM). There are other techniques such as Semi-Markov reward models [Ciardo 1992b], Markov regenerative reward models [Logothetis 1997] and the monolithic Markov model [Greiner 1999]. The monolithic method combines the structural oriented performance model with the failure oriented reliability model. This model presents a very high solution overhead compared to the MRM.

3.3.1. Markov reward models

A MRM is a homogeneous Markov process together with a reward function r(i) which assigns each state i a reward r(i). Let X_t, $t \geq 0$ denote a Markov process with finite state space S. This Markov process is completely described by the generator matrix Q and the initial probability vector $\pi(0)$. If we assign the reward function $r : S \rightarrow$ to this Markov process, we get what is called an MRM. In general, there are four types of reward functions:

- Assume the Markov process X is in state i at time t_i. It then obtains the reward $t_i r(i)$ per unit of time. The reward r(i) can therefore be considered as the performance level of the system while being in state i. A reward can be positive or negative. Negative rewards connote a loss instead of a gain. Since the system produces work at a rate r(i) during the time it occupies state i, we call this a rate based reward model.
- The rate based reward model is extended so that the reward also depends on the time t and we obtain r(i,t).

- A third class of rewards are the impulse-based rewards r(i,j) which are associated with each transition from state i to state j. Whenever such a transition occurs, the cumulative system reward is increased by r(i,j).
- Impulse-based reward models can also be time dependent, which leads to r(i,j,t).

These reward models can even be mixed within the same model. Due to the general definition of the reward, MRM are a very powerful extension of Markov models. Let us review some important reward measures.

3.3.2. Certain important Markov reward measures

A MRM can yield three types of steady state as well as transient performance measures. Expected value of the reward rate, accumulated reward and distribution of the instantaneous reward. Usually we are interested in the overall reward picked up in an interval [0,t]. This reward measure is called the accumulated reward and tells us for example how much money we have lost or gained in the interval.

3.3.3. A simple Markov reward example

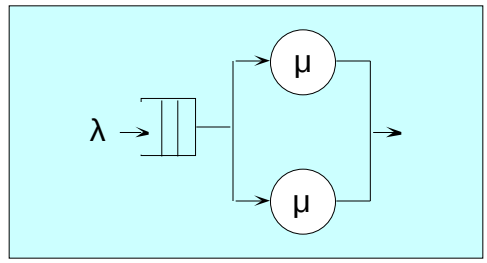


Figure 9. A simple M/M/2 system

The complexity of computing performability measures using a monolithic system model (Figure 10a) compared to using the MRM (Figure 10b) is much greater. To illustrate this, lets consider the queuing network (M/M/2/5 system) shown in Figure 9. Since we assume a finite buffer with capacity 5, the corresponding CTMC has exactly six states (i.e., the nodes denote the number of jobs in the system). Lets assume that the M/M/2 server fails with a rate *f* and is repaired with rate r. Jobs arrive at the system with rate λ and are processes with rate μ. The corresponding monolithic system model consists of three layers, the top layer (level 1), denoting the case when both servers are operational, the middle layer (level 2) denoting the case when one of the two servers fail, and the bottom layer (level 1) denoting that both servers have failed. In level 3 only jobs can arrive because both available servers are failed. Switching between layers take place with rate f (failure) and r (repair). To understand the throughput per unit time t_p we'll need to determine the number of customers served per time unit. Since the system has redundancy the failed servers can be repaired, t_p also depends on the performance level. The fact that for performability analysis using a monolithic model all performance levels have to be explicitly represented means there is a great deal more complexity. Imagine what happens if, instead of two servers, we have h servers and a maximum number of n jobs. Instead of n+1 states (in the case of MRM) the monolithic model gives (n+1)(h+1) states. The overall number of states therefore grows exponentially in system complexity and linearly in the number of performance levels. In the MRM model a whole layer is

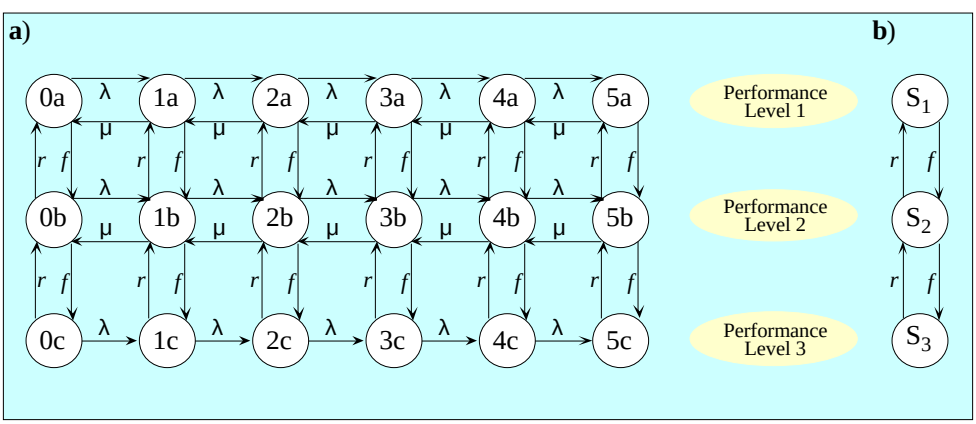


Figure 10. CTMC example a) Monolithic system model of the M/M/2/5 system as compared to b) MRM model of the M/M/2/5 system.

represented by one state and the vertical transition rates between the performance levels in the monolithic model are now the transition rates in the MRM (see Figure 10b).

The state probabilities of the resulting Markov process provide the portion of time spent in performance level i. The probabilities combined with the throughput of performance level i permit us to determine the throughput per time unit t_p. The savings in complexity are readily seen and the MRM modeling can therefore be considered as splitting the problem into a performance model and a reliability model. Actually the splitting of the monolithic model into a performance/ reliability model only approximates the monolithic model's behavior, but this is actually a very good approximation. The rates of the performance and reliability model differ in magnitudes and therefore before switching into another performance layer steady state is achieved.

Its should be clear by now that to describe the Markov reward model (or any Markov model) by hand is a complex and error prone task. It would be better if another high level description technique is needed which offers an easier modeling paradigm but still has the power of Markov models. One possible description technique is stochastic Petri nets, which are explained below.

3.4. Performance and performability models

Generally, Markov and MRM model types can be applied to a wide range of problems. They are powerful modeling techniques, however inherent in the Markov models are two basic problems. First, the appearance of a Markov model is usually far removed from the general feel and shape of the modeled system. Second, the state space grows exponentially in the system complexity and makes it *very* difficult to specify a model correctly. Stochastic Petri nets overcome these difficulties by providing a descriptive formalism that is closer to the modeled system but offers the modeling power of Markov models. There are many tools available for the specification and analysis of SPNs such as SPNP [Ciardo 1994], GreatSPN [Chiola 1985] or UltraSAN [Sanders 1995].

3.4.1. Basics of Petri nets

The PN in its simplest form is a directed bipartite graph, where the two types of nodes are known as *places* (circles) and *transitions* (bars). Places normally represent *events* while transitions represent *actions*. In modeling [Murata 1989], using the concept of conditions and events, places represent conditions, and transitions represent events. A transition has a certain number of input places and output places representing the preconditions and post-conditions of an event. The places are connected to the transitions by input and output arcs.

A transition is enabled if all its inputs contain at least one *token* (depending on the input arc multiplicity). When a transition is enabled, it can fire (asynchronously), leading the Petri net into a different arrangement of tokens. A marking represents a configuration of tokens in the places of the Petri net, and denotes the *state* of the Petri net. A marking is *reachable* if, starting in an *initial marking*, it is obtained by a sequence of firings. The *reachability graph* is the set of all reachable markings connected by arcs representing the transition firings. In a generalized stochastic Petri net (GSPN), each transition has an associated firing time, which can be zero (*immediate shown as dark bars*) or an exponentially distributed random variable (*timed shown as light bars*). Any GSPN with only immediate transitions is automatically just a simple Petri net.

Completion of the action defined by a transition causes a token (one or more depending on the output arc multiplicity) to be assigned to each of its output places. When a place is the input to several transitions, only one of

the transitions is enabled non-deterministically.[5] As transitions are enabled, the state of the Petri net moves from marking to marking. An *inhibitor* arc prevents a transition's firing when its corresponding input place contains tokens (simple Petri nets that include such arcs have been shown to be Turing equivalent).

3.4.2. Generalized stochastic Petri nets (GSPNs)

A Stochastic Petri net (SPN) is simply a Petri net which has been extended in several ways. These extensions embed the model into a stochastic environment by associating a random time with each of the transitions in the net. The most general extensions allow the usage of random variables for times (rates).[6] The underlying stochastic process is captured by the extended reachability graph (ERG), a reachability graph with additional stochastic information on the arcs. The ERG has been shown to be reducible to a CTMC [Marsan 1984] provided that exponential distributions are used for transition firing rates. Since an SPN permits a probability distribution to be associated with arcs (or transitions) they are very suitable for modeling system performance and reliability. Thus, each transition is associated with a random variable that expresses the delay from the enabling to the firing of the transition. When multiple transitions are enabled, the transition with a minimum delay fires first (known as the race model). The transition rate from state M_i to $M_j = q_{ij}$ is given by $q_{ij} = \lambda_{i1} + \lambda_{i2} + \ldots + \lambda_{im}$ where λ_{ik} is the delay in firing a transition t_k which takes the Petri net from marking M_i to M_j (when several transitions enable the firing from M_i to M_j). Another way to resolve such conflicts is the assignment of priorities to transitions. See an especially clear discussion of SPN models in chapter 7 of [Sahner 1996]. Markov and performability models are covered in the same book (chapters 4, and 6 respectively). Examples of these types of models are available in part two (chapters 9, 10 and 12). Also refer to [Levenson 1987; Lewis 1988; Murata 1989; Sahner 1993; Kavi 1994; Balbo 1995; Laprie 1995] for more details on Petri nets and SPNs, as well as Markov processes and Markov Reward processes.

3.4.3. Extensions to GSPNs

Over time, many extensions have been made to PNs and GSPNs. Some of the best known include [Greiner 1999]:

- *Arc multiplicity* is a short hand notation that replaces the need to draw multiple arcs from the same source to destination in such a way that some number of k tokens are either consumed or produced. This extension does not increase the modeling power of the GSPN.
- *Inhibitor arcs* work just the reverse from the normal input arc: they disable a transition in any marking where the number of tokens in a place is equal or bigger than one (or $k \geq 1$). They are just the complement of a regular input arc.
- *Priorities* are enforced by the rule: a transition may fire only if no transition with a higher priority (integer value) is enabled at the same time.
- *Guards* extend the concept of priorities in the sense they allow the specification of predicates about when the transition can become enabled. This extension tends to make the PN easier to understand but does not increase the Petri net's modeling power.

[5]Coincidentally, if several conflicting immediate transitions are enabled in a marking, a firing probability must be defined. If at least one immediate transition is enabled, the marking is said to be a *vanishing* marking (otherwise, if only timed transitions are enabled [or no immediate transitions are enabled] the marking is called *tangible*). If a feasible marking exists where no transitions are enabled the net is deadlocked. The term marking is often used interchangeably with "state." A marking in a Petri net is a configuration of tokens which represent a distinct state.

[6]When there are multiple transitions enabled by one token, a probability is associated with each of the involved transitions. Such a transition is immediate and its firing is instantaneous (no time is consumed).

- *Marking dependent firing rates* allows the firing rate of a marking to be any function of the number of tokens in any place(s) of the Petri net.
- *Marking dependent arc multiplicity* is applied to the problem of moving some number of n tokens from one place to the other. This works quite well when places need to be flushed once a certain event occurs.

3.4.4. Expressing the functional aspects of a PN using a process algebra

Typically, process algebraic laws allow the rewriting of a system description into another, while preserving the notion of correctness that is captured by the equivalence used in the underlying semantic model [Donatelli 1994; Donatelli 1995]. Their inherent support of compositional reasoning enables the construction of complex systems as the combination of conceptually simpler systems [Balbo 1992; Balbo 1993; Balbo 1994; Buchholz 1994]. In this work the dependability of systems is studied to devise techniques to prevent, detect and compensate for anomalies. An experimental tool for translating between the CSP (communicating sequential processes) process algebra and SPNs was developed to explore the specification and analysis of stochastic properties for concurrent systems. Moreover, the idea is to translate the functional system description (using a CSP-based language, P-CSP) into the information needed to predict behavior as a function of observable and stochastic parameters (topology, timeliness, communications and failure categories). The P-CSP language is similar to TIPP presented in paragraph 3.2.1, otherwise refer to [Hoare 1985]. The modeling approach uses a theory based on proven translations between CSP and Petri nets [Olderog 1986; Olderog 1987]. The grammar and CSP-to-Petri net (CSPN) tool enable service and failure rate annotations to be related back to the original CSP specification. The annotations are then incorporated in the next round of translations and stochastic analysis. The tool automates the analysis and iterative refinement of the model. Within this setting, one can investigate whether functional and non-functional requirements have been satisfied [Sheldon 1995; Sheldon 1995a; Sheldon 1996b; Sheldon 1998].

When the models are refined, the reliability and performance requirements can also be refined to reveal trade-offs in design alternatives such as deciding (1) what are the critical system elements; (2) what features of the system should be changed to improve the system's reliability; (3) or validating performance and reliability goals using stochastic system models. To address these issues, critical requirements are abstracted and the system is described using the P-CSP process algebra. Once the model specification has been translated, we enumerate modeling assumptions, estimate model parameters, and solve the model for specific values of the parameters using Markov analysis [Ciardo 1991; Ciardo 1992a; Ciardo 1992b; Ciardo 1992c; Ciardo 1993]. At this point it is easy to introduce timing constraints among feasible markings and to employ any of the numerous tools developed for stochastic Petri net analysis mentioned above[Ciardo 1987; Koller 1997].

4. Solution techniques

The type of model to be chosen for description depends mostly on the performance measures of interest. Yet, these models are only syntactic sugar if no method is provided to solve them. Each type of model has its own specific solution technique and may also be solved using different solution techniques. A product form queuing network may for example be solved by simulation, solving the underlying CTMC or using an analytical method.

4.1. Simulation

Simulation can be considered a very important method for the analysis and planning of complex systems. This development is supported by the availability of computers and since simulation has gained considerable acceptance by industry, it may be considered as one of the key technologies of the 90s [Gangl 1993]. Simulation possesses the ability to map from the real or planned system to a program that can be as detailed as need be. This allows us to analyze, observe and evaluate the dynamic behavior of the system under *many* different conditions. For this reason, simulation is very useful in many different application areas. Simulation may, at times, be the only feasible method for ascertaining the performance measures of interest if we want to avoid building expensive prototypes or running expensive tests. The most important and difficult task when performing a simulation is that of finding and developing a *realistic* model without being too detailed or too abstract.

4.1.1. Classification of simulation methods

Because a simulation model allows us to analyze the dynamic behavior of a system over time, the mapping of events and the continuation of time are key. Different types of simulations are possible based on the continuance of time (see Figure 11). When referring to continuous simulation, the state of the system changes constantly in time and the relations between the states are usually described by differential equations (e.g., the speed of car during acceleration). See [Liebl 1995] for more details.

4.1.2. Discrete event simulation (DES)

DES was originally designed for the solution of queuing network problems. A sequencing variable was used to represent the monotonically simulated time (the model's clock). There are two ways this is done. The first uses *time driven* advance of the clock and the second uses *event driven* time advance (so-called a time epoch). For time driven advance the period is of fixed length, while in the event driven case the clock jumps to the time of the *next event* (i.e., state changes occur only at the time of an event). The execution, creation and deletion of events (an occurrence that causes a state change), through an ordered *event list*. Implementation concepts for such lists are discussed in [Evan 1988]. Practically all modern simulation engines follow the event driven paradigm. Apart from the definition of an event, we need also to describe the concepts of activities and processes. An activity is a collection of operations that can change the current state of an entity (anything that is described by one or more attributes) whereas a process is a sequence of logically connected events that are ordered in time. Event driven DES can be further refined into the type of scheduling strategy. Theses scheduling strategies can be considered as different worldviews of DES. See for instance [Evan 1988; Carson 1992; Barner 1994; Liebl 1995; Hein 1996] for a more detailed description of these techniques.

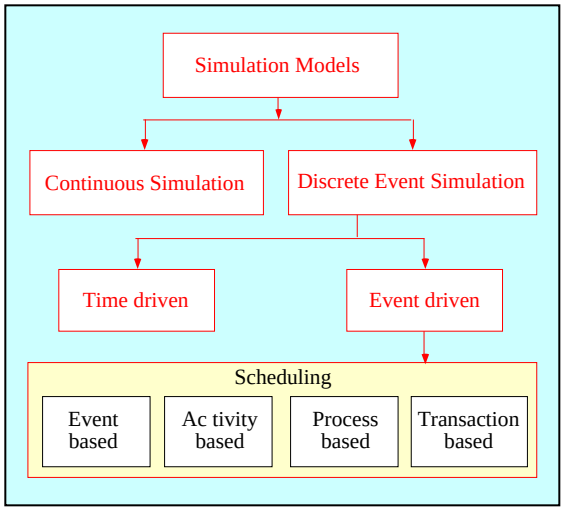


Figure 11. Classification of simulation methods.

4.2. Numerical solution techniques

Simulation is not always the method of choice when it comes to assessing system models. Simulation may be very time and resource consuming. It is often the case that

numerous repetitions of runs are necessary to achieve the confidence levels required. Also, if you are trying to predict an extremely rare event (e.g., MTTF for a high assurance system) then simulation may be completely infeasible. However, Parallel DES has enabled a quantum leap with respect to these limitations. In many cases Markov models provide an attractive alternative and very powerful modeling paradigm.

4.2.1. Generation methods

Since it is cumbersome and errors prone to derive the Markov model directly, GSPNs are used to describe the underlying Markov model. This separation into a higher level model (GSPN model) and the computational model (underlying system of equations) gives the modeler more freedom to concentrate on the system being described. Before being able to solve the underlying system of equations, first the different states of the Markov model are generated form the GSPN description. In general, the analysis of a GSPN can be separated into four steps:

- Generation of the ERG (extended reachability graph) which defines the underlying Semi-Markov process,
- Transformation of the Semi-Markov process into a CTMC by eliminating the vanishing markings (markings with zero sojourn times),
- Transient, or steady state analysis of the CTMC, and
- Calculation of the performance measure(s).

An ERG is a directed graph where the nodes correspond to the markings (states) of the GSPN and the weighted arcs represent the rates or probabilities from which the state transitions occur. There are two basic algorithms used in generating the CTMC (i.e., reachability graph –with vanishing markings eliminated): (1) Post elimination after generating the complete ERG, and (2) On-the-fly elimination which requires less memory because fewer states are actually generated. See [Allmaier 1997; Greiner 1999] for more details.

4.2.2. Methods for linear systems of equations

For the solution of linear systems of equations (steady state case) many exact as well as approximate methods can be applied. Since in the case of a CTMC the underlying Q matrix is usually sparse and has a high dimension, an iterative method is used. These methods work fine as long as the CTMC is not stiff (meaning the values in the Q matrix are different by orders of magnitude) since in this case numerical solution techniques usually do not converge or the convergence is very slow. The solution of a non-stiff CTMC may be accomplished using various standard methods and for an efficient solution method for stiff CTMCs see [Horton 1994; Horton 1995; Greiner 1996]

4.2.3. Methods for differential systems of equations

To compute the vector p(t) of the state probabilities, the matrix P is solved to determine the transient solution using the Kolmogorov differential system of equations. The problem thereby reduces to the computation of the sum of the matrix powers which is numerically critical since negative elements on the main diagonal of Q can lead to numerical instability. A well-known method for transient analysis is Uniformization [Stewart 1994], also known as randomization or Jensen's method. This method is based on the idea that the generator matrix Q be transformed into a stochastic (probability) matrix P with $||P|| < 1$. The fact that the norm of the matrix is smaller than one guarantees that the algorithm for computing the *n*th power is stable. The reader is referred to [Greiner 1999] for additional details.

4.3. Analytical solution techniques

In contrast to the numerical solution techniques, which are based on the model's state space, analytical solution techniques are not. Many efficient algorithms for calculating the performance measures of product form networks have been developed. The two most important exact methods for closed product form queuing networks (PFQN) are the *mean value analysis* (MVA) and the *convolution* algorithm. Both algorithms are iterative methods and unfortunately both (as is the nature of exact methods) are very time and memory consuming once the network grows in size and/or number of nodes. Therefore, approximations to these exact methods are needed. A very well known approximation method is the *summation* method. Methods for closed PFQNs have a simpler formulation than methods for open and mixed PFQNs. To fill the gap between the closed and open/mixed methods, the *closing* method is used which delivers either exact or approximate results, depending on the nature of the underlying solver. Another important class of queuing networks is the non-product form type (NPFQN) which can not be solved using an exact method. The obtained results are always approximate (take for instance the *method of Marie* which delivers very good results). To provide specifics concerning these methods is beyond the scope of this paper. The reader is referred to [Greiner 1999] for additional details.

5. Example

Operating systems that use dynamic priority mechanisms are very common. Nevertheless, it is difficult to develop an accurate analytical model to evaluate their performance, mainly due to the different forms of dependency between the various constituent parts. To illustrate this difficulty, we examine the performance of the BS2000 OPERATING SYSTEM (running the Siemens 7.500 system family [Dedie 1988; Stiegler 1988]), that implements dynamic priorities. A Stochastic Reward Net model (SPN implemented with rewards (i.e., MRM as defined above)) is developed that comprehends the dynamic priority mechanism, several different contexts (e.g., kernel and user contexts), as well as the ability to change the hardware configuration (i.e., adding CPUs). As will be seen, this model suitable for using empirically acquired data. Systematic performance evaluation methods (aside from direct measurement) of computer and operating systems have gained more importance due to their increasing complexity.

5.1. Functional level abstraction

The functionality of a general purpose operating system can be grouped into the following four elementary blocks: I/O, USER, SYSTEM and KERNEL. The I/O block models the activities of the system like disk access. The USER block provides the machinery for running the user's processes. When a user application is started, an interrupt occurs and the interrupt number is determined in the KERNEL block, which decodes it and activates the appropriate interrupt handling routine (IHR). The IHR executes in the SYSTEM block. For each elementary block an analytic sub-model is defined, whose degree of accuracy depends on the purpose of the analysis. In the following, we concentrate on the modeling of one priority mechanism implemented inside the USER block. The priority mechanism is crucial because it guarantees that high priority jobs will have shorter response times than lower priority jobs. Two possible priority strategies include dynamic (priorities can move up and down to prevent a job from starvation) and static priorities (often a mixture of preemptive and non-preemptive static service). Here we evaluate increasing-decreasing dynamic priorities since this type of priority assignment is commonly used in real time systems.

5.1.1. Block diagram model

Because it is expensive to run code in the kernel context, we try only to run basic (short lived) tasks there, while other work is performed by lower level hardware (e.g., DMA). Thus, the kernel blocks other tasks less time who then can perform more useful work. This approach, where the kernel context performs only the basic work of a system call is called micro-kernel approach [Deitel 1990] (in contrast to the macro-kernel approach where all operations are performed in the kernel). The micro-kernel determines only the interrupt number and initiates the corresponding routine before it exits. Consider, the case of a page fault. A micro-kernel system traps to the system kernel whom performs the corresponding routine to load the new page and then goes back to the interrupted program. In this way, its not necessary for the CPU to stay in the kernel context all of the time. As soon as the location of the page is found and loaded into the main memory, the CPU is free to do other useful work. Older operating systems use the macro-kernel approach, new ones use the micro-kernel approach. Only basic interrupt handling and error recovery is done in the micro-kernel while the rest is done outside the kernel (see Figure 12 in which the central process bubble represents the micro-kernel).

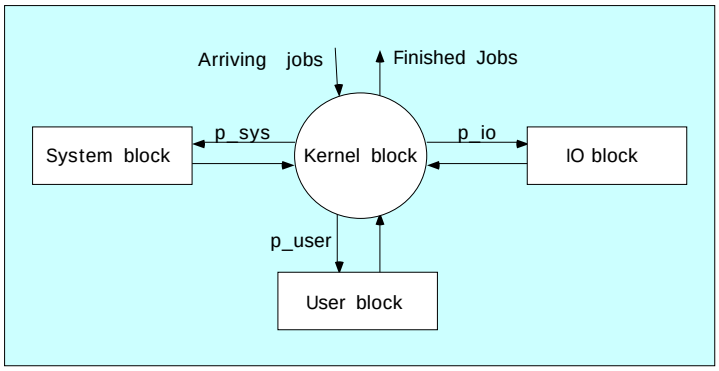


Figure 12. Block diagram for a micro-kernel operating system.

5.1.2. Dynamic priorities

Having developed the block diagram, let's apply this representation to the real operating system with dynamic priorities. Compared to Unix, BS2000 supports three different processing modes [Deitel 1990]: transaction, dialog and batch processing. The transaction-processing mode gains importance as many commercial products are based on distributed database access. In this case, mutual exclusion is needed as is provided by the transaction-processing mode. Take, for instance, airline reservation databases distributed over the world. In such a system, we must be guaranteed that any held reservation can not be overbooked. BS2000 supports this mechanism and to model this mode, we include the task scheduler PRIOR to show how the priority mechanism works (see [Dedie 1988; Stiegler 1988; Greiner 1995a; Greiner 1995b]).

5.2. Modeling the operating system with a stochastic reward net (SRN)

System complexity makes it impossible to manually construct the entire state space. We therefore use SPNs as shown in Figure 13. The basic model is provided in the top-half and the details of the USER context are provided in the bottom half. It is straightforward to define the different subsystems from the block diagram representation. There is a 1:1 mapping between the block diagram of the Figure 12 representation and the BS2000 job model in Figure 13. The block diagram model can be mapped directly to the SRN model. Guards (also known as enabling functions) and multiplicity arcs are used for a more compact description.

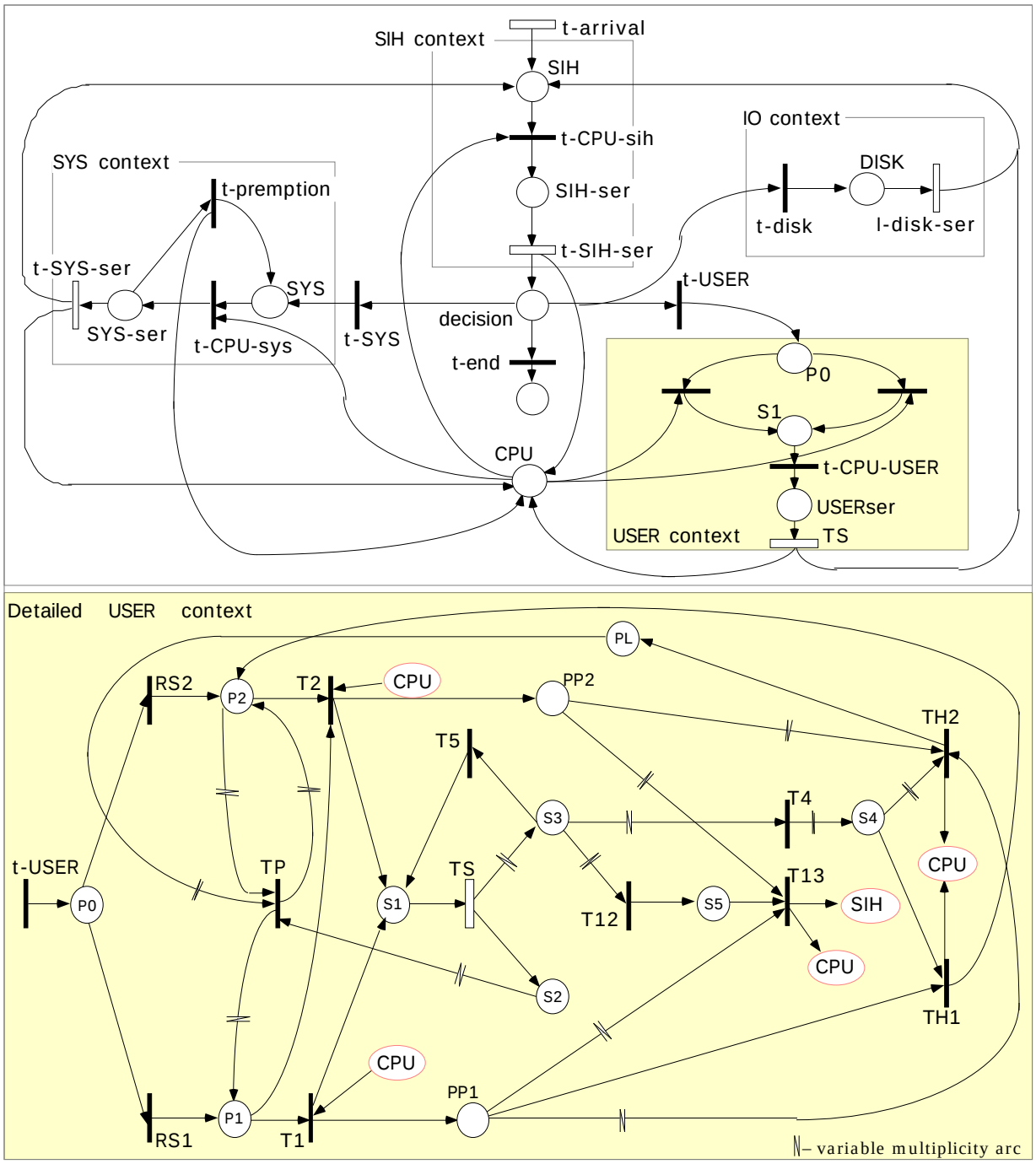


Figure 13. SPN model of the BS2000 operating system (bottom half gives the *Detailed User Context*).

The arrival of jobs to the system is represented by the transition $t_{arrival}$. To limit the number of states, a maximum number *K* of jobs in the system is given. If this number is reached, no other jobs may enter the system. A guard is associated with the transition $t_{arrival}$ which returns TRUE when the total number of jobs in the SIH (interrupt handling and error recovery), SYS (perform system calls), I/O (set up I/O, memory access, disk access) and USER contexts is $\leq K$. The guard function associated with transition $t_{arrival}$ assuming only two priority classes is shown here in Figure 14. The notation #(P) is used to indicate the number of tokens in place *P*:

```
enabling_type ftarrival() {
    int tot;
    tot = #(SIH) + #(SIHser) + #(SYS) + #(SYSser) +
        #(Disk) + #(P1) + #(P2) + #(PL) + #(PP1) + #(PP2);
    return(tot <= K ? 1 : 0);}
```

Figure 14. Enabling function for transition t-arrival.

A token in place CPU (Figure 13 top) indicates the CPU is available. All contexts require the CPU be activated and is represented by the immediate transitions, t_CPU_sih, t_CPU_sys, t_disk, T1, ..., Tn. When a new job enters, it is processed inside the SIH context. Transition t_SIH_ser represents the service time. To exit the SIH context, a token is put back in the *CPU* place to indicate the CPU is now free. At this point, there are four possible actions as described in the Table 5.

Table 5. Alternative paths for a job leaving the System Interrupt Handler context.

Actions A Job May Take After Leaving the SIH Context:	
1.	Move to the I/O context, where jobs are served on a FCFS basis. When the I/O is complete, the CPU is informed by an interrupt. Therefore the token is given back to the SIH queue to wait for further service. The I/O works independently from the rest of the system, which means that both the CPU and I/O can work in parallel.
2.	Move to the SYS context to perform a system call. If some other job arrives in the SIH context during the system call, the job currently being served in the SYS context is preempted (i.e., jobs in the SIH context have higher priority). The token that indicates the job has been preempted is moved to the SYS queue (the SYS place in the diagram) and a new token is put back in the CPU place (indicating the SIH job can now have the CPU). If no preemption occurs, the job (i.e., token) that was just served (to completion), is moved back into the SIH queue and a token that represents the CPU is free is moved onto the CPU place.
3.	Move to the USER context. This context is given here as a block diagram and shall be described in more detail in the following section. Up to here we can assume any kind of micro-kernel general purpose operating system because nothing has been said about the characteristics of the USER context. To model BS2000 we choose a dynamic priority system.
4.	Leave the system because the job has completed.

The decision (with a certain probability) of which context (SYS, I/O or USER) the job switches after it is served by the SIH context is done at the place *decision*. This probability can be given in the program as a variable. We do not consider the system as one big complex system but use a hierarchical approach to represent the system. It is up to the modeler what level of detail to use for the different boxes. In this case, apart from the USER context, all other contexts are modeled in a very simple way. It is possible to give a more detailed description for each box but we are mostly interested in the dynamic priority structure of the system. Thus, a detailed model of the USER context is used. However, a more detailed model of the I/O behavior can be developed while giving a simpler model for the USER context, etc. In this way, a hierarchical representation of the system as an SRN with different components (boxes) is very flexible given the choice of which box to study and model in more detail. It is also possible to consider several system aspects at the same time in the model (like dynamic priority system and I/O behavior). In contrast, the state space of the system will probably be very big so that it would be necessary to use hierarchical system solution techniques [Ciardo 1993]. Note, by varying the number of tokens in the place CPU, it is possible to change the number of CPUs available. Thus, it is possible to model a single processor and a multiprocessor operating system with only minor changes. For a detailed discussion of the USER context see [Greiner 1995a; Greiner 1995b]. The assumptions used for this example are given in Table 6.

Table 6. The assumptions made about the BS2000 operating system in developing the SRN of Figure 13.

System Assumptions	
1.	Jobs start and finish in the kernel context. For BS2000, this context is called SIH (*System Interrupt Handling*).
2.	The system is open with arrival rate $\lambda_{arrival}$. To prevent the state space from becoming infinite we assume a maximum number *K* of jobs in the system. The maximum number of jobs in the system is 10.
3.	Context switches within the CPU are assumed to take place in zero time (it is easy to relax this assumption) and jobs waiting for service increase their priority. It is difficult to model the real increase-decrease of priorities and thus, this behavior is approximated by linear functions.
4.	As soon as a job obtains the CPU it works until it is preempted or it finishes.
5.	Priority order within contexts is: SIH > SYS > USER (and only one *user* job class is allowed).
6.	All jobs are preemptable. When a higher priority job arrives, the job in the CPU is preempted (i.e., its priority is lower).
7.	Only jobs waiting in the USER context increase and decrease their priority (dynamic priorities).
8.	Typically, during day only 5% of the arriving jobs are batch-jobs [Stiegler 1988]. Therefore, we assume no batch jobs are run during daytime. At night, most jobs are batch, so this assumption is not valid. It is possible to introduce another job class for batch jobs, but this would severely increase computation time. Our model is based on daytime data.

The USER context of Figure 13 shows two priority levels (only two are shown for the sake of simplicity, but we could easily add more levels). This context is entered by a job when the transition t_{user} fires (priority 1 > priority 2 … > priority n). The priority class to which an arriving job belongs is determined according to a probabilistic distribution (random switch) modeled by the immediate transitions (RS1 and RS2). Place P1 models the queue of priority class 1. Place PL represents a macro-state where all the jobs with priority lower than n are stored (here n is 2). Transitions T1 and T2 are enabled when there are no other jobs being served in side the USER context. In other words, if the total number of tokens in places PP1, and PP2 is equal to zero. A guard (enabling function) can be defined as shown in Figure 15. Moreover, as the USER has a lower priority that any of the other contexts (except for the IO context), it can get the CPU only when there are no other jobs in any of the other contexts. This is enforced by assigning a lower priority to transitions Ti than transitions t_CPU_sys and t_CPU_sih. As soon as either of the transitions T1 or T2 fires, a token is removed from the CPU place indicating that the CPU is now busy processing the USER context job. Note, jobs must be sequentially processed according to their priority. Thus, the inhibitor arc from place P1 to transition T2 (or in general, Pi to Ti+1) guarantees that the priority class i+1 jobs wait for the class i jobs to finish. Also, a token in place S1 indicates the CPU is busy processing a USER context job whose priority is identified as i if a token is in place PPi.

```
Int fT1T2() {return((#(PP1)==1 || #(PP2)==1) ? 0:1);}
```

Figure 15. Enabling function (guard) for transitions Ti (i=1,2)

Transition TS models the time slot assigned to the job. For each time slot the priority of the running job is decreased by one while the priority of the jobs waiting are increased by one. Thus, when the time slot has completed (i.e., TS fires) one token is deposited in place S2 and two tokens are deposited in place S3. When the number of tokens in place S3 is equal to a predefined value PI (i.e., cycles allowed prior to inverting the priority), then transition TP is enabled and the priority of waiting jobs (i.e., those which have not been processed) is increased by one. Note that the arcs entering/departing this transition TP are variable multiplicity arcs that remove from a priority class as many tokens as they find and then transferring them to the immediately higher priority class. By varying the PI cycle value, different priority changing thresholds can be tested.

```
Int fT4() {
if (#(S3)==0 || #(S1)!=0) return(0);
      else { if (#(P1)>0) return (1);
            else { if (#(P2)==0) return(0);
                  else { if (#(PP1)==0) return(1);
                        else { if ((#(S3)-1)==2) return(1);
                              else return(0);
                        }
                  }
            }
      }
}
```

```
Int fT5() {
if (#(S1)!=0 || #(P1)>0 || #(S3)==0 || (#(S3)-1)==N ||
      #(SIH)>0) return(0);
      else { if (#(P2)==0) return (1);
            else { if (#(PP1)==0) return(0);
                  else { if ((#(S3)-1)==2) return(0);
                        else return(1);
                  }
            }
      }
}
```

Figure 16. Enabling functions (guards) for the transitions T4 and T5.

The value obtained by decreasing the number of tokens in place S3 by one indicates how much the priority level of the running job has been reduced (remember, a new token is deposited in S3 each time transition TS fires). Transitions T4 and T5 are enabled according to this value (i.e., based on the priority of the job being served and on the priority of the waiting jobs). Transition T4 indicates the probability Ppre that a job preempted by higher level priority jobs still remains inside the USER context. Thus, 1-Ppre is the firing probability of the immediate transition T12 which models a job exiting the User context. Transition T5 models that instead of leaving, the job cycles again back for another slot of CPU time inside the User context with probability Pcycle. The enabling functions that define this behavior are shown in Figure 16.

```
int fTH1(){return((#(S4)==1 && #(PP1)==1) ? 1:0);}

int fTH2(){return((#(PP1)==1 && #(S4)>1) || (#(PP2)==1 && #(S4) > 0) ? 1:0);}
```

Figure 17. Enabling functions (guards) for the transitions TH1 and TH2.

Transitions THi (i=1,2 in this example) are enabled according to the original priority of the preempted job and based the number of CPU cycles that the job has already taken. These transitions deposit a token to a priority queue according to the final priority of the processed job which is evaluated as the difference between its original priority and the number of tokens in place S3 minus one. Place PL, the macro queue (described above) is entered by all preempted jobs whose final priority is (theoretically) less than n [n=2 here]. Since, in this example, only two priority classes are assumed which gives rise to the enabling functions of Figure 17. In such a case, place PL is entered by all priority one jobs that had more than two CPU cycles and by all priority two jobs that had at least one cycle. A token is deposited in the CPU place when transition THi or T13 fires to release

Table 7. The guards, variable arc functions and transition-priority relationship for the SRN of Figure 13.

Transition	**Guard**
t-arrival	See Figure 14
T1 and T2	See Figure 15
T4 and T5	See Figure 16
TH1 and TH2	See Figure 17
TP	#(S2) = PI

Transition-priority relationship
(t-sys, t-end, t-USER, tDISK) > t-pre > t-CPU-SIH > t-CPU-SYS >
(T1, T2, T4, T5, TH1, TH2, RS1, RS2, T12, T13) >
TS, t-arrival, t-SIH-ser, t-SYS-ser, t-DISK-ser)

Arc designations	**Variable arc functions**
PP1 → T13	#(PP1)
PP2 → T13 and PP2 → TH2	#(PP2)
S3 → T12 and S3 → T4 and T4 → S4	#(S3)
PL → TP and TP → P2	#(PL)
P2 → TP and TP → P1	#(PPL)
T2 → S3	#(S2)=PI

the CPU. When a job leaves the USER context (i.e., the firing of transition T13) it immediately enters the SIH context as represented by the arc from T13 to place SIH. Table 7 gives the enabling (guard) functions, the transition-priority relationship and the variable (multiplicity) arc functions for the two priority class model presented in this example. In addition, to extend this example by adding additional priority classes would require that for each new job class j, a new place Pj be added into the SRN of Figure 13.

5.3. Results and conclusions

The results from solving the SRN model of the BS2000 with the s_io, s_sys, s_user and s_sih firing times (expressed in msec) of the transitions t_disk_ser, t_sys_ser, TS and t_SIH_ser, are presented in Table 8 (p_io, p_sys, p_user and p_end represent the distribution of probability for leaving the decision place). To find the meaning of the different transition probabilities see Figures 12 and 13. Transient and steady state behavior of the system are shown first. The results for the dynamic priority system are then compared to an operating system with static priorities. Note, the network is not analyzed for liveness, boundedness, etc.

Table 8. Parameterization of the BS2000 system SRN.

System Parameters (job arrival rate $\lambda_{arrival}$ = 0.005)		
Component Definition	**Transition Probability**	**Service Time**
I/O Subsystem Context	p_io = 0.05	s_io = 20
System Context	p_sys = 0.40	s_sys = 1.0
User Subsystem Context	p_user = 0.54	s_user =1.0
Kernel Subsystem Context	p_end = 0.01	s_sih = 0.5

5.3.1. Transient analysis

At first we considered transient analysis to find out how long it takes for the operating system to reach steady state and how the jobs move between different priority classes over time. The result is shown in Figure 18 (left). It can be seen that within some seconds the system reaches steady state. If we compare this time to the time it takes to boot the system then the transient behavior of the system can be neglected. Thus, the steady state assumption for our models is justified. Furthermore it can be seen that for our chosen values after a very short time most of the jobs move to the highest priority queue. This depends of course a lot on the initial distribution of the jobs over the priority classes and the system parameters. But in our case nearly no jobs are waiting in the lowest priority queue.

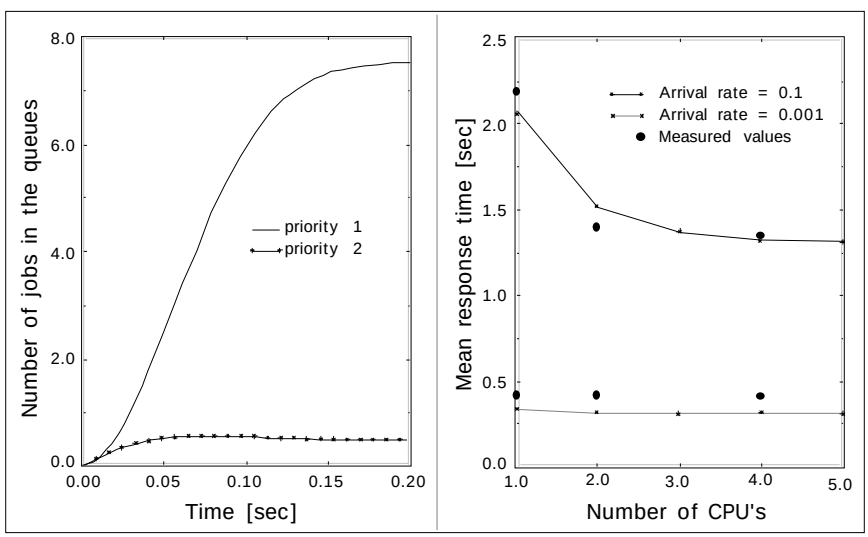


Figure 18. Queued jobs over time, arrival rate 1.0 (L) and variation in the number of CPUs (R).

5.3.2. Steady state analysis

By varying the number of CPU's in the system we can discover some interesting results. One might expect, that

the higher the number of CPU's, the lower is the mean response time. This is not the case as can be seen in Figure 18 (right), because the higher the number of processors, the more the system is I/O bound (the system must wait for I/O to finish). For the chosen system parameter values (shown above) the mean response time for a low arrival rate is nearly constant even as we increase the number of CPU's. For a high arrival rate, the mean response time drops considerably by adding a second or third CPU but its not worth having more than three processors since the corresponding decrease in the mean the mean response time is so small. The results for this plot are also compared to real system measurements and it can be seen that our model results are very close to the measured results.

Table 9. Results derived from the different numbers of phases in the Erlang distribution function.

Results for different distribution functions				
Erlang Distr. Phases	Arrival rate	$\bar{t}$	λ	Number of states
1	2	3.734	1.900	22780
2	2	3.693	1.917	45550
4	2	3.674	1.925	91010
6	2	3.667	1.928	136630
8	2	3.664	1.930	182170

One can argue that the assumption of an *exponential distribution* for transition *TS* is not very realistic. Therefore, consider the analysis of the Petri net using a phase type distribution. Results are shown in the Table 9 and there are no real differences in the performance values for throughput and response time as the number of phases is increased. However the number of states is increased considerably! Similar results were obtained when analyzing the UNIX operating system [Greiner 1993].

To increase (optimize) the system performance without changing the number of CPUs we need to determine which context a jobs spends most of its computation time. In Figure 19 (top) the relative time spent in each of the contexts is provided based on the given parameter set. The longer the service time for a USER job the more time that is spent in the USER context. This is natural based on the type of workload that was chosen (i.e., transaction processing) environment. In the chosen environment, jobs spend most of their time in the USER context. As soon as a transaction is started, only a few kernel commands are needed. The rest of the transaction is performed outside the kernel in the USER context. If we want to increase the performance of the system with its high service times it is necessary to increase the performance of the USER context because almost ninety five percent of its CPU time is spent in the USER context. In this case its useless to increase IO performance because, as can be seen in Figure 19 (bottom), most of it's lifetime a job spends using the CPU (assuming long service times). The lower service time becomes, the more time a job spends in the IO and for very short service times the job stays about fifty percent of its computation time in the IO. In such cases, it is worth trying to increase the IO performance because as is evident from Figure 19 (bottom), the CPU is idle for about fifty percent of the job's computation time. Naturally, this is due to fact that IO is so much slower than the CPU (i.e., only one IO cycle for the job will cause many idle cycles for the CPU).

This example shows how to model dynamic priorities in an operating system using a SRN technique. The system was specified with certain assumptions. The most important assumption is the approximation of the priority schema using linear functions. The system model is quite open. This is the case because after the block diagrams for

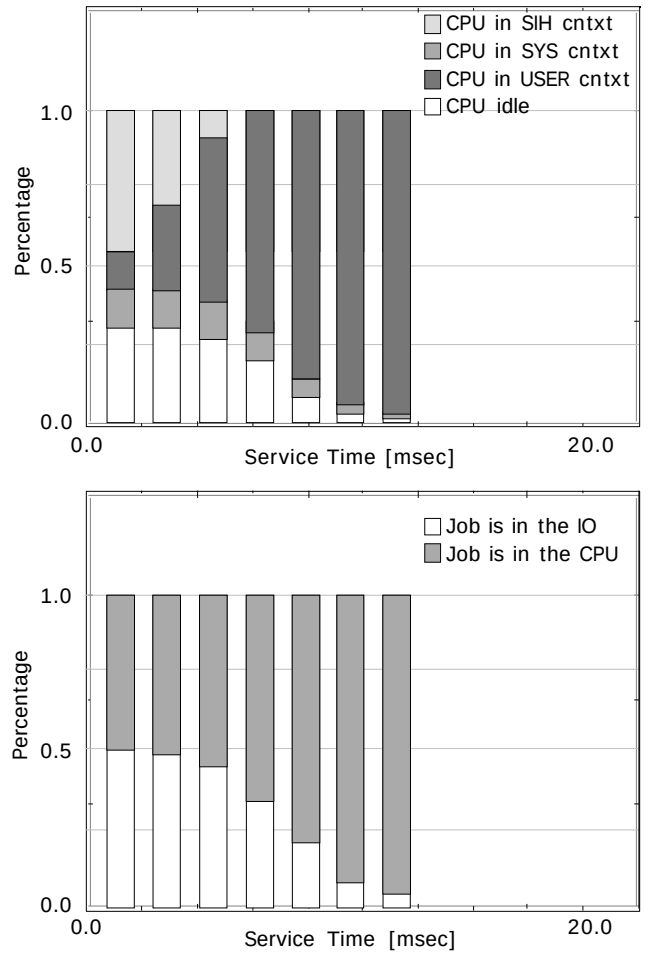


Figure 19. Percentages: a job spends in the different CPU contexts (top), a job spends in the CPU or IO (bottom).

micro kernel system were developed, the BS2000 system could easily be mapped to the SPN model by characterizing the internal behavior of each subsystem of the block diagram. The SRN was described using CSPL (C-based SPN language [Ciardo 1994]). The block diagram can be easily transformed into a Petri net. Emphasis is put on the BS2000priority schema and consequently, a more detailed model for the USER block was presented. It is also possible to concentrate on other parts of the system, like I/O, by specifying this box in greater detail. In this sense, the model is very flexible. It is up to the modeler to decide on which part of the system to focus. For the plots in Figure 18, only 2 priority classes were used, but a more general Petri net model with *n* priority classes can be seen in [Greiner 1995a]. The actual number *n* is open to the modeler.

6. Summary

In this paper we have motivated the need for tool supported rigorous methods used for reasoning about software and systems, introduced a framework codified by the *modeling cycle* (Figure 2a and 2b). This approach is predicated by the following premise:

> *The correctness, safety and robustness of a critical system specification are generally assessed through a combination of rigorous specification capture and inspection; formal modeling and analysis of the specification; and execution and/or simulation of the specification (or possibly a model of such)* [Yen 1998].

Furthermore, we have introduced some systematic formal techniques for the creation and composition of software models through a process of abstraction and refinement, and enumerated several formal modeling techniques within this context currently available to the modeler (i.e., reliability and availability models, performance and functional models, performability models etc.). This discussion has included a more precise dialogue on stochastic methods (i.e., DTMC and CTMC) and their formulation. In addition we briefly reviewed the underlying theories and assumptions that are used to solve these models for the measure of interest (i.e., simulation, numerical and analytical techniques). Finally, we presented a small example that employs a stochastic Petri net model of an operating system with dynamic priorities.

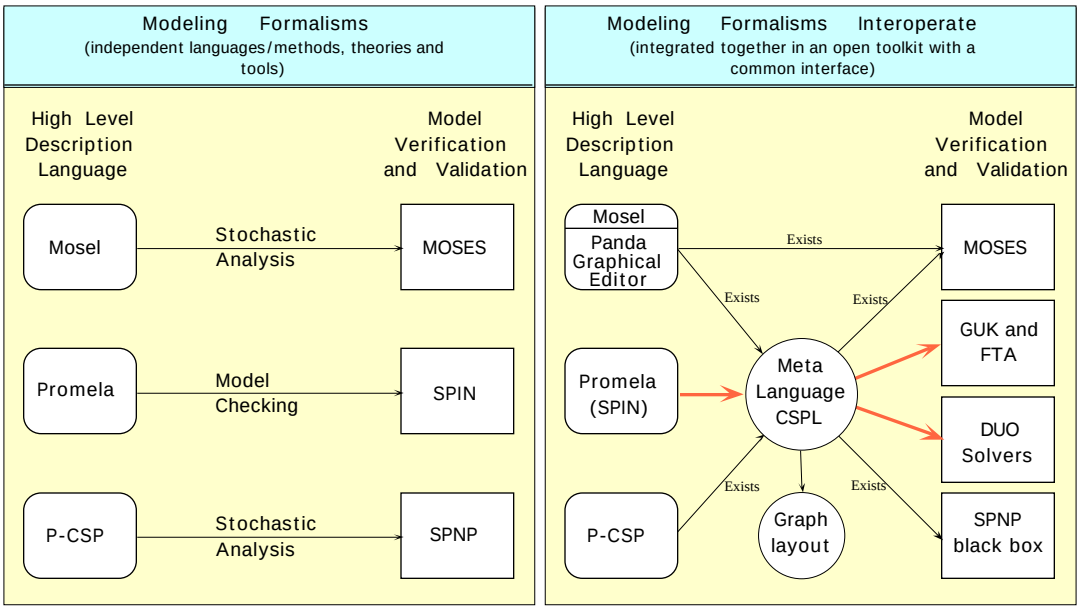


Figure 20. Framework architecture for model based predictive analysis.

7. Future work

Our goal is to continue to develop and refine this approach as an open framework coupled with useful formal representations and analysis capability for architectures that relate specifications to programs and programs to behavior. Figure 20 shows a general architecture for DOU which defines a prototype modeling toolkit that promotes interoperable use of various formalisms (including at least one language used in very powerful model checking programs) [Burch 1992; Holzmann 1993; Holzmann 1997]. This environment will more easily facilitate the modeling process and support the analyst with regard to prediction and assessment. The meta-language used as an intermediate canonical representation form is CSPL (C-based Stochastic Petri net Language) [Ciardo 1994].

The challenges that lie ahead for software and systems are underscored by the seemingly irreversible trend toward increased complexity as our industry strives to conquer faster, cheaper and better technologies. This evolutionary trend underscores the need to develop and validate methods and tools for the creation of safe and correct software.

8. References

Allen, O. (1978), *Probability, Statistics and Queuing Theory*, Academic Press, New York, NY.

Allen, O. (1990), *Probability, Statistics and Queuing Theory*, 2nd, Academic Press, New York, NY.

Allmaier, S. C., Kowarschik, M. and Horton, G. (1997), "State Space Construction and Steady-State Solution of GSPNs on a Shared-Memory Multiprocessor," *7th. Int'l Wkshp on Petri Nets and Performance Models*, IEEE Computer Society Press, Los Alamitos, CA, pp. .

Balbo, G., Donatelli, Susanna and Franceschinis, Giuliana (1992), "Understanding Parallel Program Behavior through Petri Net Models," *Journal of Parallel and Distributed Computing*, *15*, 3, 171-187.

Balbo, G. (1993), "Performance Evaluation and Concurrent Programming," *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, Springer-Verlag, Berlin, pp. 1-13.

Balbo, G., Donatelli, S., Granceschinis, G., Mazzeo, A., Mazzocca, N. and Ribaudo, M. (1994), "On the Computation of Performance Characteristics of Concurrent Programs Using GSPNs," *Performance Evaluation*, *19*, 18, 195-222.

Balbo, G. (1995), "On the Success of Stochastic Petri Nets," *Int'l Wkshp on Petri Nets and Performance Modeling*, IEEE CS Press, Los Alamitos, CA, pp. 2-9.

Barner, J. (1994), "Entwicklung, Implementierung und Validierung von Analytischen Verfahren zur Anlyse von Prioritatsnetzen," Masters Thesis, In CS Dept. IMMD IV, Univ. of Erlangen-Nurnberg, Germany.

Blake, J. T., Reibman, Andrew L. and Trivedi, Kishor S. (1988), "Sensitivity Analysis of Reliability and Performability Measures for Multiprocessor Systems," *SIGMETRICS*, ACM, New York, NY, pp. 177-186.

Bolch, G. (1989), "Leitungsbewertung von Rechensystemen," TR-1989-IMMD IV, Leitfaden und Monographien der Informatik - B.G. Teubner Verlagsgesellschaft, Stuttgart.

Buchholz, P. (1994), "Compositional Analysis of a Markovian Process Algebra," *Proceedings PAPM*, CLUP, pp. 233-245.

Burch, J. R., Clarke, E.M., and McMillan, K.L. (1992), "Symbolic Modeling Checking: 10^{20} States and Beyond," *Information and Computation*, *98*, 42, 142-170.

Buzen, J. (1971), "Queuing Network Models of Multiprogramming," Dissertation (Ph.D.), In Division of Engineering and Applied Physics, Harvard, Cambridge, MA.

Calzarossa, M., and Massari, M. (1991), "Workload Analyzer Tool -WAT- User Guide," *Fifth Int'l Conf. on Modeling Techniques and Tools for Computer Performance Evaluation*, Univ. di Milano, Diparimento di Scienza dell'Informazione, pp. 1-15.

Carson, J. (1992), "Modeling," *Winter Simulation Conference (Group on Simulation)*, SIGSIM, ACM New York, NY, pp. 82-87.

Chiola, G. (1985), "A Software Package for the Analysis of Generalized Stochastic Petri Net Models," *Int'l Wkshp on Timed Petri Nets*, IEEE Computer Society Press, Los Alamitos, CA, pp. 136-143.

Choi, H., Mainkar, Varsha and Trivedi, Kishor S. (1993), "Sensitivity Analysis of Deterministic and Stochastic Petri Nets," *MASCOTS'93 International Workshop on modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 271-276.

Ciardo, G. (1987), "Toward a Definition of Modeling Power for Stochastic Petri Net Models," *Int'l Wkshp on Petri Nets and Performance Models*, IEEE Computer Society Press, Los Alamitos, CA, pp. 54-62.

Ciardo, G., Muppala J. and Trivedi, K.S. (1989), "SPNP: Stochastic Petri Net Package," *3rd Int'l Wkshp on Petri Nets and Performance Models*, IEEE Computer Society Press, Los Alamitos, CA, pp. 144-151.

Ciardo, G., Muppala J. and Trivedi, K.S. (1991), "On the Solution of GSPN Reward Models," *Performance Evaluation*, *12*, , 237-253.

Ciardo, G., Muppala J. and Trivedi, K.S. (1992a), "Analyzing Concurrent and Fault-Tolerant Software Using Stochastic Reward Nets," *Jr. of Parallel and Distributed Comp.*, , 255-269.

Ciardo, G., Marie, R., Bruell, S., and Trivedi, K. (1992b), "Performability Analysis Using Semi-Markov Reward Processes," *IEEE Transactions on Computers*, *39*, 10, 121-1264.

Ciardo, G., Blakemore, A., Chimento, P.F., Muppala, J.K., Trivedi, K.S. (1992c), "Automated Generation and Analysis of Markov Reward Models Using Stochastic Rewards Nets," In *IMA Volumes in Mathematics and its Applications*, C. Meyer and R.J. Plemmons, Springer-Verlag, Berlin, Germany, pp. .

Ciardo, G., Trivedi K.S. (1993), "A decomposition approach for stochastic reward net models," *Performance Evaluation*, *18*, 1, 37-59.

Ciardo, G., Fricks, R., Muppala J. and Trivedi, K.S. (1994), "SPNP Users Manual Version 4.0," TR-1993-10, Duke Univ., Dept. of EE, Durham, NC.

Dedie, G. (1988), "Nucleus BS2000 -- Technical Description," TR-BS200-88, .

Deitel, H. M. (1990), *An Introduction to Operating Systems*, Addison Wesley, Reading, MA.

Donatelli, S., Franceschinis, G., Mazzocca, N., and Russo, S. (1994), "Software Architecture of the EPOCA Integrated Environment," *7th Int'l Conference on Performance Evaluation, Modeling Techniques and Tools (LNCS 794)*, Springer-Verlag, Berlin, pp. 335-352.

Donatelli, S., Ribaudo, M. and Hillston, J. (1995), "A Comparison of Performance Evaluation Process Algebra and Generalized Stochastic Petri Nets," *5th Int'l Wkshp PNPM*, IEEE Computer Society Press, Los Alamitos, CA, pp. 158-168.

Evan, J. (1988), *Structures of Discrete Event Simulation: An Introduction to the Engagement Strategy*, Ellis Horwood Limited, Chicester (W. Sussex) UK.

Fleischmann, G. (1989), "Modellierung und Bewertung Paralleler Programme," Ph.D. Dissertation, In Computer Science IMMD IV, Univ. of Erlangen-Nurnberg, Erlangen, Germany.

Gangl, P. (1993), "Simulation - eine Schlusseltechnologie der 90er Jahre: Hoher Nutzen aber geringer Kentnisstand in der Wirtschaft," *8th Symp. on Simulation Techniques*, Vieweg-Verlag, Germany, pp. 103-106.

Gordon, W., Newell, G. (1967), "Closed Queuing Systems with Exponential Servers," *Operations Research*, *15*, 2, 245-265.

Greiner, S. (1993), "Leistungsbewertung des Betriebssystems UNIX mit Hilfe approximativer analytischer Methoden," TR-1993-9, Univ. of Erlangen, CS Dept. IMMD IV, Erlangen, Germany.

Greiner, S. (1995a), "Performance Evaluation of Dynamic Priority Operating Systems," *5th Int'l Wkshp PNPM*,

IEEE Computer Society Press, Los Alamitos, CA, pp. .

Greiner, S. (1995b), "An Analytical Model for the Operating System BS2000,", CS Dept. IMMD IV, Univ. of Erlangen, Erlangen, Germany.

Greiner, S., and Horton, G. (1996), "Analysis of Stiff Markov Chains with the Multi-level Method," *European Simulation Symposium*, pp. 801-805.

Greiner, S. (1999), "Stochastic Analysis of Computer Science Applications: Theory, Models and Solution Methods," Ph.D. Dissertation, In Computer Science, IMMD IV, Univ. of Erlangen-Nurnberg, Germany.

Heidelberger, P., and Goyal, Ambuj (1988), "Sensitivity Analysis of Continuous Time Markov Chains Using Uniformization," *Computer Performance and Reliability. Proc. of the 2nd Int'l MCPR Wkshp*, North-Holland, Amsterdam, pp. 93-104.

Hein, A., and Goswami, K.K. (1996), "Conjoint Simulation - A Technique for the Combined Performance and Dependability Analysis of Large-Scale Computer Systems," *Advances in Simulation*, IEEE Computer Society Press, Los Alamitos, CA, pp. 68-77.

Hoare, C. A. R. (1985), *Communicating sequential processes*, Prentice-Hall Int'l, London, London.

Holzmann, G. J. (1993), "Design and validation of protocols: a tutorial," *Computer Networks and ISDN Systems*, *25*, , 981-1017.

Holzmann, G. J. (1997), "The Model Checker SPIN," *IEEE Trans. on Software Engineering*, *23*, 5, 279-95.

Horton, G., and Leutenegger, S. (1995), "On the Utility of the Multi-Level Algorithm for the Solution of Nearly Completely Decomposable Markov Chains," *Second Int'l Wkshp on the Numerical Solution of Markov Chains*, Kluwer, Boston, pp. 425-442.

Horton, J., and Leutenegger, S. (1994), "A Multi-level Algorithm for Steady State Markov Chains," *SIGMETRICS*, ACM, New York, NY, pp. 191-200.

Jackson, J. (1963), "Jobshop-Like Queuing Systems," *Management Science*, *10*, 1, 131-142.

Kavi, K. M., Sheldon, F.T. (1994), "Specification of Stochastic Properties with CSP," *Int'l Conference on Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 288-293.

Kleinrock, L. (1975), *Queuing Systems, Volume 1: Theory*, John Wiley, New York, NY.

Koller, D., McAllester, David, and Pfeffer, Avi (1997), "Effective Bayesian Inference for Stochastic Programs," *Fourteenth Nat. Conf. AAAI*, AAAI, Menlo Park, CA, pp. 740-747.

Laprie, J. C., Kaaniche, M. and Kanoun, K. (1995), "Modeling Computer Systems Evolutions: Non-Stationary Processes and Stochastic Petri Nets - Application to Dependability Growth," *Int'l Wkshp on Petri Nets and Performance Models*, IEEE Computer Society Press, Los Alamitos, CA, pp. 221-230.

Lazowska, E. D., Zahorjan, J. , Graham, G.S., Sevcik, K.C. (1984), *Quantitative System Performance -Computer Sys. Analysis Using Queuing Network Models*, Prentice Hall, Int'l, London.

Levenson, N. G., and Stolzy, J.L. (1987), "Safety Analysis Using Petri Nets," *IEEE Trans. on Software Engineering*, *SE-13*, 3, 386-397.

Lewis, A. D. (1988), "Petri Net Modeling and Software Safety Analysis: Methodology for an Embedded Military Application," Masters, In Computer Science, Naval Postgraduate School, Monterey, CA.

Liebl, F. (1995), *Simulation*, Oldenbourg, Munich.

Logothetis, D. a. T., K.S. (1997), "The Effect of Detection and Restoration Times for an Error Recovery in Communication Networks," *Journal of Network and Systems Management*, *5*, 2, 173-95.

Mainkar, V., Choi, Hoon and Trivedi, Kishor (1993), "Sensitivity Analysis of Markov Regenerative Stochastic Petri Nets," *5th Int'l Wkshp on Petri Nets and Performance Modeling*, IEEE Computer Society Press, Los Alamitos, CA, pp. 180-189.

Malhotra, M., and Trivedi, K.S. (1993), "A Methodology for Formal Expression of Hierarchy in Model Solution," *Fifth Int'l Wkshp on PNPM*, IEEE Computer Society Press, Los Alamitos, CA, pp. 258-267.

Marie, R., and Jean-Marie, Alain (1993), "Quantitative Evaluation of Discrete-Event Systems: Models, Performance and Techniques," *Fifth Int'l Wkshp on Petri-Nets and Performance Modeling*, IEEE Computer Society Press, Los Alamitos, CA, pp. 2-11.

Marsan, M. A., Balbo, G. and Conte, G (1984), "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Trans. on Computer Systems*, *2*, 1, 93-122.

Milner, R. (1989), *Communication and Concurrency*, Prentice-Hall, Int'l, London.

Molloy, M. K. (1982), "Performance Analysis Using Stochastic Petri Nets," *IEEE Trans. on Computers*, *C-31*, 9, 913-917.

Muppala, J. K., Wang, Wei and Trivedi, Kishor, S. (1994), "Dependability Evaluation Through Measurements and Models," Fnl. Rpt. NSF Grant CCR-9108114, Duke Univ., EE Dept., Durham, NC.

Muppala, J. K., Ciardo, Gianfranco and Trivedi, Kishor S. (1994), "Stochastic Reward Nets for Reliability

Prediction," TR-1994-12, Duke Univ., EE Dept., Durham, NC.

Murata, T. (1989), "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, *77*, 4, 541-580.

Olderog, E.-R. (1986), "TCSP - Theory of communicating sequential processes," *LNCS-255*, , 441-465.

Olderog, E.-R. (1987), "Operational Petri Net Semantics for CCSP," , *LNCS-266*, , 196-223.

Reibman, A., Smith, R., and Trivedi, K. (1989), "Markov and Markov Reward Model Transient Analysis: An Overview of Numerical Approaches," *European Journal of Operational Research*, *40*, 2, 257-267.

Reibman, A., and Veeraraghavan, M. (1991), "Reliability Modeling: An Modeling Overview for System designers," *Computer*, *24*, 4, 49-57.

Rettelbach, M. (1996), "Stochastic Process Algebras with Timeless Activities and Probabilistic Branching Probabilities," Dissertation (Ph.D.), In Computer Science, Friedrich Alexander Universitaet Erlangen-Nuremberg, Erlangen, Germany.

Sahner, R., and Trivedi, K.S. (1986), "Reliability Modeling Using SHARPE," TR-1986-19, Computer Science, Duke University, Durham, NC.

Sahner, R. A., and Trivedi, Kishor S. (1993), "A Software Tool for Learning About Stochastic Models," *IEEE Transactions on Education*, *36*, 1, 56-61.

Sahner, R. T., K. and Puliafito, A. (1996), *Performance and Reliability Analysis of Computer Systems - An Example-Based Approach Using the SHARPE Software Package*, Kluwer Academic, Boston, MA.

Sanders, W., Obal, W., Qureshi, A. and Widjanarko, F. (1995), "The UltraSAN Modeling Environment," *Performance Evaluation*, *24*, 1, 89-115.

Sanders, W. H. a. M., J.F. (1991), "Reduced Base Model Construction Methods for Stochastic Activity Networks," *IEEE Journal on Selected Areas in Communications*, *9*, 1, 25-36.

Schweitzer, P. (1991), "A Survey of Aggregation-Disaggregation in Large Markov Chains," In *Numerical Solution of Markov Chains*, W. Stewart, Marcel Dekker, New York, NY, pp. 150-230.

Sheldon, F. T., Kavi, K.M., and Kamangar, F.A. (1995), "Reliability Analysis of CSP Specifications: A New Method Using Petri Nets," *Computing in Aerospace 10*, AIAA, Reston, VA, pp. 317-326.

Sheldon, F. T., and Kavi, K.M. (1995a), "Linking Software Failure Behavior to Specification Characteristics II," *4th Int'l Wkshp on Evaluation Techniques for Dependable Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. .

Sheldon, F. T. (1996b), "Specification and Analysis of Stochastic Properties for Concurrent Systems Expressed Using CSP," Ph.D. Dissertation, In , UMI and CSE Dept., Univ. of TX at Arlington, Ann Arbor, MI.

Sheldon, F. T. (1998), "Analysis of Real-Time Concurrent System Models Based on CSP Using Stochastic Petri Nets," *12th European Simulation Multi-conference*, SCS Int'l, Amsterdam, pp. 776-783.

Siegle, M. (1995), "Bescgreibung und Analyse von Makovmodellen mit Grossem Zustandsraum," Ph.D. Dissertation, In Computer Science IMMD IV, Univ. Erlangen-Nurnberg, Germany.

Sommerville, I. (1996), *Software Engineering*, Fifth Edition, Addison-Wesley, Reading, MA.

Stewart, W. J. (1994), *Introduction to the Numerical Solution of Markov Chains*, Princeton University Press, Princeton, NJ.

Stiegler, H. (1988), "System Overview BS2000 -- Technical Description," 15-1988, Siemens Corp., Munch, Germany.

Trivedi, K., Malhotra, Manish (1993), "Reliability and Performability Techniques and Tools: A Survey," *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, Springer-Verlag, Berlin, pp. 27-48.

Trivedi, K. S. (1982), *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, Prentice Hall, Englewood Cliffs, NJ.

Villemeur, A. (1992), *Reliability, Availability, Maintainability and Safety Assessment*, John Wiley and Sons, New York, NY.

Yen, I.-L., Paul, Raymond, and Mori, Kinji (1998), "Toward Integrated Methods for High-Assurance Systems," *IEEE Computer*, *31*, 4, 32-34 (including pp. 35-46 by others).