A TECHNIQUE FOR SIMULATING COMPOSED SAN-BASED REWARD MODELS

by

Roberto Freire

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfilment of the Requirements
For the Degree of

MASTER OF SCIENCE

WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1990

STATEMENT BY AUTHOR

SIGNED: ____________________

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

____________________ ____________________

William H. Sanders
Assistant Professor of
Electrical and Computer Engineering

Date

To Paula and my parents

ACKNOWLEDGMENTS

Working on my Master's thesis was a very challenging and satisfying experience that was made possible by the advice and support of a number of people. First of all, I am thankful to Dr. Wiliam H. Sanders for constantly inspiring me to bring out the best in myself. I am also grateful to the master's committee members, Dr. Sy-Yen Kuo and Dr. Jerzy Rozenblit, for reviewing my thesis.

I would like to acknowledge Intel Corporation, Digital Equipment Corporation and Bell Communications Research. Their financial support made this work possible.

I am also thankful to Manish Rai for his help and encouragement.

Finally, I would like to thank my family, friends and colleagues for their moral support in bringing all my work together.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Stochastic activity networks (SANs) have been used in the modeling of computer systems because of their suitability in representing distributed systems. SANs may be solved by analysis or by simulation. When simulation is used, future events list management is very time consuming, as with other simulation techniques. New methods that take advantage of the SAN model structure are presented which significantly reduce the cost of future events list management. Multiple future events lists are used to reduce operations required upon each state change.

CHAPTER 1

Introduction

1.1 System Evaluation

System evaluation techniques can be broadly classified into two categories: measurement and modeling. Measurement requires that a real system be built, tested and functioning. The advantage of this method is the accuracy of the results, since data is collected from the actual system. However, its use in evaluating large and complex systems is extremely difficult and costly.

Modeling is the alternative to avoid measurement drawbacks. Instead of experimenting with the real system, a model is built and characteristics are estimated from the model. When specifying a model, assumptions are made about the system. Although the actual system is not evaluated, if the assumptions are made correctly, the results obtained by use of the model yields correct answers.

Modeling has several advantages over measurement. First, a model can be constructed at any phase of the system life-cycle. Second, it is typically much less costly.

Models can be solved by analysis or by simulation. If it is simple enough such that mathematical solution is possible, analysis should be the choice. Unfortunately, in general, that is not the case. Models that cannot be solved by analysis must be solved by simulation.

1.1.1 Modeling Techniques

Modeling techniques can be classified as: deterministic or non-deterministic. Turing machines are an example of deterministic models. A non-deterministic model can either be probabilistic or non-probabilistic. Petri nets fall in the latter category. This technique is useful when modeling concurrent, distributed and parallel systems. However, their non-probabilistic nature makes them unsuitable for performance/dependability evaluation. Probabilistic models can better represent real systems for performance evaluation purposes. These include: failure, repair and service rate models, reliability graphs, fault trees, queueing networks and stochastic extensions to Petri nets [7]. While the first three are convenient for modeling system inputs, the remaining ones are used to model system behavior. Reliability graphs and are also categorized as combinatorial models. These enumerate all the combinations of failed and working elements to represent either a success or a failure of a system. These models suffer from the problem of state space explosion and become prohibitive if modeling is to be done under a certain level of detail.

Queueing network theory is well known and widely used but are limited in modeling power. Stochastic extensions to Petri nets provide much of the power lacking in queueing networks.

1.1.2 Stochastic Extensions to Petri nets

Petri net theory was introduced by Carl Adams Petri [17]. Petri nets were developed for modeling systems graphically and mathematically. As systems grew in complexity, a need for extensions to Petri nets was realized. Due to there non-probabilistic nature, standard Petri nets are not suitable for performance/dependability evaluation [16].

First efforts were to add probabilistic delays to the transitions of the net [12, 15, 24]. This extension enabled the modeling of complex concurrent systems, but there was need for further extensions. Next, delays other than exponential were added followed by the creation of transitions with zero time delays. Finally, the development of additional constructs to ease in the representation of systems without changing the probabilistic behavior of the net.

Transitions with zero delays were introduced in "Generalized Stochastic Petri Nets" (GSPNs) by Marsan, Balbo, and Conte [11]. To determine the probability of the firing of an immediate transition (i.e. zero time delay), a *random switch* was used. States in which some time was spent were called *tangible*. Otherwise, the state was termed *vanishing*. A model construction technique was proposed to eliminate vanishing states reducing the state space.

"Extended Stochastic Petri Nets" are another generalization that allowed transition delays to have arbitrary distribution functions [4]. *Probabilistic arcs* were introduced to create uncertainty about state trajectories.

A third independent extension proposed was "Stochastic Activity Networks" [13]. *Instantaneous activities* represent events that take negligible amount of time and *timed activities* represent probabilistic delays. *Cases* for activities create uncertainty about state trajectories. For convenience in representing large systems, input gates and output gates between activities and places were introduced. Several evaluation tools based on these extensions to Petri nets have been developed.

1.1.3 Evaluation Packages Based on Stochastic Extensions to Petri Nets

Although there are several tools based on stochastic extensions to Petri nets, only a few support solution by simulation. "GreatSPN", developed by Chiola [2], allows solutions both by simulation as and analysis. Based on GSPNs, it allows formal validation of quantitative behavior of the underlying Petri nets. The simulator generates confidence intervals using Monte Carlo simulation.

METASAN [22] is based on stochastic activity networks. It supports evaluation either by simulation or by analysis. Four analytic solvers are provided. Terminating and steady state simulation is supported for estimation of means, variances, percentiles and intervals.

The most recent tool available is SPNP [3]. Only analytical solutions are supported. Reward model specification at the net level is supported as in [20].An automated sensitivity analyzer is provided.

Other tools based on stochastic Petri nets include Simula by Torn [25], FORCASD based on "evaluation nets" by Behr [1], FUN [5] and DEAMON [10] based on "function nets".

1.2 Research Objectives

While there exists simulators based on stochastic activity networks, little work has been done to improve solution times. Composed SAN-based reward models [20] exhibit structural properties that can be used to improve simulation times of highly replicated systems such as distributed systems. The goal of this study is to develop a tool, based on composed SAN-based reward models, for solution of models by simulation. This simulator should take advantage of the structure of the composed model to reduce solution times.

More specifically, we will:

a) Provide an efficient technique to generate possible state trajectories of a composed SAN-based reward model.

b) Develop both a steady state and a terminating composed SAN-based reward model simulators.

c) Illustrate the usefulness of the new technique in simulating a highly replicated system.

Chapter 2 provides an overview of SANs. Basic SAN definitions are given followed by the composed SAN-based reward model construction methodology. An example of a composed model is given.

Chapter 3 is presents the procedures that are specific to composed SAN-based reward model state generation. An example state generation is given at the end of the chapter.

Chapter 4 explains the algorithms and calculations performed by both terminating and steady state simulators. The procedures presented in Chapter 3 are encapsulated in the main simulator loops.

Chapter 5 presents an example study to illustrate the usefulness of the techniques introduced in this thesis. A 100 CSMA/CD local area network is simulated for several variables in steady state and transient phase.

Chapter 6 summarizes the results of this thesis and suggests areas of further research.

CHAPTER 2

Stochastic Activity Networks

2.1 Overview

Stochastic activity networks originated from Petri net theory [14, 16]. A Petri net is a network of *places* connected to *transitions* by directed *arcs*. A place connected to a transition is an *input place* or an *output place* to the transition, depending on the direction of the arc, which determines the flow of *tokens* held by the places. The marking of a Petri net is a vector of positive integers representing the number of tokens in each place. An arc can have a *weight*, a positive integer. A transition *fires* removing a number of tokens from its input places and depositing a number of tokens on the output places equal to weights of the corresponding arcs. It may fire when it is *enabled* in a marking, meaning that there are at least as many tokens in each of its input places as the weights for the corresponding arcs.

Figure 2.1 is an example of a Petri net. Transition t_1 is enabled in this marking because its input place p_1 holds three tokens (represented by the dots) which is more than two, the weight of the connecting arc. Hence, it may fire removing two tokens from p_1 and depositing three tokens in output place p_2 and one token in output place p_3.

The non-determinism in Petri nets is not quantified probabilistically. For instance, in Figure 2.1, activities t_1 and t_3 are enabled. It is not clear which transition should fire first. Also, delays before a transition fires cannot be modeled. Although it is a powerful modeling technique for verifying certain system properties, performance and dependability evaluation is not possible.

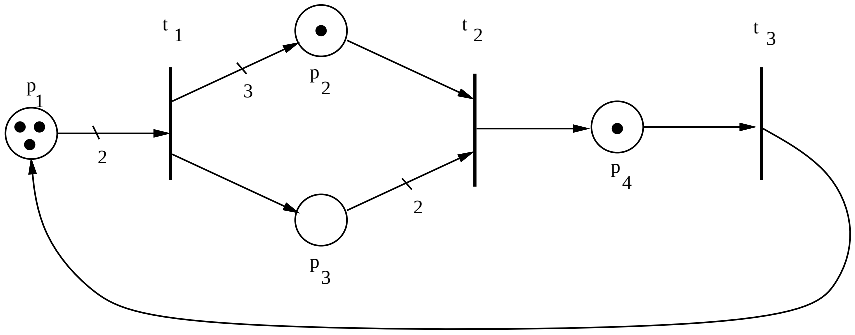


Figure 2.1: Petri Net Example

A stochastic activity network (SAN) [13, 20] is a stochastic extension to a Petri net. Such as in Petri nets, SANs have places and tokens. Petri Net transitions find equivalents in activities, which can be *timed* or *instantaneous*. Timed activities have probability distributions associated with them to represent delays. While transitions fire, activities *complete*. Instantaneous activities complete as soon as they are enabled. Activities may have *cases*, each case associated with a probability. When an activity completes, a case is chosen according to the probability distribution defined for the cases. Unlike Petri nets, SANs have *input gates* and *output gates* connected to activities. An input gate connects an input place to an activity. It has a predicate function and an output function associated with it. An output gate connects an activity to an output and only has an output function. An activity is enabled in a marking μ if all the input predicates are true. An enabled activity completes and executes both the input and output gates' output functions. Note that an enabled timed activity that would have completed after a certain delay is *aborted* if, during this period, the SAN reaches a "stable marking" where the activity is no longer enabled. A *stable marking* is a marking such that no instantaneous activities are enabled. The behavior of a SAN is characterized by a sequence of markings and activity completions.

Finally, a *reactivation function* is specified for each timed activity. This function relates each marking to a set of *reactivation markings* for an activity. Whenever an activity is *activated*, i.e. becomes enabled, it will be reactivated, i.e. aborted and immediately reactivated, if a marking

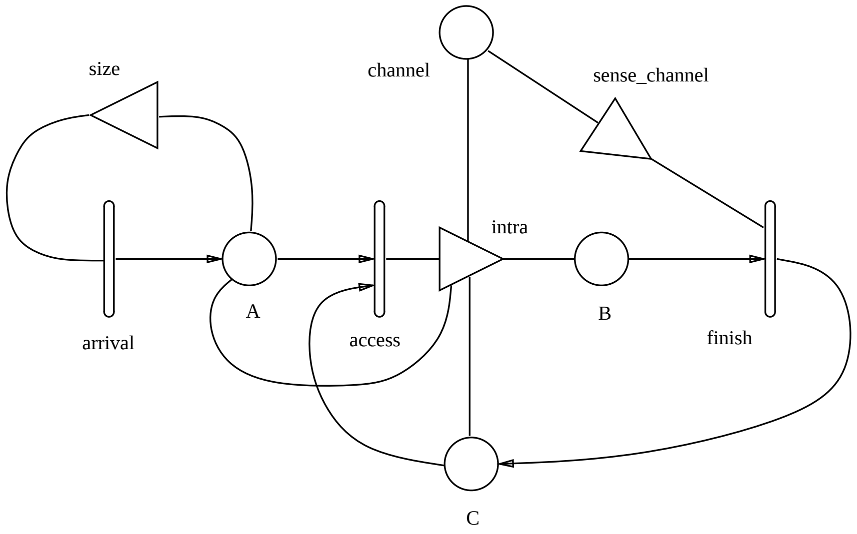


Figure 2.2: Station Submodel

from its set of reactivation markings is reached before it completes, given that it was activated in a specific marking. The reactivation function is useful when representing a process that should abort and start over given certain conditions of the system are met.

In order to be solved by simulation, SANs must be “well specified” [20]. Informally, a SAN is *well specified* if, given that more than one instantaneous activity is enabled in a marking, the probabilities of reaching the possible next stable markings are the same no matter which instantaneous activity completes.

As an example of a SAN, take Figure 2.2, Table 2.1 and Table 2.2. They specify a SAN model of a station for a CSMA/CD local area network. The model comprises of a graphical representation, depicting the structure, and tables that specify functions for the gates and time distributions for the timed activities. Timed activity *arrival* is enabled in the present marking because the input predicate of the input gate $size$, $MARK(A) < 2$, is satisfied. If it completes, the output function for $size$, *identity* (i.e., do nothing), will be executed. Then the output function for each output gate of *arrival* would have been executed, if some existed. Finally, the number of tokens of the

Gate	Type	Enabling Predicate	Function
size	input	$MARK(A) < 2$	identity

intra	output	–	if (MARK(*channel*)==0) {
			MARK(*channel*) = 1;
			MARK(*B*) = 1; }
			else if (MARK(*channel*)==1) {
			MARK(*channel*) = 3;
			MARK(*C*) = 1;
			MARK(*A* = MARK(*A*) + 1; }
			else if (MARK(*channel*)==2) {
			MARK(*C*) = 1;
			MARK(*A*) = MARK(*A*) + 1; }
			else if (MARK(*channel*)==3) {
			MARK(*C*) = 1;
			MARK(*A*) = MARK(*A*) + 1; }

sense_channel	input	MARK(*channel*)==2) || MARK(*channel*)==3	MARK(*channel*) = 0;

Table 2.1: Gates for Station Submodel

Activity	Distribution Type	Parameter (Rate)
arrival	exponential	varied
access	exponential	10 (normal prio. station)

finish	exponential	if (MARK(*channel*)==2)
		rate = 1
		else
		rate = 5

Table 2.2: Activity Parameters for Station Submodel

output place A will be incremented by one, the number of incoming arcs. Arcs, in SANs, act the same way as in Petri nets. Now, both *arrival* and *access* are enabled.

In order to aid in the understanding of the following chapters, it is helpful to define, more formally, several concepts related to SANs. In particular:

1. If P is the set of all places in a stochastic activity network, then the *marking of the network* is a mapping $\mu : P \rightarrow \mathbb{N}$.
2. Similarly, if S is a set of places ($S \subseteq P$), the *marking of* S is a mapping $\mu_S : S \rightarrow \mathbb{N}$. We say that μ_S is a *restriction* on the marking μ to the places in S.
3. Markings can also be represented as a vector, as in Petri nets, where each of its component is the number of tokens in a particular place. The correspondence between the components and the places is achieved by designating each component of the vector to a place of the set of places considered. This is usually done by assuming the usual lexicographical ordering based on place names. Then, the marking of a place $p \in P$, $\mu(p)$, is n, a positive integer component of μ.
4. For a marking μ, the set of enabled activities in μ is denoted as En_μ.
5. For an active activity a there is a set $React_a$ of reactivation markings. This is the notation used in the procedures described in the next chapter.

2.2 Performance Variables

The purpose of a modeling technique is to provide methods that allow solutions to questions about a system. To solve these questions, it should be possible to conveniently describe the variables that represent these questions. Traditionally, performance variables for stochastic Petri nets have been specified at the state level. This can be very cumbersome, considering the high degree of complexity of present computer systems. SAN-based reward models have variables that are specified at the network level, which is more natural to the modeler [21].

The variables that have been used with SANs are specified in terms of *reward structures* [6]. Informally, a reward structure associates behaviors of the system with performance variables. This is done by specifying "bonuses" (or "penalties") for certain events that may occur in the system and reward rates for the system being in some "state". For SAN modeling, we associate a bonus with an activity completion and a rate reward with a particular marking of the SAN. Since activity completions occur at instants of time, we call these "bonuses" impulse rewards. An impulse reward with a rate reward form a "reward structure". Formally, an *activity-marking reward structure* of a stochastic activity network with places P and activities A is a pair of functions [21]:

$\mathcal{C} : A \to I\!R$ where for $a \in A$, $\mathcal{C}(a)$ is the reward obtained due to completion of activity a, and

$\mathcal{R} : \mathcal{P}(P, I\!N) \to I\!R$ where for $\nu \in \mathcal{P}(P, I\!N)$, $\mathcal{R}(\nu)$ is the rate of reward obtained when for each $(p, n) \in \nu$, there are n tokens in place p,

where $I\!N$ is the set of natural numbers and $\mathcal{P}(P, I\!N)$ is the set of all partial functions between P and $I\!N$. Elements of the set $\mathcal{P}(P, I\!N)$ can be considered as *partial markings.* They are the results of the restriction of the function μ on a subset of places of P.

Having defined a reward structure, we now present different categories of variables that can be estimated by simulation.

2.2.1 Instant-of-Time Variables

A variable that measures the behavior of stochastic activity network at a particular time t is the instant-of-time variable V_t. It is defined as:

$$V_t = \sum_{\nu \in \mathcal{P}(P, I\!N)} \mathcal{R}(\nu) \cdot I_t^{\nu} \; + \; \sum_{a \in A} \mathcal{C}(a) \cdot I_t^{a},$$

where

I_t^ν is an indicator random variable representing the event that the SAN is in a marking such that for each $(p, n) \in \nu$, there are n tokens in p at time t, and

I_t^a is an indicator random variable representing the event that activity a is the activity that completed most recently at time t.

If the indicator random variables converge in distribution as $t \to \infty$, V_t is considered in "steady state". It is then possible to evaluate the steady state reward at an instant of time t with the variable defined as

$$V_{t\to\infty} = \sum_{\nu \in \mathcal{P}(P, \mathbb{N})} \mathcal{R}(\nu) \cdot I_{t\to\infty}^\nu + \sum_{a \in A} \mathcal{C}(a) \cdot I_{t\to\infty}^a,$$

where

$I_{t\to\infty}^\nu$ is an indicator random variable representing the event that the SAN is in a marking such that for each $(p, n) \in \nu$, there are n tokens in p in steady-state, and

$I_{t\to\infty}^a$ is an indicator random variable representing the event that activity a is the activity that completed most recently in steady-state.

Characteristics such as queue length can be represented by these variables.

2.2.2 Interval-of-Time Variable

Interval-of-time variables are used to quantify accumulated benefits (or penalties) during some interval of time. Although there are other variables defined in this category for solution by analysis [20], we have chosen one to implement for solution by simulation. In this case, the variable accumulates reward during an interval of time, thus expressing the total reward for that period. We denote this variable by Y_t and define it as:

$$Y_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(P,I\!N)} \mathcal{R}(\nu) \cdot J^{\nu}_{[t,t+l]} \; + \; \sum_{a \in A} \mathcal{C}(a) \cdot N^{a}_{[t,t+l]}, \text{ and}$$

where

$J^{\nu}_{[t,t+l]}$ is a random variable representing the total time that the SAN is in a marking such that for each $(p,n) \in \nu$, there are n tokens in p during $[t, t+l]$, and

$N^{a}_{[t,t+l]}$ is a random variable representing the number of completions of activity a during $[t, t+l]$.

2.2.3 Time-Averaged Interval-of-Time Variable

The final category of reward variables, the time-averaged interval-of-time variables, quantifies accumulated benefits (or penalties) averaged during some interval of time. As in the interval of time category, there are several types of variables of this type for use in solution by analysis. We shall present the one we have implemented for simulation. This variable is defined as:

$$W_{[t,t+l]} = \frac{Y_{[t,t+l]}}{l}$$

where

$J^{\nu}_{[t,t+l]}$ is a random variable representing the total time that the SAN is in a marking such that for each $(p,n) \in \nu$, there are n tokens in p during $[t, t+l]$, and

$N^{a}_{[t,t+l]}$ is a random variable representing the number of completions of activity a during $[t, t+l]$.

The throughput of a system can be estimated using a variable of this type.

2.2.4 Time Between Completions Variable

While the reward variables described can be used to represent many measures of performance and dependability, one useful variable that cannot be quantified is one that represents the time between completions of activities. Since this is an important measure, and can be estimated using simulation, we define a variable of this type. In particular, let $TB^a_{i,j}$ $(i < j)$ be the time between the i^{th} and j^{th} completion of activity a. Times between arrivals of jobs to a computer system can be estimated by a variable of this type.

If $TB^a_{i,j}$ converges in distribution, we can define a variable that amounts to the time between completions of an activity a in steady state. We define this variable as:

$$TB_a = \lim_{i \to \infty} TB^a_{i-1,i}$$

2.3 Composed SAN-Based Reward Model Construction

Now that the model representation (SANs) and performance variables have been specified, we can define the structure on which the simulator will operate, a "composed SAN-based reward model (SBRM)". A composed *SBRM* is obtained by the use of the *replicate* and *join* operations on a SAN-based reward model. A SBRM is defined in [20] as a SAN, S, with places, P, together with a reward structure $(\mathcal{C},\mathcal{R})$ and set of distinguished places $P_D \subseteq P$. Formally it is a four-tuple $(S, \mathcal{C}, \mathcal{R}, P_D)$.

The *replicate* operation replicates a SAN-based reward model a certain number of times, holding some subset of its places (the "distinguished places") common. It is through these unreplicated places that the replicated submodels interact. Each replicate will have values for impulse reward and reward rates specified in the same way as they were in the original model but now they are are assigned respectively to the corresponding replicate activities and partial markings.

The result of the replicate operation is also a SAN-based Reward Model. Formally, as in [23], it is a SBRM $(\tilde{S}, \tilde{\mathcal{C}}, \tilde{\mathcal{R}}, \tilde{P}_D)$ with places $\tilde{P}$ and activities $\tilde{A}$, where :

1. $\tilde{S}$ is a SAN constructed by replicating S n times. $\tilde{P}_D \subseteq P_D$ is the set of places to be used in the replicate operation. All activities and gates are replicated; all places except those in $\tilde{P}_D$

are replicated. Connections to each $p \in \tilde{P}_D$ are identical for each replicated submodel in $\tilde{S}$ and as in S.

2. For each $a \in A$ and corresponding set of replicate activities $a_1, a_2, \ldots, a_n \subseteq \tilde{A}$, $\tilde{\mathcal{C}}(a_i) = \mathcal{C}(a)$ for each replicate submodel i.

3. For each $\nu \in \mathcal{P}(P, I\!N)$ and corresponding partial markings $\nu_i \in \mathcal{P}(\tilde{P}, I\!N)$, $\tilde{\mathcal{R}}(\nu_i) = \mathcal{R}(\nu)$ for each replicate submodel i. $\tilde{\mathcal{R}}(\nu) = 0$ for all other $\nu \in \mathcal{P}(\tilde{P}, I\!N)$.

4. $\tilde{P}_D \subseteq P_D$ is the set of places not replicated.

As the name of the operation suggests, the replicate operation is suitable for representing models that consist of several identical submodels. An operation that allows us to combine models of different structure is needed. The join operation can be used for this purpose.

The join operation acts upon individual SAN-based reward models combining them producing another SBRM. It uses a list of places associated with each component SAN-based reward model, where each list of places is a subset of the set of distinguished places in the constituent SBRMs. As with replication, the component submodels interact with each other through these places. In particular, the first place in each of the lists is merged to form a single place in the new submodel, the second place in each of the lists is merged forming another single place in the new submodel, and so on. Note that, unlike in [23], we allow particular elements on the lists of places to be null, allowing the case where certain places are created from places in a subset of the SBRMs that are joined. This extension to [23] presents no problem with either solvability or support, but does introduce more flexibility in defining SAN-based reward models.

The reward model for the resulting SBRM is obtained in a similar manner to that for a replication operation. Each activity in the new model has assigned to it an impulse reward equal to that in the corresponding activity in the original model and each partial marking that had a

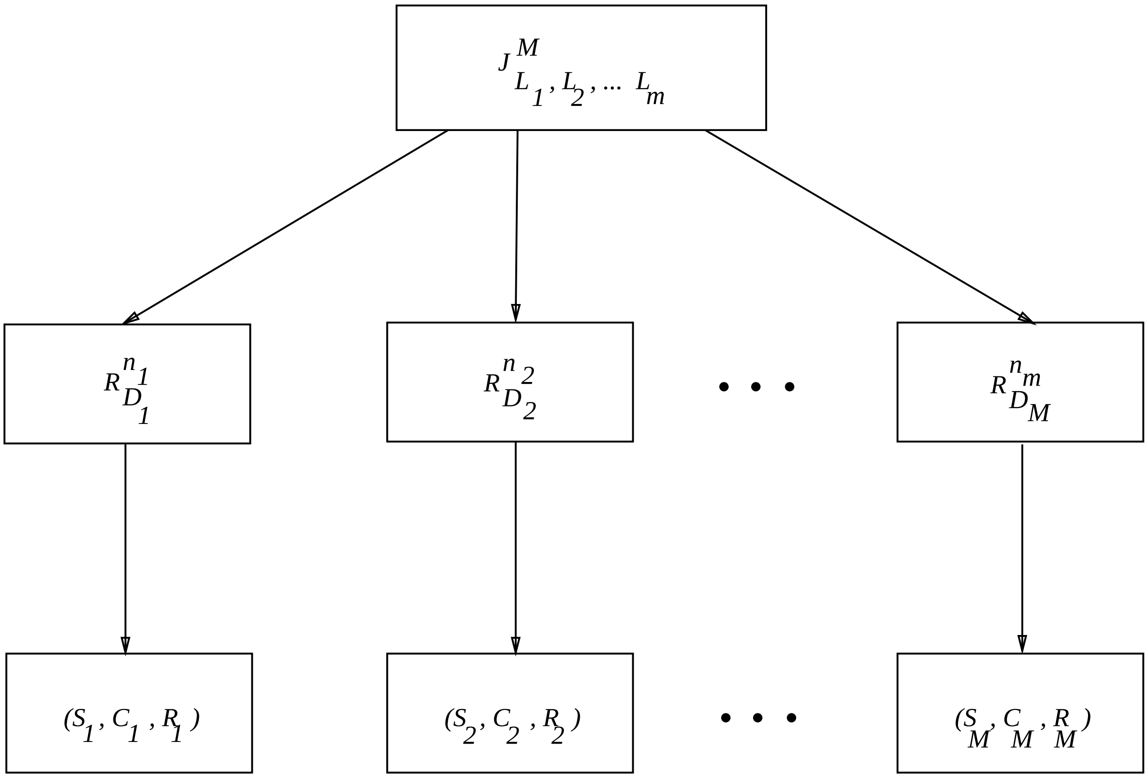
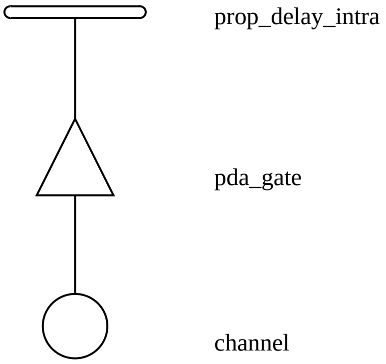


Figure 2.3: Example Composed SAN-Based Reward Model

reward rate associated with it will have a corresponding partial marking with an equal reward rate assigned to it.

The composed SBRM is built using these operations, starting with individual SAN-based reward models at the bottom. Graphically, we can represent the composed model as a directed tree. To illustrate this idea, consider as an example Figure 2.3.

There are three types of nodes: *model/reward structure nodes*, *replicate nodes*, and *join nodes*. The model/reward nodes represent individual submodels, defining a distinct SAN S with impulse reward C and rate reward R. They will either be replicated or joined to another SBRM depending on the operation situated at the parent node. Replicate nodes have information regarding to the number of times its child is to be replicated (represented by the integers $n_1, n_2, ..., n_m \in \mathbb{N}$) and the set of distinguished places from the set of places of the child submodel (represented by



Gate	**Type**	**Enabling Predicate**	**Function**
pda_gate	input	MARK(*channel*)=1	MARK(*channel*)=2;

Activity	**Distribution Type**	**Parameter (Rate)**
prop_delay_intra	exponential	20

Figure 2.4: Network Submodel

$D_1, D_2, ..., D_m$). Join nodes have information regarding to the number of SBRMs to be joined (given as the integer M) and lists of distinguished places for each of these SBRMs (given as $L_1, L_2, ..., L_m$). The arcs tell us which SBRMs are used by an operation. Note that join nodes have a degree equal to the number of SBRMs they operate upon. Replicate nodes always have a degree of one since they act on a single SBRM. Although not necessary, for convenience in computer programming, we create the restriction that only timed activities can be connected to places in $\tilde{P}_D$.

To illustrate the composition on a more realistic application, consider a CSMA/CD local area network with a certain number n of stations connected to it. The SAN model for a station was presented earlier (Figure 2.2), and the model for the network is in Figure 2.4.

Figure 2.5 shows how the tree for the composed model would look like if we assume all stations are identical. Nodes with no outgoing arcs are the individual submodels for a station and the network. The station submodel is replicated n times with a place called "channel" from the set of distinguished places of $S_{NETWORK}$ held as common to the replicates. The produced SBRM is

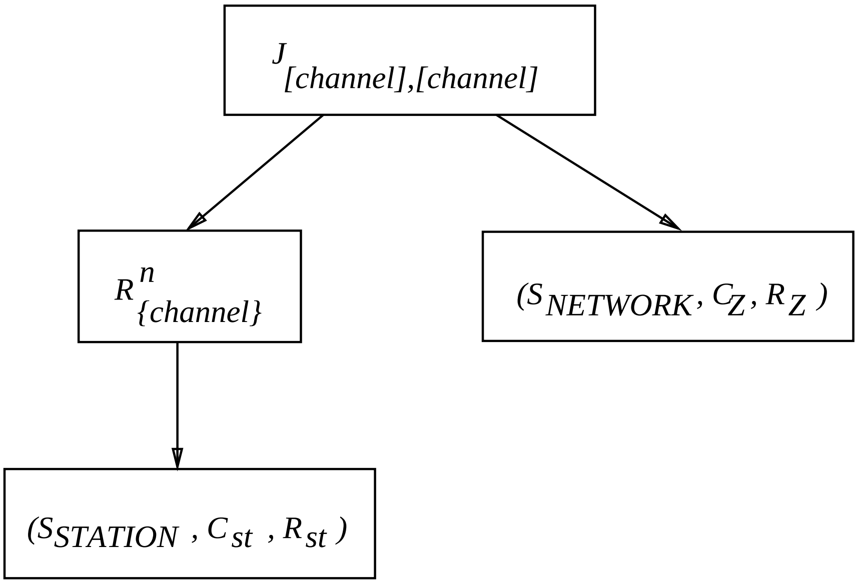


Figure 2.5: Example Composed SBRM for CSMA/CD LAN

then joined to the network submodel by two places, both named “channel”, from the respective set of distinguished places for each component submodel. This model is actually solved via simulation and explained in Chapter 7.

CHAPTER 3

State Change Mechanism and Future Events List Management

3.1 Introduction

The previous chapter presented a review of the concepts of stochastic activity networks and how a composed SAN-based reward model can be constructed. Once the model is available, it can be solved by simulation. An important aspect of simulation is the mechanism for changing states and managing the list of future events. Efficiency in these mechanisms is essential for simulation of large models. A method introduced in Sanders [20] has been used for simulation of distributed systems modeled by SANs (METASAN[1] [22]). While this method has been used to evaluate many computer systems and networks, it suffers, like many simulation techniques, from long run times for large models. In this chapter, we make use of the composed model structure to develop a new simulation engine for SAN-based reward models. The method makes use of the idea of "compound events" and the structure of the SBRM to improve the efficiency of the state change and future events list management algorithms.

---

[1] METASAN is a registered trademark of Industrial Technology Institute

3.2 Traditional Method

In this method, which is described in [20], each activity of the model is an *event type.* Every activity completion is an *event.* During execution, at each event, the model will behave according to its event type.

Future events are scheduled based on *potential completion times* for the enabled activities. The term "potential" refers to the possibility of completion of an activity that has been activated. The procedure **GenerateEvent** generates a potential completion time for an activity enabled in a current marking.

State trajectories are generated by completing the earliest activity scheduled to complete. Procedure **Earliest** determines this activity, cur_a, given the list of future events, E. Informally, at every activity completion, a new marking, μ', for the model is chosen probabilistically from the set of next stable markings generated by **ExecuteSAN**. This procedure executes a SAN submodel repeatedly until all possible next stable markings are reached and the probabilities of reaching them are determined given a particular stable marking and an activity completion. It is then necessary to check all scheduled events to see if they are still enabled in the current marking and remove the ones that are not. Also, if an activity should be reactivated, the corresponding event should be rescheduled. Finally, since there could be activities other than the ones that were scheduled that should be activated in the new marking, all activities in this condition are added to the future events list.

The algorithm translated, from Sanders [20], is as follows:

Procedure 3.2.1 GenerateBehavior

```
Begin
    $E = \emptyset$
    $\mu =$ INITIAL MARKING
    for each $a \in En_\mu$
        $e_a =$ GenerateEvent($a, \mu$)
        $E = E \cup \{e_a\}$
    while $E \neq \emptyset$
```

```
        $cur_a$ = Earliest(E)
        $E = E - \{e_{cur_a}\}$
        $\mu'$ = ExecuteSAN($cur_a$,$\mu$)
        for each $e_a \in E$
            if $a \notin En_{\mu'}$
                $E = E - \{e_a\}$
        for each $e_a \in E$
            if $\mu \in React_a$
                $E = E - \{e_a\}$
        for each $a \notin En_\mu$
            if $a \in En_{\mu'}$
                $e_a$ = GenerateEvent($a$, $\mu'$)
                $E = E \cup \{e_a\}$
End.
```

We are particularly interested in the way future events list management is done in this procedure. This is generally where most of the computation time for a simulation run is spent. The procedure suggests keeping a list of events that represent potential completion times, i.e. the times for enabled activities to complete if they are not aborted prior to completion. Note that there is a one to one relationship between the set of enabled activities in the composed model and the set of future events. Thus, all active activities in μ, replicates or not, will have to be checked for their status in μ' (enabled or disabled). Furthermore, all active activities must be checked for reactivation at each state change. Finally, the loop ends with another check on all activities to find the ones that were not enabled in μ but enabled in μ'. As the composed model grows in size, the computation time spent on these operations can make the solution by simulation infeasible.

3.3 Making Use of the Composed Model Structure

In this section, we develop procedures that make use of the composed SAN-based reward model structure to reduce simulation time. The procedures are introduced after some new concepts are presented. To illustrate the procedures, an example state generation is provided at the end of this section.

3.3.1 State Representation

A definition of "state" is needed to make possible the description of a composed model execution procedure. Recall that a composed SBRM is a result of operations on SBRMs that themselves may be also a composed model or just individual SBRM submodels. We can represent this composition graphically by a tree, as described in Chapter 2. A notion of "state" can be determined at each level of the tree structure. At a join operation, we keep a vector of "states" for each joined submodel. The number of replicates in each existing submodel "state" is kept at a replication node representing the operation. Finally, at the lowest level, the "state" of a SAN model is its marking. The complete state is then the impulse reward due to the last activity completion and the completion and the composed state formed as above.

A convenient way of describing the state of a composed SBRM is to use equations that relate SBRMs at on level to those at the next lower level based on the operation done at that level. Each replication operation is represented as a bag [16] of states of the replicate submodels. Join operations are represented by a vector of states joined submodels. At the lowest level of operation, the elements of the vectors or bags are the projected markings of the complete marking on places of individual SAN submodels.

For example, consider the SAN-based reward model of Figure 3.1. The composed state for that composed SBRM can be represented as the following:

$$\begin{aligned}
State &= (\mathcal{C}(a), V) \\
V &= (B_1, \mu^2) \\
B_1 &= < V_{11}, V_{12} > \\
V_{11} &= (B_{111}, B_{112}) \\
V_{12} &= (B_{121}, B_{122}) \\
B_{111} &= < \mu^{1111}, \mu^{1112} >
\end{aligned}$$

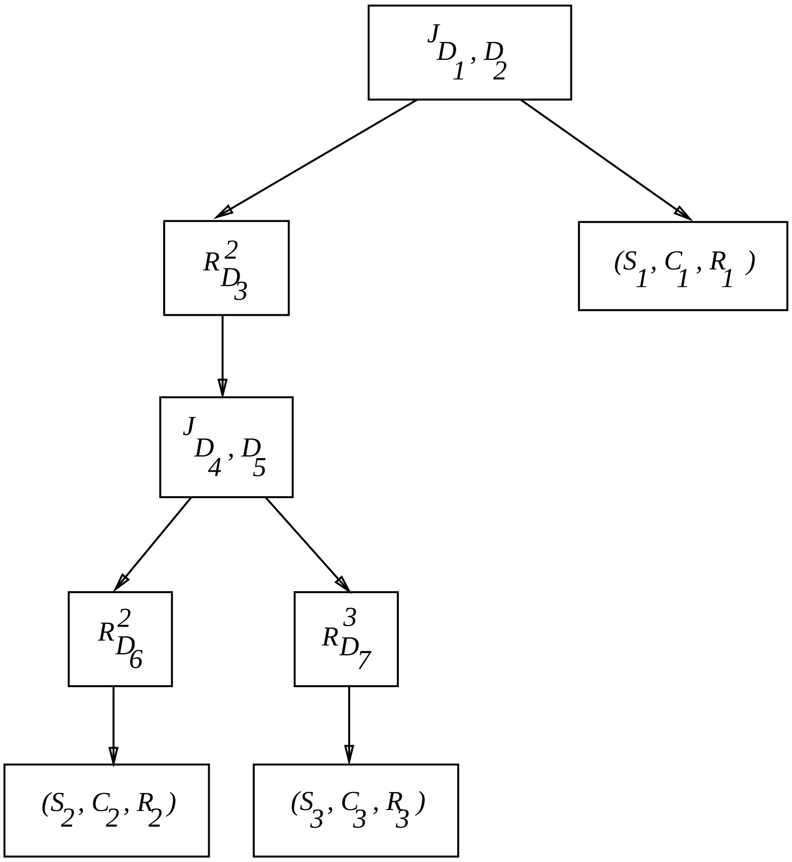


Figure 3.1: Example Composed SAN-based Reward Model

$$\begin{aligned} B_{112} &= < \mu^{1121}, \mu^{1122}, \mu^{1123} > \\ B_{121} &= < \mu^{1211}, \mu^{1212} > \\ B_{122} &= < \mu^{1221}, \mu^{1212}, \mu^{1223} > . \end{aligned}$$

Here, the letter B denotes a bag at the next lower level and the letter V denotes a vector. Bags are useful to represent replicate operations since the number of replicate submodels in the same state can be found by the $\#(x, B)$ (number of occurrences of element x in B) operation [16]. The superscripts designate projections of the global marking μ on the places of particular submodels. For instance, μ^{1123} denotes the projection of the global marking on the places of the first SBRM of the highest level of operation, the first SBRM of the operation at the next level, the second SBRM of the operation at the third level below the highest, and the third SBRM from the operation at the fourth level below the highest.

3.3.2 State Trees

While the notation used in the previous section is convenient for formally describing the state of a composed SAN-based reward model, it is not convenient for describing the mechanisms of the state change and future events list management. We do this using a *state tree.*

State trees have three types of nodes: *join nodes*, *replicate nodes*, and *SAN nodes.* All leaves of a state tree are of type SAN. Nodes that are not leaves have types join or replicate. A node of type SAN, in particular, also has a *sub-type* which relates the node to an individual SAN model. The type of a node i is referred to as $type_i$. In the discussion that follows, we denote the levels of the node by their distance from the root node, whose level is 0.

The state tree is directly related to the description of state given in the previous section. A vector of markings is represented by a join node. A replicate node represents a bag. A SAN node represents a SAN in a particular marking.

Every node on the state tree has a corresponding node in the composed model diagram. A node on a particular level on a state tree corresponds in type and level with a node on the diagram. On both the state tree and the composed model diagram, nodes related to SANs are at the lowest levels on the tree. Furthermore, each state tree node has associated with it a subset of the distinguished places of the corresponding node in the composed model diagram. These are the places that are distinguished at the node, but not to its parent node. For convenience, we use the expression "a place at node i" to characterize this relationship between nodes in the state trees and places in the composed SAN-based reward model.

Given this assignment of places to node in the state tree, we define μ_i as the restriction of the global marking to the places at node i. The marking μ_i appears next to a node i and is ordered according to the alphabetical order of the places at that node.

Nodes are connected by directed arcs. An arc that connects a parent node i to a node j has an associated integer $n_{i,j}$, where $n_{i,j}$ is the number of occurrences of the marking of the SBRM represented by a node j in the bag corresponding to i. By definition, each outgoing arc j from a join node i has $n_{i,j}$ equal to one, since there is one copy of each constituent SBRM used in the join operation. Finally, for every node i representing an operation, we denote its set of of children nodes as C_i.

To illustrate the use of state trees, consider once again Figure 3.1. In this figure, the letters "D_i" denote sets of distinguished places at certain levels in the composed model and let P_1, P_2 and P_3 be the set of places for SAN submodels, S_1, S_2 and S_3, respectively. Let $P_1 = \{a, b, c, d\}$, $P_2 = \{e, f, g\}$ and $P_3 = \{h\}$. Furthermore, let $D_6 = \{e, g\}$, $D_7 = \{h\}$, $D_4 = \{e\}$ $D_5 = \{h\}$,

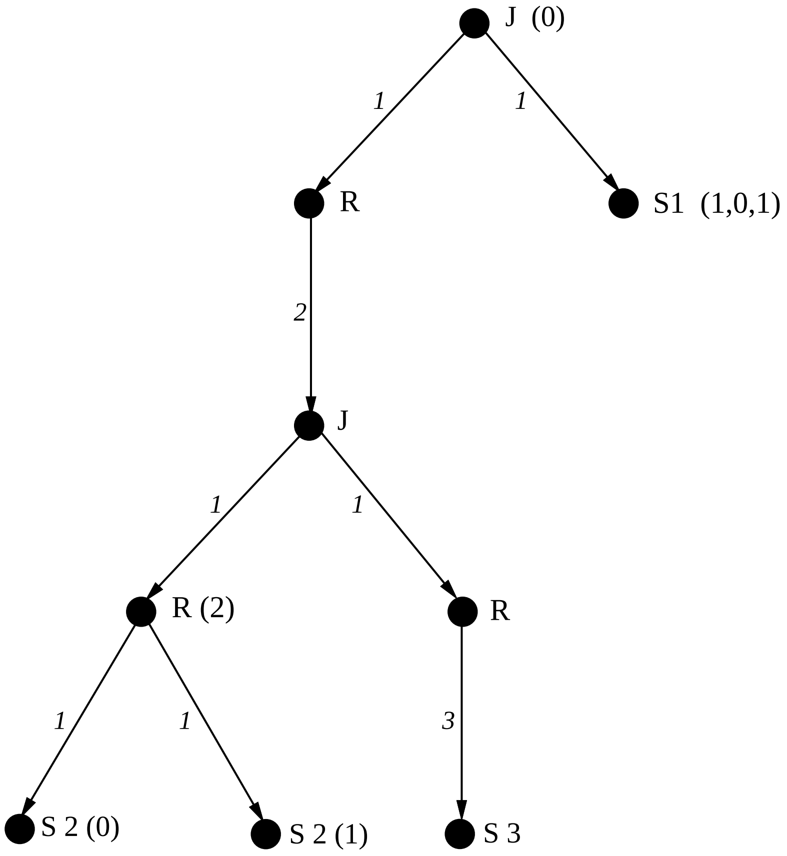


Figure 3.2: Example State Tree

$D_3 = \{h\}$, $D_2 = \{a\}$ and $D_1 = \{h\}$. A possible state tree for this composed SBRM could be as in Figure 3.2. The vectors at the nodes represent possible markings of the places at those nodes.

Nodes that do not have places associated with them, do not have a marking associated with them. The places that are distinguished at the represented operation are also distinguished at the next higher level of operation. The place h of S_3, for example, is at the root node. This place was distinguished at the leaf and at every operation at the next higher level up to the root level. The marking of this place is 0. Similarly, the marking of place a, the distinguished place of S_1 for the join operation represented by the root node, is 0, as is the marking of place e of S_2.

As can be seen on the 3.2, the representation of state tree closely parallels the representation of the composed state using bags and vectors . The number on the arcs depict the number of occurrences of a child (SBRM) in the possible bag or vector at the next higher level of composition. Note that the union of the markings at the nodes on a path from the root to a leaf results in a marking that includes the particular projection of the global marking on the places of the submodel represented by the leaf. A path from the root node to a leaf is referred to as a "route". We define the *route* as a list of nodes that are on a directed path from the root to a leaf. To refer to a node at position l on a route to a leaf j, the notation $route_j[l]$ is used. For example, the route to the left most leaf node is the list formed by all the left most nodes at each level on the tree.

3.3.3 Compound Events and Multiple Future Events Lists

The procedures described later in this chapter make use of the state tree to efficiently perform future events list management. This is achieved by reducing the number of activities that are checked for a change in their "status" (i.e., enabled or disabled) from one composed SBRM stable state to a next stable state.

It is possible to achieve this reduction because of two facts:

1. If an activity a is in a particular status in some marking μ, all replicates of this activity which have their input places in the same marking as the input places of a will be in the same status as a.
2. From one composed SBRM stable state to the next stable state, the only places that could have had a change in their marking are the ones connected to the activity that just completed. Furthermore, the only activities that could have had a change in status are the those connected to these places. We can, thus, identify a region in the composed SAN-based reward model that was affected by the state change. The checks to determine how the state change affected the the status of activities can be limited to that region.

Building upon the first fact, we define a *representative activity* as an activity that "represents" the set of replicate activities $a_1 \in A_1$, $a2 \in A_2$, …, $a_i \in A_i$, …, $a_n \in A_n$, where A_i is the set of activities of a replicate submodel i of a set of n replicate submodels of a particular type in identical markings. Each representative activity is an *event type* in the new simulation technique, whereas activity completions are events.

As seen in the previous section, the state tree is organized in a manner that allows for identification of sets of replicates in a particular marking, as well as the number in that marking. During simulation, for list management purposes, instead of having every replicate activity checked for its status in the current marking, we use the representative activities. These checks are then reduced to a single check per set of replicate activities for a set of submodels in identical markings. The events for each of these replicate activities can be grouped into a list related to the representative activity. We call this list of potential completion times a "compound event". More formally, we define a *compound event* e_a for representative activity a as the list of potential completion times $\{t_1, t_2, \ldots, t_n\}$, where n is the number of activities represented by a.

The number of submodels in a particular marking (n) can be found for every leaf, using the state tree, by multiplying the numbers on the arcs on the route to the leaf. Then, compound events can be built, each with n elements, from the list of future events for each set of submodels represented by a leaf node.

The LAN example is useful to illustrate a possible state tree and corresponding multiple future events lists. Specifically, consider the SBRM of Figure 2.5, where $STATION$ is the submodel of Figure 2.2. $STATION$ is replicated three times holding place *channel* as common. The result is then is joined with the network submodel of Figure 2.4, holding the same place, *channel*, as common.

Figure 3.3 illustrates a possible state tree for the LAN model. The vectors beside nodes represent the marking of the set of places at the nodes. Note that the SAN $NETWORK$ only

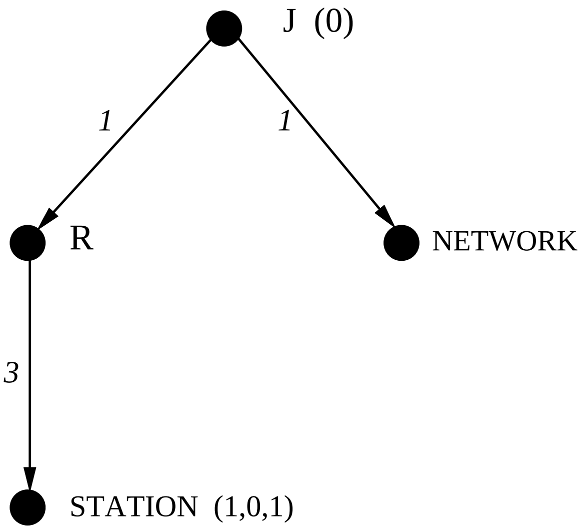


Figure 3.3: A Possible State Tree for LAN model

has one place, the one used in the join operation. The nodes have labels that show what they represent. Arcs to children nodes have an integer associated with the "number of occurrences" of the child SBRMs in that state. There are two routes, one formed by the nodes that lead to the leaf corresponding to submodels of type $STATION$ in marking (0)(1,0,1) and another by the nodes that takes us to the submodel NETWORK in marking (0). Note that we use the notation $(0)(1,0,1)$ to denote the marking of the submodel STATION when the marking of *channel* is 0, the marking of A is 1, B is 0, and C is 1.

Figure 3.4 shows how the multiple future events lists are formed. The compound events are represented by the names of the corresponding representative activity. One leaf is associated with the the set of compound events E_1 and the other with E_2. E_1 has two compound events scheduled, *arrival* and *access*. The first number of the subscript to e relates it to a particular set, and the second identifies it with a representative activity in SAN submodel of type $STATION$. Since there are three replicates of type $STATION$ in the same marking, the result of multiplying the integers at each arc on the route from the node at the highest level to the leaf, *arrival* has three

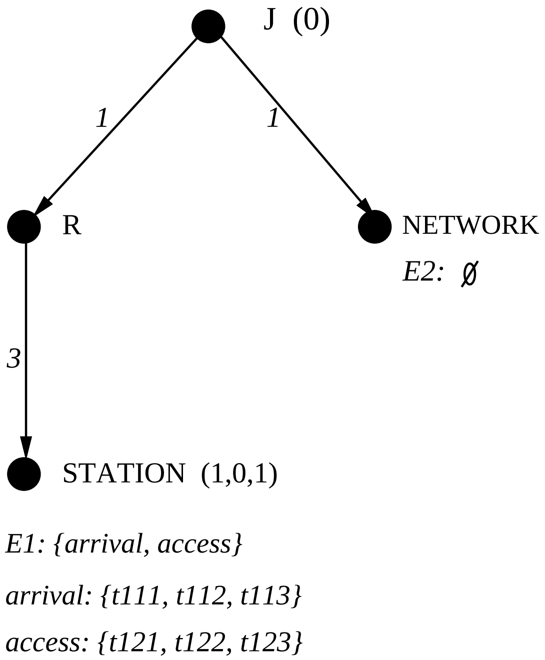


Figure 3.4: Future Events Lists for a Possible State Tree

potential completion times. The compound event *access* similarly has the same number of times. E_2 has no compound events, since there are no activities enabled in $NETWORK$ in this marking.

Figure 3.5 shows a possible state tree resulting from an activity completion, in this case *access*, in one of the submodels of type $STATION$. Because this caused a change in the marking of a place at the lowest level, there is one more leaf. Each leaf represents a set of submodels of the same type in identical markings. For instance, the leftmost leaf is representing a set of two submodels of type $STATION$ in identical markings. The compound events for the future events list associated with this leaf have, therefore, two elements each.

The procedures to execute a composed SBRM are presented in the next subsection.

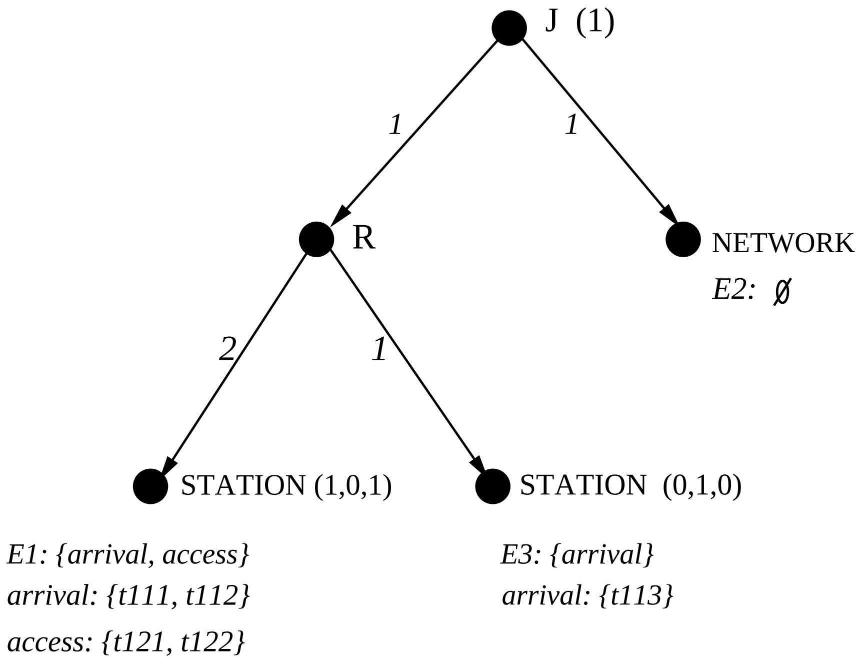


Figure 3.5: Another Possible State Tree

3.3.4 Composed SBRM Execution Procedures for Simulation

Having introduced the notion of a state tree and compound event, it is now possible to describe the methods that can perform the multiple future events list management. We start with Procedure 3.3.1.

Procedure 3.3.1 Initialize(i, n, μ)

(Initializes all sets of compound events associated with leaves on the state tree)

```
Variables :
    $i$ : a node on the state tree.
    $n$ : the number of submodels in same marking on a particular route on state tree.
    $\mu$ : a marking.
Begin :
    $\mu = \mu \cup \mu_i$
    if $type_i$= REP
        for each $c \in C_i$
            Initialize($c$, $n * n_{i,c}$, $\mu$)
    if $type_i$= JOIN
        for each $c \in C_i$
            Initialize($c$, $n$, $\mu$)
    if $type_i = SAN$
```

```
        $E_i = \emptyset$
        for each $a \in En_\mu$
            for $m = 1$ to $n$
                $e_a = e_a \cup$ Generate($a$, $m$, $\mu$)
            $E_i = E_i \cup \{e_a\}$
End.
```

Procedure **Initialize** schedules all enabled activities on the multiple future events lists, given the top node of an initial state tree. The parameters n and μ should be initialized to one and the the empty set, respectively, before the call to this procedure. It recursively traverses the state tree performing the "number of occurrences" operation and forming projected markings. At the leaf nodes, pairs of the type (n, μ) are available, where n is a number of submodels of a particular type in identical marking μ (a projection of the global marking on the submodel represented by a leaf). Compound events are then formed with n potential completion times for each activity enabled in μ This is done by checking if each representative activity is in the set En_μ, the set of enabled activities in marking μ. Recall that all replicates of an activity in replicate submodels in a particular marking will have the have the same status. Thus, only one check per representative activity is needed. Function **Generate**(a, m, μ) is used to generate a potential completion time and a set of reactivation markings, $React_a$, for activity a in the m^{th} submodel, given that it is activated in μ.

Procedure 3.3.2 generates a new state tree. The procedure recursively traverses the path on the tree defined by the route to a representative activity with the earliest potential completion time. When this procedure is called, i is initialized to the root node on the state tree, j is initialized to the leaf node associated with the submodel where the activity completed and a is the activity that completed. k is the position of the earliest time on the list of potential completion times of the compound event associated with a on the set of compound events related to j. μ is initialized to the empty set and $level$ is initialized to 0. On each level of the reverse recursive traversal on the route to leaf j, **BuildNewTree** returns the topmost node for a newly created subtree representing the child of the node at the current level in the state tree for the new state. This child node will substitute the child node that was on the route to the j in case there was a change in marking. The new marking for the new state is also returned (μ').

The projected marking for the replicate submodels represented by the leaf (μ) is constructed by the unions at the beginning of the procedure. This submodel may then be executed by completing the activity. The new marking μ' for the submodel is generated by procedure **ExecuteSAN** [18]. This procedure generates a set of possible next stable markings for a submodel along with a probability distribution defined on the set. A new marking is then chosen probabilistically using the probability distribution. Procedure **BuildNewTree** will then check to see if μ' and μ differ. If a difference is found, a set P of places whose marking has changed is generated by Procedure **Differ**.

The distribution of this set of places on the state tree is very important in the generation of the new state tree. In particular, by finding the highest level on the state tree (hld) related to a place in set P, we can identify the subtree that contains the leaf nodes related to the submodels that changed in marking. All other submodels need only be checked for activity reactivations. The root of the subtree is the the node in the list of nodes in the route to leaf j at level hld.

After this initial check is made for a difference in marking, the procedure will begin to return recursively from the calls made on the route to the leaf node. If no difference is found at the leaf level, the procedure will check μ' and μ at each node, checking to see if the marking of the places at that node differ. Once a difference is found, the flag $difBelow$ will be set to TRUE to signify that a new state tree should be built during the remaining reverse recursive traversal of the tree. Assuming that $difBelow$ is TRUE, at each level where there is a replicate node when returning from recursion, the new state tree built up to that level will be checked to see if this subtree matches any other child subtree in the old state tree. If it does, the integer representing the number of replicates in that marking will be incremented and their compound events will be merged together. Otherwise, a new child node is created with the weight on the arc equal to one.

For the join nodes, the child in the old state tree on the route, will be replaced by the new child configuration which was built on the way up. The subtree for a child that is not on the route will be recursively copied. Different actions are then taken on the lists of future events for the leaves of the copied subtree depending on the level of its top most node. If this node is on a level above *hld*, the projected marking for the submodels represented by the leaves will not have changed, and hence need only be checked to see if any activities should be reactivated. On the other hand, if the level is lower than *hld*, a set of activities connected to the places that changed in marking is created at each node. These are the activities that need to be checked to see if they should:

1. be aborted, if they were active;
2. continue to be active;
3. be activated, if they weren't active.

Since the events are now organized into compound events, only one activity check per compound event is needed. The individual events will then be removed or scheduled n times resulting in new compound events denoted by the e' notation.

The old compound events for a particular leaf on the old state tree will gradually decrease in cardinality as the potential completion times are removed or moved to new leaves created by a split on the tree while the new state tree is being created. At the end of this procedure, a new state tree with the new sets of compound events at the leaves is the result if a difference in marking was found. If no change in marking was detected, state tree is not modified.

Procedure 3.3.2 BuildNewTree(i, j, a, k, μ, *level*)

(Returns the root node for a subtree in state tree for new state and the new projected marking on the places at the nodes on the route to j at and above the level at which the procedure is called.)

Variables:

i: a node on the state tree.

j: the leaf on the state tree corresponding to activity completion.

a: the activity that completed.

k: the position on the list of potential completion times of compound event for activity that completed.

```
        $\mu$: a marking.
        $level$: level on state tree.
Begin:
        $\mu = \mu \cup \mu_i$
        if $type_i = JOIN$
                $(c', \mu') =$ **BuildNewTree**$(route_j[level+1], j, a, k, \mu, level+1)$
                if $difBelow =$ FALSE
                        if $\mu'_i \neq \mu_i$
                                $P =$ **Differ**$(\mu', \mu)$
                                $hld =$ **FindHighestLevelOfDiff**$(P)$
                                if $hld \leq level$
                                        $difAbove =$ TRUE
                                $i' =$ **BuildSubTree**$(i, difAbove, 1, \mu')$
                                $difBelow =$ TRUE
                                return $(i', \mu' - \mu'_i)$
                        else return $(i, \mu' - \mu'_i)$
                else
                        if $hld > level$
                                $difAbove =$ FALSE
                        $i' =$ **BuildJoinNode**$(i, c', difAbove, \mu', route_j[level+1])$
                        return $(i', \mu' - \mu'_i)$
        if $type_i = REPLICATE$
                $(c', \mu') =$ **BuildNewTree**$(route_j[level+1], j, a, k, \mu, level+1)$
                if $difBelow =$ FALSE
                        if $\mu'_i \neq \mu_i$
                                $P =$ **Differ**$(\mu', \mu)$
                                $hld =$ **FindHighestLevelOfDiff**$(P)$
                                if $hld \leq level$
                                        $difAbove =$ TRUE
                                $i' =$ **BuildSubTree**$(i, difAbove, 1, \mu')$
                                $difBelow =$ TRUE
                                return $(i', \mu' - \mu'_i)$
                        else return $(i, \mu' - \mu'_i)$
                else
                        if $hld > level$
                                $difAbove =$ FALSE
                        $i' =$ **BuildReplicateNode**$(i, c', difAbove, \mu', route_j[level+1])$
                        return $(i', \mu' - \mu'_i)$
        if $type_i = SAN$
                $\mu' =$ **ExecuteSAN**$(a, \mu)$
                if $\mu'_i \neq \mu_i$
                        $P =$ **Differ**$(\mu', \mu)$
                        $hld =$ FindHighestLevelOfDiff$(P)$
                        $i' =$ **BuildNewLeafNode**$(i, P, \mu', k)$
                        $difBelow =$ TRUE
                        return $(i', \mu' - \mu'_i)$
                else
                        $difBelow =$ FALSE
                        return$(i, \mu' - \mu'_i)$
```

End.

Procedures 3.3.3 and 3.3.4 directly take actions on the future events lists. Procedure **BuildNewLeafNode** creates a new leaf node and a new future events list for the node, if a difference between μ and μ' was found at the leaf node associated with the activity that completed. The parameters are the original leaf node (i), the set of places that changed in marking (P), the new projected marking (μ') and the position on the list of potential completion times, within the compound event, for the activity with the earliest potential time (k). It returns a leaf for the new state tree. The node is created with same type and subtype of the original leaf node by procedure **CreateNode**. This is needed since a new tree with a possibly different structure is being built recursively. The operations are done once on each k^{th} component for each compound event, where k is the index of the earliest time on the compound event with earliest completion time. These include moving a potential completion time from a compound event to another, deleting a time, reactivating, and scheduling an activity completion. We make the assumption that the indices of the remaining times on the lists of times of each original compound event that are greater than index k are decremented by one representing a decrease by one in the size of each list.

A set of representative activities in the SAN submodel associated with node j connected to the places that changed in marking, P, is generated by a procedure **ActChanged**(P, j). This reduces furthermore the number of activities that need to be checked.

Procedure 3.3.3 BuildNewLeafNode(j, P, μ', k)

(Returns a leaf for the new state tree.)

```
Variables :
    $j$: leaf node on original state tree.
    $P$ : set of places that had a change in marking.
    $\mu'$: the new projected marking on the places at $i$ and nodes above.
Begin :
    $j' = \textbf{CreateNode}(j)$
    $A = \textbf{ActChanged}(P, j)$
    $E_{j'} = \emptyset$
    for each $a \in A$
        if $a \in En_{\mu'}$
            $e'_a = \emptyset$
            if $e_a \in E_j$
                if $\mu' \in React_a$
```

```
                        $e'_a = e'_a \cup$ **Generate**$(a, 1, \mu')$
                    else
                        $e'_a = e'_a \cup t_k$
                    $e_a = e_a - \{t_k\}$
                else
                    $e'_a = e'_a \cup$ **Generate**$(a, 1, \mu')$
                $E_{j'} = E_{j'} \cup \{e'_a\}$
            else
                if $e_a \in E_j$
                    $e_a = e_a - \{t_k\}$
        for each $e_a \in E_j$
            if $a \notin A$
                $e'_a = \emptyset$
                if $\mu' \in React_a$
                    $e'_a = e'_a \cup$ **Generate**$(a, 1, \mu')$
                else
                    $e'_a = e'_a \cup \{t_k\}$
                $e_a = e_a - \{t_k\}$
                $E_{j'} = E_{j'} \cup \{e'_a\}$
End.
```

Procedure 3.3.4 copies a subtree and does future events list management. It performs basically the same operations as Procedure 3.3.3, when there is a difference between the new marking and the old marking of places at nodes that are at the level of the subtree being built or above. A flag, $difAbove$, is passed to this procedure for this purpose. The operations are done leaf by leaf while traversing recursively on the subtree. Every operation on a compound event is done n times, where n is the number of replicate submodels represented by the leaf in the new state tree if no "merging" of leaves occur later in Procedure **BuildReplicateNode** (to be explained later). These include moving a potential completion time from a compound event to another, deleting a time, reactivating, and scheduling an activity completion. As in Procedure **BuildNewLeafNode**, the indices of the remaining times in the original compound events change. In this case, they are decremented by n, reflecting a decrease in size of the compound event.

The activities are checked only for reactivation if there is no difference in the marking of the places related with nodes above the level of this subtree. The parameters for procedure **BuildSubTree** are i, the root of the subtree on the original state tree, $difAbove$, the flag mentioned earlier, n, an integer that should be initialized to 1, and μ', the projected marking on places at nodes on the route to the leaf associated with the activity that completed above and at the level at which the procedure was called. The root node of the new subtree is returned.

Procedure 3.3.4 BuildSubTree(i, $difAbove$, n, μ')

(Returns the topmost node of a subtree for the new state tree)

```
Variables :
    $i$: a node on the state tree.
    $difAbove$ : a flag that indicates if there is a difference between the old and the new marking
    on or above a level above this subtree.
    $n$: the number of submodels in same marking on a particular route on state tree.
    $\mu'$: the new projected marking on places at $i$ and nodes above.
Begin :
    $i' = $ CreateNode($i$)
    $\mu' = \mu' \cup \mu_i$
    if $type_i = REPLICATE$
        $C_{i'} = \emptyset$
        for each $c \in C_i$
            $c' = $ BuildSubTree($c$, $difAbove$, $n * n_{i,c}$, $\mu'$)
            $n_{i',c'} = n_{i,c}$
            $C_{i'} = C_{i'} \cup \{c'\}$
    if $type_i = JOIN$
        $C_{i'} = \emptyset$
        for each $c \in C_i$
            $c' = $ BuildSubTree($c$, $difAbove$, $n$, $\mu'$)
            $n_{i',c'} = 1$
            $C_{i'} = C_{i'} \cup \{c'\}$
    if $type_i = SAN$
        if $difAbove = $ FALSE
            $E_{i'} = \emptyset$
            for each $e_a \in E_i$
                $e'_a = \emptyset$
                if $\mu' \in React_a$
                    for $m = 1$ to $n$
                        $e'_a = e'_a \cup$ Generate($a$, $m$, $\mu'$)
                else
                    for $m = 1$ to $n$
                        $e'_a = e'_a \cup \{t_m\}$
                        $e_a = e_a - \{t_m\}$
                $E_{i'} = E_{i'} \cup \{e'_a\}$
        else
            $A = $ ActChanged($P$, $i$)
            $E_{i'} = \emptyset$
            for each $a \in A$
                if $a \in En_{\mu'}$
                    $e'_a = \emptyset$
                    if $e_a \in E_i$
                        if $\mu' \in React_a$
                            for $m = 1$ to $n$
```

```
                        $e'_a = e'_a \cup$ **Generate**$(a, m, \mu')$
                        $e_a = e_a - \{t_m\}$
                else
                    for $m = 1$ to $n$
                        $e'_a = e'_a \cup \{t_m\}$
                        $e_a = e_a - \{t_m\}$
            else
                for $m = 1$ to $n$
                    $e'_a = e'_a \cup$ **Generate**$(a, m, \mu')$
            $E_{i'} = E_{i'} \cup \{e'_a\}$
        else
            if $e_a \in E_i$
                for $m = 1$ to $n$
                    $e_a = e_a - \{t_m\}$
    for each $e_a \in E_i$
        if $a \notin A$
            $e'_a = \emptyset$
            if $\mu' \in React_a$
                for $m = 1$ to $n$
                    $e'_a = e'_a \cup$ **Generate**$(a, m, \mu')$
                    $e_a = e_a - \{t_m\}$
            else
                for $m = 1$ to $n$
                    $e'_a = e'_a \cup \{t_m\}$
                    $e_a = e_a - \{t_m\}$
            $E_{i'} = E_{i'} \cup \{e'_a\}$
    return$(i')$
End.
```

The algorithm described by Procedure 3.3.5 creates a join node on the new state tree. It makes use of Procedure 3.3.4 to copy the subtrees on the original state tree of all the children of the join node other than the child that contains the leaf for the submodel that had an activity completion ($oldChild$). This child gets replaced by the new child that was built. Parameter i is initialized to the join node on the original state tree. Parameter c' is the child node on the new state tree that was built previously. $difAbove$ is the flag mentioned earlier and μ' is the projected new marking on the places at node i and nodes above its level on the route to the leaf associated with the activity completion. A join node for the new state tree is returned.

Procedure 3.3.5 BuildJoinNode$(i, c', difAbove, \mu', oldChild)$

(Returns a join node for the new state tree)

Variables :
    i: a node on state tree.

```
    $c'$: the child that will go in place of old child.
    $\mu'$: the new projected marking on places at $i$ or at nodes above.
    $oldChild$: the child subtree where the activity completed.
Begin :
    $i' = \textbf{CreateNode}(i)$
    $C_{i'} = \emptyset$
    for each $c \in C_i$
        if $c \neq oldChild$
            $c'' = \textbf{BuildSubTree}(c,\ difAbove,\ 1,\ \mu')$
            $n_{i',c''} = 1$
            $C_{i'} = C_{i'} \cup \{c''\}$
        else
            $n_{i',c'} = 1$
            $C_{i'} = C_{i'} \cup \{c'\}$
    return $(i')$
End.
```

Finally, Procedure 3.3.6 builds a replicate node for the new state tree. This procedure may actually split the tree by creating a new child node or merge two child nodes because they are found to be the same. Two children are said to be the same if the markings of the places associated with their nodes are the same. Since we are building the state tree from the bottom up, returning on the route that took us down, it is known that the markings for the places associated with the nodes above the replicate node on this route will be the same. We may, thus, conclude that the projected markings on the submodels represented by the leaves below will be respectively equal. If two nodes are merged, every compound event for each set of compound events, E_i, for a leaf i on the "old" child (c) can be merged with the corresponding compound event on the new set $E_{i'}$ for new leaf i', on the new child (c'). This is done by a procedure (**MergeEvents**(c, c')) that traverses both children and performs the merge operation described above while at the leaves. As with Procedure **BuildJoin**, the parameter i is initialized to the join node on the original state tree. Parameter c' is the child node on the new state tree that was built before the call to the procedure. $difAbove$ is the flag mentioned earlier and μ' is the projected marking on the places at node i and nodes above its level on the route to the leaf associated with the activity completion. A replicate node for the new state tree is returned.

Procedure 3.3.6 BuildReplicateNode(i, c', $difAbove$, μ', $oldChild$)

(Returns a replicate node for the new state tree.)

```
Variables :
    $i$: a node on state tree.
    $c'$: the child that will go in place of old child.
    $\mu'$: the new marking.
    $oldChild$: the child subtree where the activity completed.
Begin :
    $i'$ = **CreateNode**($i$)
    $C_{i'} = \emptyset$
    $found$ = FALSE
        for each $c \in C_i$
            if $c \neq oldChild$
                if **CompareSubTree**($c$, $c'$) = NOT_SAME_MARKING
                    $c''$ = **BuildSubTree**($c$, $difAbove$, $n_{i,c}$, $\mu'$)
                    $n_{i',c''} = n_{i,c}$
                    $C_{i'} = C_{i'} \cup \{c''\}$
                else
                    $n_{i',c'} = n_{i,c} + 1$
                    **MergeEvents**($c$, $c'$)
                    $C_{i'} = C_{i'} \cup \{c'\}$
                    $found$ = TRUE
            else
                if $n_{i,c} > 1$
                    $c''$ = **BuildSubTree**($c$, $difAbove$, $n_{i,c} - 1$, $\mu'$)
                    $n_{i',c''} = n_{i,c} - 1$
                    $C_{i'} = C_{i'} \cup \{c''\}$
        if $found$ = FALSE
            $n_{i',c'} = 1$
            $C_{i'} = C_{i'} \cup \{c'\}$
        return ($i'$)
    End.
```

3.3.5 Example State Generation

We now illustrate how the procedures just presented can be used to generate new state trees and future events lists. Take the LAN model composed of three replicates of the station submodel of Figure 2.2 joined with the network submodel of Figure 2.4 as an example. Assume an initial marking as described by Figure 3.6 and Figure 3.7.

The state tree for the composed model is as given in Figure 3.3. The elements on the vector beside a leaf node of type *STATION* are $\mu(A)$, $\mu(B)$ and $\mu(C)$, respectively. The marking of place *channel* is in the vector besides the join node since this place is at that node.

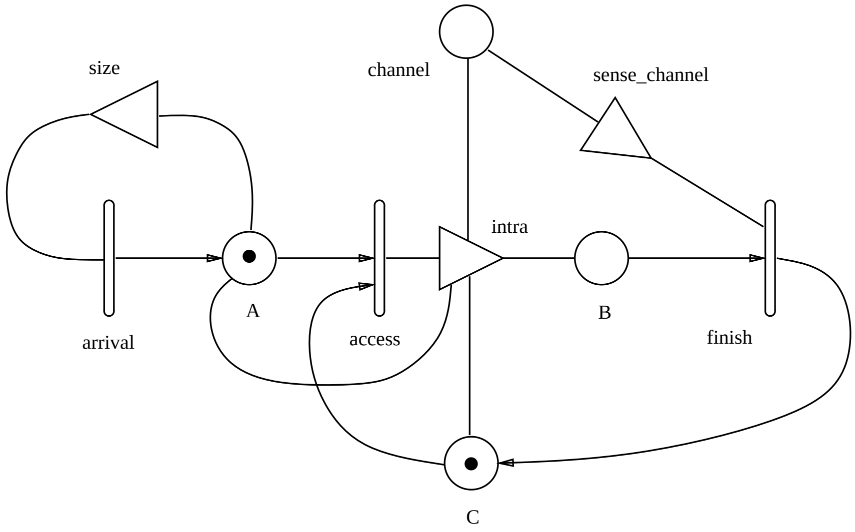


Figure 3.6: Station Initial Marking

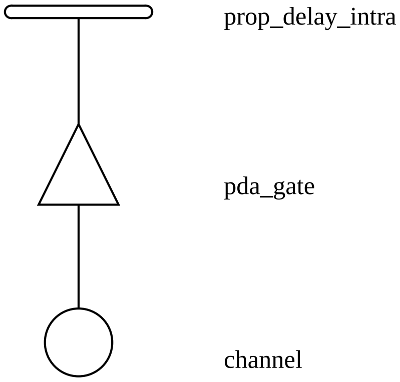


Figure 3.7: Network Initial Marking

To create the future events list, procedure **Initialize** takes the root node, $n = 1$, and μ initialized to the empty set. The procedure starts by including the marking of places at the join node, i.e., the set μ_i, in μ. Then **Initialize** is called again with i equal to the replicate node, $n = 1$, and $\mu = \mu_i$.

Now the node type is $REPLICATE$, and the union will be performed between μ and the marking of the places at this node. In this case, μ is not changed since there are no places at the node. **Initialize** is called once again resulting in a pair (n, μ), where n is now equal to three and μ is the projected marking on the places of the submodels represented by the leaf node for submodel $STATION$.

At this point, procedure **Generate** will generate three potential completion times for each activity of this submodel enabled in marking μ. The activities enabled are *arrival* and *access*. These potential completion times are inserted in compound events with same names as the enabled activities. Each compound event is inserted in the set E_i, the set of compound events for this leaf node.

Due to the recursive nature of the procedure, after several calls, we will get to the leaf node for submodel $NETWORK$. By an argument similar to that above, the elements of the pair (n, μ) are now the integer 1 and the projected marking on the places of submodel $NETWORK$. Activity *prop_delay_intra* is not in the set of enabled activities in μ and so an empty future events list is generated at this node. When the procedure terminates, the result is the state described by Figure 3.4.

Assume now that the earliest potential completion time is t_{123}. This is a completion time for the activity of type *access* in one of the replicates of the station submodel. Procedure **BuildNewTree** takes i equal to the root node of the current state tree, j equal to the node that corresponds to the set of compound events that contains the earliest potential completion time, a as representative activity related to this event type, μ equal to the empty set and *level* equal to the level at the root node, which is zero. Recall that each leaf node has a route associated with it which takes us to that node. This procedure starts by traversing the path on the route all the way to the leaf. At this

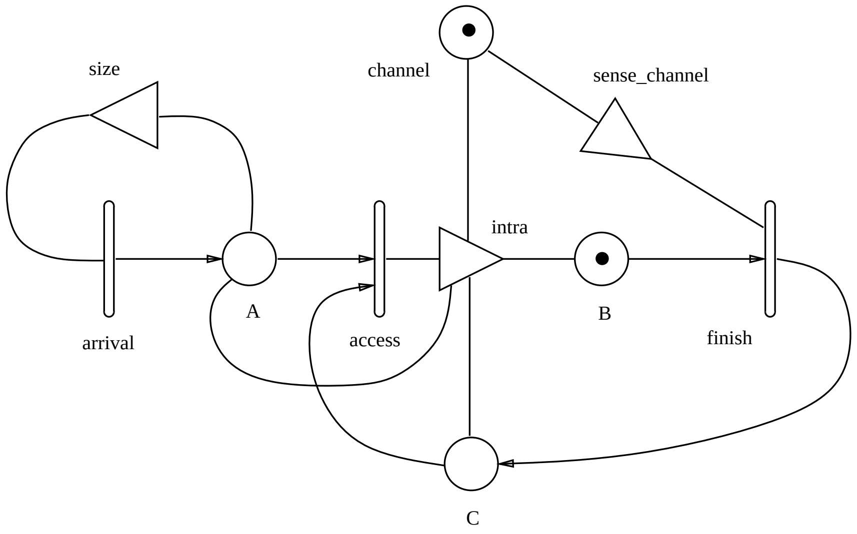


Figure 3.8: Station In New Marking

point, μ is the projected marking of the submodel in which *access* will complete. **ExecuteSAN** then generates possible next stable markings reached upon completion of the activity, i.e., a set of projected markings on the places of the submodel type represented by the leaf node which can result on the completion of *access*. Then a marking (μ') is chosen probabilistically from this set to be the new marking for the third replicate submodel. The submodel of the station in this new marking is seen in Figure 3.8.

Since there is a difference between the old marking and the new marking in places at the leaf node, a set of places of places that had a change in marking is generated along with the highest level on the route at which a place had a change in marking. In this case, the set is $\{A, B, C, channel\}$ and the highest level of difference is 0, meaning that all submodels had a change in marking.

Procedure **BuildNewLeaf** then creates a leaf node for submodel $STATION$ in the new marking. Compound events are now assigned to the new leaf node. The set of activities that are connected to places in P is, in this case, the set of all its activities. The only activity in this set that is enabled in μ' (the new submodel marking) is *arrival*. A compound event will be, thus, be created for *arrival* and the potential completion time t_{113} will be moved from the compound event *arrival* in E_1 into the newly created compound event of same name. The new compound

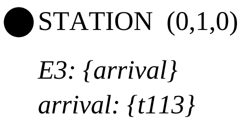


Figure 3.9: Node for Station In New Marking

event *arrival* will be inserted in E_3, the set of compound events corresponding to the new leaf. No activities need to be reactivated, since the set of reactivation markings is null for each activity. Finally, this compound event is inserted in the newly created set of compound events. Compound event *access* in E_1 has the 3^{rd} potential completion time ($t123$) deleted since the activity *access* is no longer enabled in μ'. Figure 3.9 shows the result of this procedure.

BuildNewTree then returns the new node from the recursion and calls Procedure **BuildReplicateNode**. A new replicate node is created followed by a pass through the set of child nodes. **BuildSubTree** is called to build the subtrees for the new replicate node. When this procedure has control, the projected marking for the submodels represented by the other leaf node is found using a union operation. It is known that there is a difference above or at the level at which **BuildSubTree** was called. The activities in this leaf node thus need to be checked for a change in status. We generate the set of activities connected to the places of the set of places where changes occurred. For this node, the only activity on this set is $finish$ (the places at the leaf node for these station submodels remained in the same marking), which is not enabled in the new marking. For all other compound events in the old set, there will be created a new compound event of the same type. Two elements will be transferred from each compound event to each newly created corresponding compound event.

BuildSubTree then returns the new leaf node with the corresponding new future events list. Since there is only one other child (the one from which the new leaf node was created) we decrement its number by one. There are now only two submodels in identical markings. Finally, both leaves are inserted in the set of children. Figure 3.10 is the result.

A join node will then be built by **BuildJoinNode** when **BuildNewTree** returns from the recursion to the highest level on the tree. Now all children except the child that has processed

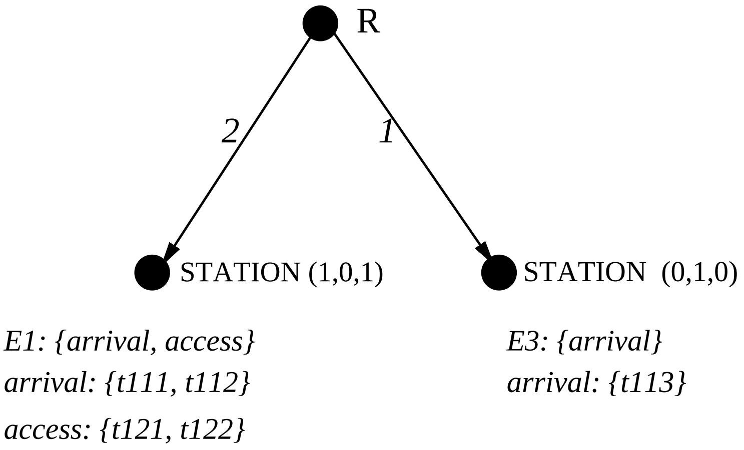


Figure 3.10: Replicate Node for New State Tree

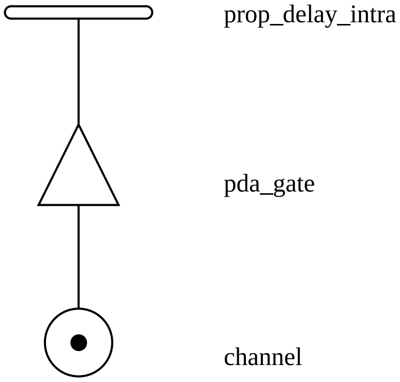


Figure 3.11: Network New Marking

on the way back from the traversal (*oldChild*) will be copied recursively by **BuildSubTree**. There is only one child other than the one that was on the route down to the leaf for the station submodel where the activity completed. It is the leaf node for submodel of type $NETWORK$. **BuildSubTree** builds the projected marking, in this case the marking of place *channel*. This submodel is in a marking as illustrated by Figure 3.11.

The set of activities for this submodel connected to places that changed in marking consists of the single element *prop_delay_intra*. This activity is not in $En_{\mu'}$ and there was no compound event for it in E_j (it was not in En_{μ}). There is no other event in E_j. Therefore, a null future events list is created for this leaf node and the control returns to **BuildJoinNode**. The node returned by **BuildJoinNode** is inserted in the set of children for the join node as is the replicate node from Figure 3.10 node. When **BuildSubTree** is finished, we have the state tree and future events as in Figure 3.5.

3.4 Comparison of Efficiency with Traditional Method

The example of a state generation just presented, although simple, is sufficient to illustrate the advantages of the new procedures over the traditional method when there are replications in the composed model. In the example, there were a total of six events initially scheduled. To generate the new state, the traditional method would require a total of six activity checks for the first step in future events list management. The checks for reactivation would amount in the same number as in the method presented in this chapter, since reactivations require that all activities which were enabled and continue enabled be reactivated if they were activated in a given marking. This requires a total check on the list of future events. The last step in the traditional method would result in four checks (one for each of the three $finish$ activities in the three replicates and one for activity *prop_delay_intra*) in addition to the checks done in the first step. The total adds to ten checks. The new method results in a total of five checks. **BuildNewLeaf** does one check for each representative activity since all were connected to places which had a change in marking. **BuildSubTree** does one check for representative activity $finish$, when building the replicate node, and one on activity *prop_delay_intra* when building the join node. This is a 50% reduction in the amount of times a costly operation is performed. Simulation of composed SBRM models that contain large number of replications will thus benefit significantly if the new methods are used.

CHAPTER 4

Simulator Execution and Variable Estimation

4.1 Introduction

The purpose of a simulation is to study the dynamic behavior of a system. This is achieved by constructing a model that allows us to investigate several dynamic characteristics of the system. These characteristics are represented by variables defined for the model. Typically, the end results of a simulation are confidence intervals for characteristics of the variables. Estimates of the mean and variance are, in general, most often wanted. These can be obtained by inferences on the output data generated by a simulation.

This chapter explains how both a steady-state and a terminating simulator were developed, using the principles of the last chapter, for simulating composed SAN-based reward model.

4.2 Random Number Generator

To truthfully represent real system events that occur randomly in time, the simulator must have a random number generator that generates uniformly distributed $[0,1]$ random numbers that appear to be independent. There are two uses for random numbers in a composed SBRM simulator. In particular, every timed activity has a distribution function defined. Samples from this distribution are used to generate potential completion times. In addition, after activity completions, a next stable marking needs to be selected, probabilistically, from a set of possible next stable markings. These are done by transformations on random numbers. In a sense, we generate a possible numerical value that the random variables "potential completion time" and "case chosen" can take on, according to their distribution.

One of the most performed operations in simulation is random number generation. It is, then, of tremendous importance that, besides being reliable, the mechanism for generation of random numbers be fast. Tauseworthe generators are widely known random number generators [9]. These generators exhibit very interesting characteristics. In particular, they are very fast, independent of machine architecture and generate fairly good random numbers with very wide period. This is the type of random number generator used in the implementation of the simulator.

The algorithm is based on shift register operations. The algorithm is summarized below, assuming that the seed U_i is stored in a machine word X:

1. Copy X to T.
2. Left shift X by q bits filling with zeroes.
3. Let $X = X \; xor \, T$, copy X to T (bitwise exclusive or)
4. Right shift T by $q - p$ bits filling with zeroes.
5. $X = X \; xor \, T$ now contains U_{i+1}

The values for p and q were chosen to be 31 and 13, respectively, from a table in [19]. These values give maximum period, which is the length of the sequence of subsequent values generated starting at the initial seed and ending before the initial seed is repeated.

4.3 Confidence Interval Generation

Confidence intervals are constructed for the chosen variables by generating several, say n, independent realizations of model behavior, each of them resulting in an observation. The confidence interval is then given by:

$$\overline{X}(n) \pm t_{n-1,1-\alpha/2}\sqrt{\frac{s^2(n)}{n}}$$

where $\overline{X}(n)$ is the mean for the estimators $X_1, X_2, \ldots, X_n$ resulting from the n realizations, $t_{n-1,1-\alpha/2}$ is a value from the students t distribution, $s^2(n)$ is the sample variance and $1-\alpha/2$ is the level of confidence. We iteratively generate realizations until the width of the interval reaches an acceptable size.

The manner in which realizations are generated depends on the type of measure to be estimated. If one wants to define measures relative to an interval or point of simulated time, then a *terminating* simulation used. This type of simulation is normally used when transient characteristics of the system are of interest. It terminates when all variable observations are collected. If the goal is to estimate measures defined as limits as the length of the simulation goes to infinity, we use *steady-state* simulation. In this case, the length of the simulation is made long enough to produce "good" estimates of the measure.

The interval-of-time variable $Y_{[t,t+l]}$, the timed-averaged-interval-of-time variable $W_{[t,t+l]}$ and the instant-of-time variable V_t should be estimated with terminating simulation. In addition, if one wants to estimate time between specific completions of an activity using a variable of type $TB^a_{i,j}$, then terminating simulation should be used. Estimates of instant-of-time variables $V_{t\to\infty}$ and time between completions of activities, TB_a, are estimated using steady-state simulation.

Two estimators were implemented for each variable: the sample mean and the sample variance. Both are calculated using methods that avoid problems of machine overflow, which may occur when performing a long series of summations.

Terminating simulation is done using the method of independent replications [9]. It works by running several simulations (replications) of the system, each with different random number streams. By using different streams of random numbers, we make the runs independent. Each replication is "terminated" when the variables for that replication are collected. At each end of a replication a new observation for each variable is obtained and confidence intervals for the estimates are generated. Additional replications are made until the width of the confidence interval for each estimator is within the desired range. This range is specified as a width relative to the sample mean, the *relative width.*

A confidence interval generation for the mean using this method is as follows:

1. Generate M independent replications $X_1, \ldots, X_M$ of the random variable.
2. Calculate the sample mean

$$\hat{\mu} = \overline{X} = \frac{1}{M} \sum_{m=1}^{M} X_m$$

3. Calculate the sample variance

$$s^2 = \frac{1}{M-1} \sum_{m=1}^{M} X_m^2 - \frac{M}{M-1} (\overline{X})^2$$

4. Calculate confidence the interval half width, hw:

$$t_{M-1,1-\alpha/2} \sqrt{\frac{s^2}{M}}$$

5. Find *relative width* $hw/\hat{\mu}$
6. If relative width is acceptable stop, else continue.
7. $M = M + 1$
8. Generate another observation, X_M and go to step 2.

For the variance, one could simply use s^2 as the estimate and build the confidence interval similarly to the mean. However, the confidence interval for the mean was constructed making the necessary assumption that the X_m be normally distributed. Unlike the the result for the mean, the confidence interval for the variance is quite sensitive to deviations from the normal assumption. Use of this method could thus produce incorrect estimates of the confidence interval width. Instead, we use a method known as *jackknifing* [8], which is much less sensitive to the distribution of the X_m. Using this method, we find the sample variance as in the method for the mean confidence interval, but we also find s_j^2 for $j = 1, \ldots, M$, the sample variance of $X_1, \ldots, X_M$ with the observation X_j removed:

$$s_j^2 = \frac{1}{M-2} \sum_{m \neq j} X_m^2 - \frac{M-1}{M-2} (\overline{X}_j)^2$$

where

$$\overline{X}_j = \frac{1}{M-1} \sum_{m \neq j} X_m$$

Next, we calculate "pseudovalues" $Z_j = Ms^2 - (M-1)s_j^2$, for $j = 1, \ldots, M$. The expected value of Z_j is the true variance, σ^2.

Now we calculate the sample mean $\overline{Z}$ of the Z_j, $j = 1, \ldots, M$ and their sample variance, s_z^2, where

$$\overline{Z} = \frac{1}{M} \sum_{j=1}^{M} Z_j$$

and

$$s_z^2 = \frac{1}{M-1} \sum_{j=1}^{M} (Z_j - \overline{Z})^2.$$

The confidence interval is then

$$\overline{Z} \pm t_{M-1,1-\alpha/2} \sqrt{\frac{s_z^2}{M}}$$

The method of batch means, used in steady-state simulation, parallels the method of independent replications except that the sequences are adjacent, subsequences of the output of a single simulation run. We first determine a length that can be used to partition the sequence into batches of data. We then assume that two observations of the sequence of output that are far enough apart such that they can be considered uncorrelated. Each batch will thus be considered an independent experiment such as with the independent replications but will generate one observation of the mean and one of the variance for the variables.

To generate the confidence intervals correctly, the initial data that are found statistically correlated have to be extracted. This is done by specifying an initial transient cuttoff point. Variables will only be collected after that point. The batches should be at least the same size in length as the initial transient for them to be considered independent. The mean and variance are estimated for each batch and will act as observations for the confidence interval generation.

4.4 Reward Variable Collection

Several variables can be estimated by the simulators simultaneously. For the variables types in the interval-of-time category, the terminating simulator has to be able to detect the limits of their intervals. The variables should be collected during this period of time. Similarly, the instants of simulated time specified for each instant-of-time variable have to be detected to collect an observation of these variables. At each of these points in time, the simulator has to take some actions related to the variables. We will call these points "distinguished time points".

As with the terminating simulator, the steady-state simulator has to act upon the "status" of the variables. Specifically, at a time t_{IT_X}, the initial transient cutoff point for variable X_t, the first batch for this variable should start. This is when the distribution of X_t is considered to be in steady state and therefore the estimations for it can be collected. At a time t_{EB_X}, a batch for the same variable should end. This point in time denotes the end of an experiment with the model and we have an outcome for the chosen estimators for the variables. A new batch for X_t should, then, be started with a new time for it to end. Variable updating at these times is done in a different manner than within a batch, since ends of batches, as well as any other limit in time specified for reward variables, might occur at times between two subsequent advances of the simulator clock.

One can think of these as points in time where an "event" occurs at which the simulator must act upon the mechanisms of data collection for a variable. Similar to the future events, a list of these distinguished time points is kept, now in increasing order.

The actions on the variable due to one of these time points can be performed by a procedure such as **ExecuteTimePointsList** in the general procedures for both types of simulators (Procedures 4.6.1 and 4.6.2, to be discussed later). At each new state, before the simulation clock time advances to $nextActivityCompletionTime$, if the first distinguished time point in the list is less than $nextActivityCompletionTime$, then the list of distinguished time points is iteratively traversed until this is true. Then some action is taken on the variable to which the time point was specified.

The distinguished time points can all be characterized as end points of some length of time both in steady-state and terminating simulation. A time point may be a "start" endpoint or an "end" endpoint. In steady-state simulation, each variable will have a start time and a batch length. During variable initialization, the distinguished time points list is created by inserting in the proper position on the list a time point of type "start", t_{IT_X} for reward variable X_t and another of type "end", t_{EB_X}, which is obtained by adding t_{IT_X} to the batch length. We are scheduling ends of batches.

When a time point t_{IT_X} arrives, the "status" of X_t is changed from $NOT\_STARTED$ to $JUST\_STARTED$ and a time t_{SB_X}, a start time for the variable, is initialized to t_{IT_X}. Observations of the estimates defined for X_t are obtained when a t_{EB_X} arrives. The confidence interval half widths and the relative widths for each estimate are then calculated. If the relative widths are acceptable, the status of the variable is changed to $ACCEPTABLE$. Otherwise, t_{SB_X} is assigned equal to t_{EB_X} which is afterwards increased by the batch length and reinserted properly in the distinguished time points list. The variable is re-initialized for the new batch to start. The variable changes from $STARTED$ to $JUST\_STARTED$. Figure 4.1 shows these variable status changes for different distinguished points. Times t_1, t_2, t_3, and t_4 are event times.

For terminating simulation, the limits of the intervals $[t, t+l]$ for variables of type $Y_{[t,t+l]}$ and $Y_{[t,t+l]}$, and the instant of time t of a variable of type V_t are distinguished time points. Every reward variable has a time t_E on the list, which is the time to make the observation and update the estimators. Variables of of the interval of time categories also have a time t_S. This is the start

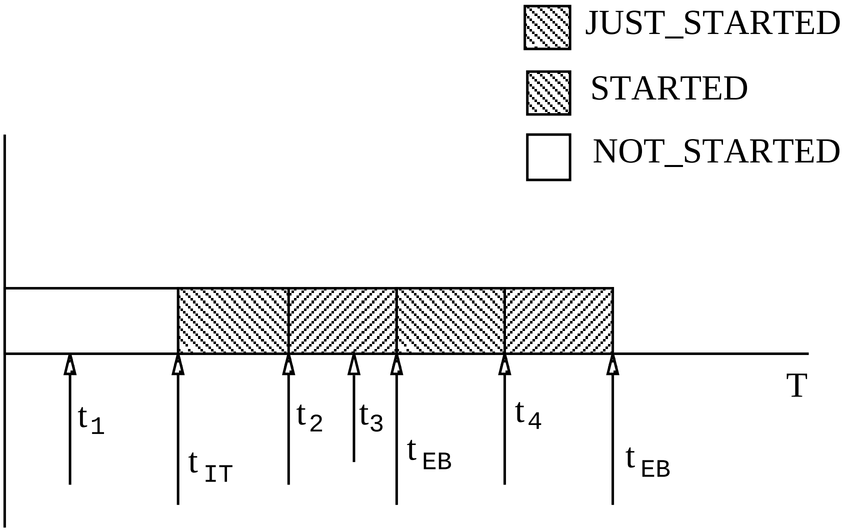


Figure 4.1: Variable Status Changes in Steady State Simulation

time of the interval $[t, t+l]$. Again, these distinguished points in time are inserted in the list at variable initialization.

During execution, when a time t_S arrives, the "status" of the corresponding variable is changed from $NOT\_STARTED$ to $JUST\_STARTED$. At time t_E, time has come to make an observation for the variable as mentioned. The variable changes in status from $STARTED$ (explained later) to $ENDED$ and, if the variable is of the interval category, its value is updated up to t_E. Then the estimators are updated and confidence intervals half widths and relative widths are calculated for the range check. The status of the variable is changed from $ENDED$ to $ACCEPTABLE$, if the check result is satisfactory. When the last t_E on the list arrives, if the all the reward variables have acceptable confidence interval widths together with the ones for the time between completion variable, the simulation has ended. Otherwise, the variables are re-initialized and a new replication is started with a different random number stream.

After executing the actions for the time points in the list, we then advance the simulation clock to the next activity completion time and proceed to update the variables.

4.5 Updating the Variables

4.5.1 Terminating Simulation

The variables need to be updated at each state change. This done differently according to the type of index set for the variable. A sequence of time-between-completions variables is a stochastic process $\{TB^a_{i,j} \mid i \in I\!N,\, j \in I\!N,\, j > i\}$, whose index set is countable. In terminating simulation, we fix the values of i and j defining a random variable for which we obtain one observation per simulation replication. We will then have a set of discrete observations of $TB^a_{i,j}$ that enables us to estimate the mean and variance of the variable and generate confidence intervals. More specifically, when an event of type a occurs for the i^{th} time, we wait for the j^{th} event of this type for an observation to be made. At this point, the estimators are updated and the status of the variable is changed from $NOT\_OBSERVED$ to $OBSERVED$. The confidence interval is built and the relative width is checked. If acceptable, the status of the variable is changed to $ACCEPTABLE$.

The variables of the interval-of-time categories are updated at each occurrence of an event. A variable of type $Y_{[t,t+l]}$ representing an accumulated reward during the interval is updated according to its status. A $JUST\_STARTED$ variable of this type will initialized to its rate reward in the current marking multiplied by the difference $(currentSimulationTime - t_S)$, where t_S is its starting limit. The status will be changed to $STARTED$. When the variable is this status, the last simulation clock time is used in place of t_S to weight the rate reward to the time the rate was in effect. The result is then added to the impulse reward of the last activity which completed and accumulated.

A variable of type $W_{[t,t+l]}$ is just the variable $Y_{[t,t+l]}$ averaged on the interval. The variable is indexed continuously in time requiring. The mean of a continuous time process can be evaluated iteratively by the following recursion:

$$m_i = \frac{t_1 m_{i-1} + (t_2 - t_1) value}{t_2}$$

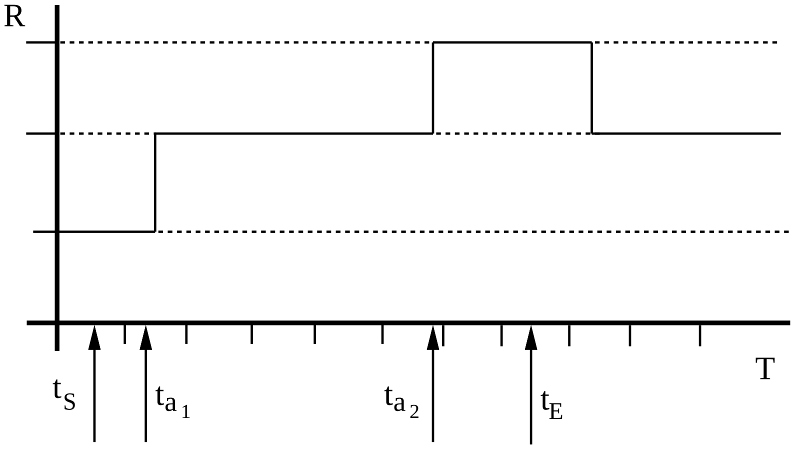


Figure 4.2: Time Averaged Interval of Time Variable Updating

To illustrate how this is done, consider Figure 4.2. This figure has a possible path for the a possible reward defined for some variable. At t_{a_1}, an event of type a occurs causing the variable to be initialized using *value* equal to the reward rate for the marking the SAN is in during the interval $[t_s, t_{a_1}]$, $t_1 = t_s - t_s = 0$, and $t_2 = t_{a_1} - t_s$. At t_{a_2}, we use *value* equal to the impulse reward due to an activity of a_1 plus the reward rate for the marking during interval $[t_{a_1}, t_{a_2}]$. Then, $t_1 = t_{a_1} - t_s$ and $t_2 = t_{a_2} - t_s$. Finally, at time t_E, *value* is equal to the impulse reward for representative activity a_2 plus the reward rate in the marking during $[t_{a_2}, t_E]$, $t_1 = t_{a_2} - t_s$ and $t_2 = t_E - t_s$. After the last iteration, m_i is the time averaged observation.

Instant-of-time variables are not updated in this manner, since they are defined for a particular time.

4.5.2 Steady State Simulation

Recall that, using the batch means method, the variables are collected throughout the batches. There are new observations of the mean and variance at the end of each batch. The confidence intervals are generated based on these observations, making the assumption that they are statistically uncorrelated.

During a batch, at each event, the estimators for the batch are updated in a discrete manner or a continuous manner according to the type of variable. While the reward variable $V_{t \to \infty}$ has an uncountable index set, the time between completion variable has discrete index set and is, hence, computationally simple to collect. When an activity completes, we need only subtract the current simulation time from the last time the an activity of that type completed and a new observation is made. We then update the estimators for the batch using the equations for means and variances of discrete samples.

The reward variables are indexed continuously in time, requiring methods of mean and variance calculation similar to the weighting method used for the time averaged interval of time variable explained in the previous section. The batch estimators are updated also according to the status of the variables. This is done in an analogous manner, since the starts and end of batches will act as the limits t_S and t_E for that variable. The batch mean for a variable $V_{t \to \infty}$ is evaluated for intervals of length equal to the batch size. We continuously observe the variable throughout the batch. The mean of the batch, in particular, can be obtained using the same method used to collect the time averaged interval of time variable in terminating simulation. The batch for the variance for the batch can be obtained from the following derivation:

$$\sigma_N^2 = \frac{1}{T_N} \sum_{k=1}^{N} \Delta_k (X_k - \overline{m})^2 \tag{4.1}$$

$$= \frac{1}{T_N} \sum_{k=1}^{N} \Delta_k X_k^2 - 2\overline{m} \frac{1}{T_N} \sum_{k=1}^{N} \Delta_k X_k + \frac{1}{T_N} \overline{m}^2 \tag{4.2}$$

$$= \frac{1}{T_N} \sum_{k=1}^{N} \Delta_k X_k^2 - 2\overline{m} m N + \overline{m}^2 \tag{4.3}$$

where T_N is the batch length, X_k is the value at the k^{th} interval, $\overline{m}$ is mean of the batch means, and m_N is the mean for batch N.

To avoid problems of arithmetic overflow, the first term is calculated recursively, again using the method for the mean, since it is the mean of the X_k^2. The variance for the batch can be obtained at the end of the batch. The weighting of the averages are performed in the same manner, substituting t_S and t_E for a variable of type $W_{[t,t+l]}$ by t_{IT} and t_{EB}.

4.6 Basic Algorithms for the Simulators

The previous sections provided general methods for simulation by independent replications and by the batch means methods. We now present algorithms which apply those methods to solve composed SAN-Based reward models.

The algorithm for terminating simulator is as follows :

Procedure 4.6.1 Terminating Simulator

```
Begin
    while estimators are not in relative width
        InitializeVariables
        $currentTime = 0.0$
        $\mu = \emptyset$
        $i = TopInitialStateTreeNode$
        Initialize($i$, 1, $\mu$)
        CalculateRateRewards($i$)
        while $distinguishedTimePointsList \neq NULL$
            $nextActivityCompletionTime =$ Pop($leafWithEarliestEvent$)
            ExecuteTimePointsList($currentTime$, $nextActivityCompletionTime$)
            $lastTime = currentTime$
            $currentTime = nextActivityCompletionTime$
            UpdateVariables($lastTime$, $currentTime$)
            $j = leafWithEarliestEvent$
            $a = representative\_Activity\_for\_Earliest\_Event\_on\_EarliestLeaf$
            $k = position\_of\_Earliest\_Event\_on\_EarliestLeaf$
            $i =$ BuildNewTree($i$, $j$, $a$, $k$, $\mu$, 0)
            CalculateRateRewards($i$)
        change random number generator initial seed
End.
```

The algorithm starts with the basic initialization procedures. **InitializeVariables** performs initialization of the variables after every replication. Procedure **CalculateRateRewards** traverses the state tree collecting information about the marking to generate values of rate rewards in the current state. Then, a realization of a simulation is generated. Before advancing the clock, the next activity completion time is obtained by function **Pop**. This function takes a pointer to the leaf on the state tree corresponding to the earliest event. By selecting the earliest compound event for the leaf followed by the earliest time in this set, we obtain the next completion time. To avoid comparisons while gradually building the new state tree with the new multiple future events lists, this event is not yet removed from the list. Instead, we generate a new potential completion time for it. If there should be a change from the current marking to the next marking, the representative for this "possible" future event will actually be checked later in either procedure **BuildNewLeaf** or procedure**BuildSubTree**, depending on where at the tree the the differences occurred. This "possible event" will be removed in case the representative activity is no longer enabled. Otherwise, it is a future event and will be inserted in the proper compound event formed when creating the new state tree. Finally, if there should be no change in marking, the activity is activated again and its potential completion time was now generated and the structure of the state tree and multiple future events list will not change.

The checks on the reward variables which have distinguished time points greater than the current time but less than the next activity completion times are then performed along with any necessary variable collection operations described in Section 4.4 (Procedure **ExecuteTimePointsList**). After advancing the simulation clock, we update the variables according to the methods discussed previously (Procedure **UpdateVariables**) and proceed to build the new state tree executing the composed SBRM. Once the model is in the new state, we may calculate the reward rates in effect for the new marking.

When there are no more distinguished time points on the time points list, this run should terminate and re-start with a new random number stream, if necessary.

The algorithm for the steady-state simulator is very similar:

Procedure 4.6.2 Steady-State Simulator

```
Begin
    Initialize Variables
    $currentTime = 0.0$
    $\mu = \emptyset$
    $i = TopInitialStateTreeNode$
    Initialize(i, 1, μ)
    CalculateRateRewards(i)
    while estimators are not in relative width
        $nextActivityCompletionTime =$ Pop(leafWithEarliestEvent)
        ExecuteTimePointsList(currentTime, nextActivityCompletionTime)
```

```
        $lastTime = currentTime$
        $currentTime = nextActivityCompletionTime$
        **UpdateVariables**($lastTime$, $currentTime$)
        $j = leafWithEarliestEvent$
        $a = representative\_Activity\_for\_EarliestEvent\_on\_EarliestLeaf$
        $k = position\_of\_Earliest\_Event\_on\_EarliestLeaf$
        $i$ = **BuildNewTree**($i$, $j$, $a$, $k$, $\mu$, 0)
        **CalculateRateRewards**($i$)
End.
```

This procedure, at this level of abstraction, only differs from the terminating simulation in that it executes a single run. The main differences are at the lower levels of details where the batching for the variables is performed.

Both of the algorithms presented in this chapter were implemented in a performance evaluation package called *UltraSAN*. The next chapter illustrates the usefulness of these techniques through an example study of a CSMA/CD local area network.

CHAPTER 5

Results

5.1 Introduction

The LAN example discussed in earlier chapters was simulated using the procedures presented in Chapters 3 and 4, both in transient and steady state. The goal was to demonstrate the efficiency of those methods and to investigate the behavior of the LAN with respect to different variables. First, the effectiveness of the procedures on the run time was studied using the steady state simulator by varying the size of the LAN. Next the use of SANs is illustrated in an investigation of the transient behavior of a ten station CSMA/CD LAN. Finally, a 100 station LAN was simulated for an investigation on how different loads influences several variables in steady state.

5.2 CSMA/CD LAN Model Description

Although the model of the LAN was presented earlier for illustration of composed SBRM execution, we explain it in more detail here. The protocol employed is a variant of the non-persistent CSMA/CD. In it, a station begins to transmit if it senses an idle channel. Then, if the channel was actually idle, the transmission proceeds normally. Otherwise, a collision occurs, since another station had begun transmitting but there was not enough time for the signal to propagate to the first station at the time the channel was sensed. If this is the case, the collision is cleared after an exponentially distributed amount of time. When a busy channel is detected the station waits and attempts again after a delay period. The model of Figure 2.2 represents a single station connected to the channel. Figures 2.1 and 2.2 have the gates and activity parameters. Arrivals of messages to the station queue are represented by completions of activity *arrival*. Place A represents the queue. The number of messages waiting to be transmitted is represented by the marking of place A. Gate *size* represents the maximum size of the queue by means of its predicate. Activity *arrival* is only enabled if the number of messages in the queue is less than the systems capacity.

A station sensing the channel is represented by a completion of activity *access*. When this happens, the one of the following outcomes may take place:

1. If the channel is idle (marking of *channel* is zero), the marking of the place *channel* is set to one signifying the start of a transmission. The marking of B is set to one to indicate that a transmission is in progress for the station.
2. If the channel is being used but the signal has not yet propagated (marking of *channel* is one), the marking of *channel* is set to three, indicating that there is a corrupted message on the channel and the marking of A is incremented by one meaning that a retransmission is necessary.
3. If a corrupted message is on the channel (marking of *channel* is three), the marking of place A is incremented by one, since retransmission is necessary.
4. If a the transmission of a message is occurring on the channel for enough time to propagate the message to all stations the marking of place A is incremented by one, since the transmission was not possible, and the marking of place C is set to one indicating that the station is, once more, waiting to seize the channel.

Completions of activity $finish$ determine times required to transmit a message. Its distribution is dependent on the marking of *channel* having different parameters depending on whether the message on the channel is corrupted or not.

The network submodel was also presented earlier (Figure 2.4). The time for the message to propagate is represented by activity *prop_delay_intra*. The gate *pda_gate* activates this activity when a unpropagated message is on the channel. When the *prop_delay_intra* completes after an exponentially distributed time, the marking of the place *channel* is set to two, to indicate that message has been propagated to all stations.

The composed model was obtained by the operations illustrated Figure 2.5.

5.3 Variables

5.3.1 Transient Simulation

The variables studied were queue length and the probability that propagated message is on the channel assuming a load of 30% on the network. The reward structure for the first variable defined for the station submodel was:

$$\mathcal{C}_{st}(a) = 0, \quad \forall a \in A$$

$$\mathcal{R}_{st}(\nu) = \begin{cases} i & \text{if } \nu = \{(A, i)\} \\ 0 & \text{otherwise,} \end{cases}$$

To determine the probability that a propagated message is on the channel at a particular time, we define a reward structure for the network submodel as:

$$\mathcal{C}_{N}(a) = 0, \quad \forall a \in A$$

$$\mathcal{R}_{N}(\nu) = \begin{cases} 1 & \text{if } \nu = \{(channel, 2)\} \\ 0 & \text{otherwise,} \end{cases}$$

When markings and activity completions of a submodel do not contribute to the variables, we define a "null" reward structure for this submodel such that the impulse reward for all activities in the submodel and the rate reward for all markings of this submodel are zero.

Variables of type $E[V_t]$ were used to estimate both variables at different points in times using the terminating simulator.

5.3.2 Steady State Simulation

Expected queue length and fraction of time a propagated message is on the bus were also simulated in steady state. Besides these variables, four reward variable and one time-between-completions variables were investigated. The reward variables were: probability that an incoming message is blocked due to a full queue, fraction of time the bus is idle, fraction of time an unpropagated message is on the bus and fraction of time a corrupted message is on the bus due to collision. The time-between-completions variable was used to compute expected time between arrivals to the LAN.

Reward structures for the fraction of time various types of packets are on the bus were specified in a similar way. The general form for the reward structure defined on the network submodel was:

$$\mathcal{C}_N(a) = 0, \quad \forall a \in A$$

$$\mathcal{R}_N(\nu) = \begin{cases} 1 & \text{if } \nu = \{(channel, i)\} \\ 0 & \text{otherwise,} \end{cases}$$

where i is the marking of the channel when the action on the bus was taking place.

To determine the blocking probability, the reward structure defined on the station submodel was:

$$\mathcal{C}_{st}(a) = 0, \quad \forall a \in A$$

$$\mathcal{R}_{st}(\nu) = \begin{cases} 1 & \text{if } \nu = \{(A, 2)\} \\ 0 & \text{otherwise,} \end{cases}$$

5.4 Run Times for Different LAN Sizes

The steady state simulator was run for various number of stations to verify the effect of the developed algorithms on simulator run times. The values were obtained for a fixed number of

Figure 5.1: Run Times for Five Batches - Steady State

batches when four variables were being estimated. The simulator was run on DECstation 2100 with 20 Mbytes of memory.

As can be seen in Figure 5.1, the ten station model took almost 300 seconds to generate five batches. When the number of stations was doubled, there was a very small increase in time. Furthermore, the number of stations increased up to 1500% (a LAN with 150 stations), the run time only increased 300% relative to the time for the ten station model. It can be seen that, at some portions of the curve, there was little increase in time, although the number of possible future events increased significantly.

5.5 Terminating Simulation Results

The plot for expected total queue length at particular times given an initial queue length of one at each station is given in Figure 5.2. The level of confidence of the confidence intervals for

Figure 5.2: Queue Length versus Time

the points is 95%. The expected individual queue lengths can be obtained by dividing the result by the number of stations (10).

The results show that the initial transient phase ends little after ten seconds given that there was one message at the queue per station at start time.

The plot for the expected values of the probability that a propagated message is on the channel at times varying from one to thirty seconds is shown in Figure 5.3. Steady state is reached at about 15 seconds given the initial marking.

5.5.1 Steady State Simulation Results

The results of how the seven variables estimated behave when varying the load on the network in steady state are shown in plots. To simulate different loads, the arrival rate parameter defined for activity *arrival* was modified according to the desired load. For example, for a load of 50%, a rate of .005 was used for each station. The results for variable expected queue length and blocking

Figure 5.3: Probability a Propagated Message is on the Channel versus Time

probability should be divided by the number of stations to obtain the answers for individual stations.

The results show that, even at high loads, the CSMA/CD performs very well. One tends to expect large queue lengths as the load increases but the results show small queue lengths.

The results for the variables defined for the channel show that at high loads there is an increase in collisions and utilization but the performance is still very satisfactory.

Figure 5.4: Expected Queue Length versus Fraction of Full Load

Figure 5.5: Blocking Probability versus Fraction of Full Load

Figure 5.6: Fraction of Time a Propagated Message is on Bus versus Fraction of Full Load

Figure 5.7: Fraction of Time an Unpropagated Message is on Bus versus Fraction of Full Load

Figure 5.8: Fraction of Time an Corrupted Message is on Bus versus Fraction of Full Load

Figure 5.9: Fraction of Time Bus is Idle versus Fraction of Full Load

Figure 5.10: Expected Time Between Arrivals versus Fraction of Full Load

CHAPTER 6

Conclusions and Further Research

The objective of this research was to develop methods for efficient simulation of composed SAN-based reward models, and to design and implement terminating and steady state simulators based on these methods. To accomplish these objectives, we:

a) Developed a future events list management technique suitable for simulation of composed SAN-Based reward models.

b) Developed and implemented steady state and terminating simulators for composed SBRMs.

c) Illustrated the effectiveness of the techniques presented in this thesis by evaluating a CSMA/CD local area network.

The first objective was the most challenging. Much effort was made to make maximum use of the composed model structure to reduce the the cost of future events list management. We have provided formal algorithms to perform future events list management and state generation for simulation of composed SAN-based reward models. Efficient methods to locate the differences between the new state and the old state were of fundamental importance.

As for the second objective, the simulators were implemented in *UltraSAN* and will soon be used in simulation of large realistic systems. Methods for collecting several variables simultaneously were provided.

With regard to the third objective, we have verified the usefulness of the state generation methods for composed SAN-based reward models. we have simulated a highly replicated computer system. The results show small increase in run times if considered the large increase in number of possible future events. At some point, a decrease in run time was observed.

6.1 Areas for Further Research

Simulation, in general, is yet to be a completely understood field. There are several areas for future investigation. An interesting area is variance reduction. The traditional method to achieve variance reduction is to run a simulation twice using symmetric random number streams. By creating a SAN model exhibiting some symmetry in structure, one might be able to reduce the variance of a variable during estimation.

Another area of interest would be to implement spectral analysis on the output data. This method has been found to yield very accurate results, since correlation between data is taken into account, rather than relying on assumptions that correlation has been avoided.

As for the simulator implementation, it was found that most of the computational time was spent with memory management. Efforts could be made to improve the data structures in order to reduce memory allocation and deallocation. One way of improving in this area would be to keep available the maximum number of possible child subtrees for every replicate node on the state tree. In the worst case, the number of children nodes for a replicate node is equal to the number of replications performed by the operation it represents. This way, a subtree structure for a child of a replicate node will always be available. Memory allocation and deallocation would be greatly reduced in exchange for managing the available subtree structures.

Another way of reducing time spent in memory management is to have custom made memory allocation and deallocation routines. Stacks of structures used to build a state tree could be created making memory allocation less frequent. The structures would then be "popped" or "pushed" into the stacks by some custom made routines to manage these stacks.

REFERENCES

[1] J. P. Behr, N. Dahmen, J. Muller and H. Rodenbeck, "Graphical modeling with FORCASD", in *Proc. Computer Applications in Production and Engineering*, pp. 61–630, Amsterdam, North-Holland, 1983.

[2] G. Chiola, "A software package for analysis of generalized stochastic Petri net models", in *Proc. Int. Work. on Timed Petri Nets*, pp. 136–143, Torino, Italy, July 1985.

[3] G. Ciardo, J. Muppala and K. S. Trivedi, "SNPN: Stochastic Petri net package", in *Proc. Petri Nets and Performance Models*, pp. 142–151, Kyoto, Japan, December 1989.

[4] J. B. Dugan, K. S. Trivedi, R. M. Giest and V. F. Nicola, "Extended stochastic Petri nets: Applications and analysis", in *Performance 84*, pp. 507–519, Amsterdam, North-Holland, 1984.

[5] H. P. Godbersen and B. E. Meyer, "A net simulation language", in *Proc. Summer Computer Simulation Conf.*, Seattle, WA, August 1980.

[6] R. A. Howard, "Dynamic Probabilistic Systems", Vol. 2, Wiley, 1971.

[7] A. M. Johnson, Jr., and M. Malek, "Survey of software tools for evaluating reliability, availability, and serviceability", *ACM Computing Surveys*, vol. 20, no. 4, pp. 227–269, December 1988.

[8] S. S. Lavenberg, "Computer performance modeling handbook", Academic Press, 1983

[9] A. M. Law and W. David Kelton, "Simulation modeling and analysis", McGraw Hill, 1982.

[10] M. Leszak and H. P. Godbersen "A tool for performance-availability evaluation of distributed systems based on function nets", in *Proc. Int. Work. on Timed Petri Nets*, pp. 152–161, Torino, Italy, July 1985.

[11] M. A. Marsan, G. Balbo and G. Conte, "A class of generalized stochastic Petri nets for performance evaluation of multiprocessor systems", in *ACM Trans, on Computer Systems*, vol. 2, no. 2, pp. 93–122, May 1984.

[12] M. Molloy, *On the integration of delay and throughput measures in distributed processing models*, PhD thesis, UCLA, 1981.

[13] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks", in *Proc. 1984 Real-Time Systems Symp.*, Austin, TX, December 1984.

[14] T. Murata, "Petri nets: Properties, analysis and applications", in *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, April 1989.

[15] S. Natkin, *Reseaux de Petri stochastiques*, PhD thesis, CNAM-PARIS, 1980.

[16] J. L. Peterson, *Petri net theory and the modeling of systems*, Englewood Cliffs: Prentice-Hall, 1981.

[17] C. A. Petri, "Kommunication mit atomaten" Bonn: Institute fur Instrumentelle Mathematik, Schriften des IIM Nr. 3, 1962.

[18] M. Rai, *Design and Implementation of a Reduced Base Model Construction Technique for Stochastic Activity Networks*, Master's thesis, Univ. of Arizona, AZ, 1990.

[19] B. D. Ripley, "Stochastic Simulation", Wiley, 1987.

[20] W. H. Sanders, *Construction and solution of performability models based on stochastic activity networks*, PhD thesis, Univ. of Michigan, MI, 1988.

[21] W. H. Sanders and J. F. Meyer, *A Unified Approach for Specifying Measures of Performance, Dependability, and Performability*, to appear in "Dependable Computing and Fault-Tolerant Systems", Vol. 4, (ed, J. Laprie), Springer-Verlag, 1990.

[22] W. H. Sanders and J. F. Meyer, "METASAN: A performability evaluation tool based on stochastic activity networks", in *Proc. ACM-IEEE Comp. Soc. 1986 Fall Joint Comp. Conf.*, Dallas, TX, November 1986.

[23] W. H. Sanders and J. F. Meyer, "Reduced Base Model Construction Methods for Stochastic Activity Networks", to appear in *IEEE Journal on Selected Areas in Communications*, January 1991.

[24] S. Shapiro, "A stochastic Petri net with application to modeling occupancy times for concurrent task systems", *Networks*, vol. 9, pp. 375–379, 1979.

[25] A. A. Torn, "Simulation nets: A simulation modeling and validation tool", *Simulation*, pp. 71–75, vol. 45, no. 2, August 1985.