From *Discrete Event Dynamic Systems: Theory and Applications,* vol. 3 no. 2/3, pp. 271-300, July 1993.

EFFICIENT SIMULATION OF HIERARCHICAL STOCHASTIC ACTIVITY NETWORK MODELS *

William H. Sanders and Roberto S. Freire

Department of Electrical and Computer Engineering
The University of Arizona
Tucson, AZ 85721 USA
(602) 621-6181

whs@ece.arizona.edu

ABSTRACT

Stochastic extensions to Petri nets have gained widespread acceptance as a method for describing the dynamic behavior of discrete-event systems. Both simulation and analytic methods have been proposed to solve such models. This paper describes a set of efficient procedures for simulating models that are represented as stochastic activity networks (SANs, a variant of stochastic Petri nets) and composed SAN-based reward models (SBRMs). Composed SBRMs are a hierarchical representation for SANs, in which individual SAN models can be replicated and joined together with other models, in an iterative fashion. The procedures exploit the hierarchical structure and symmetries introduced by the replicate operation in a composed SBRM to reduce the cost of future event list management. The procedures have been implemented as part of a larger performance-dependability modeling package known as *UltraSAN*, and have been applied to real, large-scale applications.

Keywords: Stochastic Petri Nets, Stochastic Activity Networks, Discrete Event Simulation, Discrete-Event Dynamic Systems, Hierarchical Modeling.

This work was supported in part by the Digital Equipment Corporation Faculty Program: Incentives for Excellence.

I Introduction

Stochastic extensions to Petri nets have become a popular method to describe the dynamic behavior of discrete-event systems, as evidenced by interest during the last decade. Solution of such models can be done via numerical analysis or simulation. Early efforts consisted of identifying the possible markings of the net as states, and converting the state-level description of the system into a discrete-state, continuous-time [8, 9] or discrete-time [8] Markov process. The process could then be solved numerically using known Markov process solution techniques, yielding many different measures of a system's performance and/or dependability.

While these early attempts to use stochastic Petri nets showed promise, they also pointed out limitations of the technique. In particular, when analytic solution techniques are used, the complexity of the solution (in this case, the size of the resulting stochastic process), grows rapidly as the size of a system increases, quickly resulting in a situation where solution is no longer practical by analytic means. This led to techniques for generating "reduced" (or "compact" or "folded") state spaces, either by hand, using characteristics of the models themselves [6], by using colored Petri nets which have a particular "well-formed" property [2], or by using a hierarchical (or "composed") approach to model specification [17, 19]. All of these techniques make use of symmetries in model, and the strong lumping theorem [5], to reduce the size of the state space required. While they show promise toward alleviating the problem of state-space explosion, there are still many systems that can be represented conveniently using stochastic extensions to Petri nets, but cannot be solved numerically using such techniques.

Such systems include those that have too large (and possibly infinite) state spaces, or delays that are generally distributed. To account for such systems, researchers have turned to discrete-event simulation as a solution method. Discussion of solution of stochastic extensions to Petri nets by simulation is not new, in fact, was considered as early as 1973 by Noe and Nutt [10], who considered the possibility of generating simulation programs from systems described as "E-nets." However, no procedures were given to achieve an automatic translation from E-nets to an executable simulation program.

Later authors described implementations that did this, for example Törn [20], and Sanders et al. [16]. These procedures considered each transition (or activity, in the case of stochastic activity networks) to be an event type in the simulation, and firing of transitions

(completions of activities) to be events. While this identification provided a straightforward method for converting stochastic Petri net descriptions to executable simulations, it resulted (for large nets) in simulations that had a very large number of possible event types, and hence a large overhead per state change in the simulation.

Hence models with a large number of transitions, which (typically) led to extremely large state spaces, also led to simulation programs with high overheads associated with state changes. Fortunately, the structure of the nets themselves could be used to limit the events that must be checked upon each state change. This has been done, for example, for generalized stochastic Petri nets (GSPNs), by conducting an analysis of the conflict and causal connection relations existing among transitions [1]. In effect, a set of relations is generated that specifies which transitions could have a change of status (to enabled or un-enabled) as a result of the firing of each transition.

In addition to this savings, significant savings can be achieved by making use of symmetries in the model, as was done with stochastic process generation for analytic solution. This paper does this, using the idea of "representative activities," which represent a group of activities (or transitions) in the original model. In the new simulation procedures, representative activities become the event types, and completion of any activity in the group corresponding to the representative activity becomes an event. This idea, coupled with the structural checks described in [1], can lead to a very efficient simulation mechanism for stochastic extensions to Petri nets.

Detailed procedures built on this idea are described in this paper, in the context of "stochastic activity networks" (SANs) [7] (a variant of stochastic Petri nets), and a hierarchical model construction technique in which complete models are "composed" by replicating and joining submodels in an iterative procedure [19]. Execution of models built using these techniques can employ a less refined notion of state, relative to the marking of the net. This new state representation, known as a "state tree," uses only representative activities. The result is a set of procedures in which far fewer event types need be checked upon each state change, since each event type represents a group of one or more activities. In addition, structural checks are employed to further reduce the number of checks necessary.

The remainder of this paper is organized as follows. First, in Section II, we provide a brief review of concepts related to SANs and composed SAN-based reward models, the models used in the development of the simulation procedures. Then, in Section III, we briefly review how simulation has traditionally been done with stochastic Petri nets, pointing out

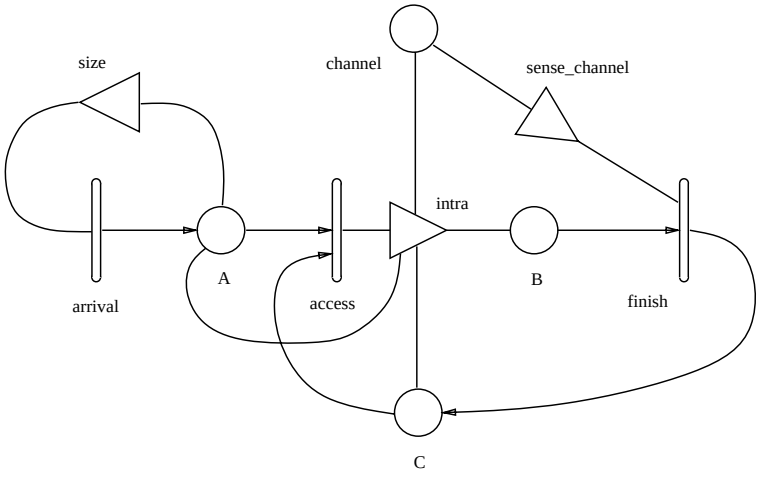


Figure 1: Station Submodel

the potential inefficiencies of this method. Section IV then explains the notion of state that will be employed in the new simulator, and how the composed model structure can be used to reduce the number of event types that need to be considered. Section V gives detailed procedures that implement the state change mechanism for the simulator based upon the notion of state described in the previous section. An example state generation based on these procedures is also given. Section VI highlights an implementation that makes use of the procedures.

II Modeling Framework

Stochastic activity networks (SANs) [7] are a stochastic extension to Petri nets. Structurally, they consist of activities, places, input gates, and output gates. To illustrate activities and other SAN components, we consider a SAN model of a station in a carrier-sense multiple-access network with Collision Detection (CSMA/CD network), shown in Figure 1.

Places are used to represent the "state" of a system (e.g., places *A, B, C,* and *channel*, in Figure 1) and may contain *tokens*. The number of tokens present in a place is called the *marking* of that place. The marking of the SAN is the vector consisting of the markings of all the places in the SAN. Tokens in a place may be given different interpretations, depending upon the purpose of the particular place in the model.

Activities, which are similar to transitions in standard Petri nets, are of two types:

Table 1: Activity timings for a station

Activity	**Distribution**
access	exponential(10)
arrival	exponential(.03)

finish	exponential(if (MARK(channel)==2)
	return(1.0);
	else
	return(5.0);)

timed and *instantaneous*. Timed activities (e.g., *arrival*) represent activities of the modeled system whose durations impact the system's ability to perform. Instantaneous activities (not shown in the example), on the other hand, represent system activities which, relative to the performance variables in question, complete in a negligible amount of time. When an activity *completes*, one token is removed from each of the places directly connected (i.e., not through a gate) to the input of the activity and one token added to each of the places directly connected to the output of the activity. Table 1 shows the timings associated with the activities present in Figure 1. As can be seen, the activity timing can depend upon the marking of places in the network. For example, activity *finish* has an exponentially distributed timing which depends on the marking of place *channel*.

Cases associated with activities (represented as small circles on one side of an activity, and not present in our example) permit the realization of uncertainty concerning what happens when an activity completes. As with activity times, case probabilities can be dependent on place markings.

Input gates and *output gates* permit greater flexibility in defining enabling and completion rules than with regular Petri nets. In particular, input gates have *enabling predicates* and *functions*, while output gates have *functions*. The enabling predicate can be any computable predicate (taking on true and false values) of the places connected to it, and, as seen in that which follows, controls the enabling of an attached activity. The function associated with each input gate describes an action (change in marking) that will occur upon completion of the activity. As with predicates, gate functions can be any computable function on the places connected to the gate.

Activities are *enabled* if there is at least one token in each of the places directly connected to the activity and if the predicate of each connected input gate is true (i.e., *holds*). As an

Table 2: Input gate specifications for station submodel

Gate	Enabling Predicate	Function
sense_channel	$MARK(channel) ==$ $2 \mid\mid MARK(channel) == 3$	$MARK(channel) = 0;$
size	$MARK(A) < 2$	*identity*

example, consider activity *arrival*, which has input gate *size* connected to it. This activity models the arrival of a packet to the MAC protocol sublayer queue. According to the enabling predicate specified in table 2, the activity is enabled only when there are less than two tokens in A. Thus, the gate represents a control of admission to a finite queue, whose size is two. When the predicate of each of the input gates holds, and there is at least one token in each input place for an activity, the activity is *activated*.

If the activity remains enabled, it will *complete* after a time determined by its activity time distribution. On completion, the gate function, which is in this case the identity function (i.e., a "do-nothing" action), will be executed. If the marking of the network changes, before the activity completes, such that it is no longer enabled, it is *aborted*. Activities may also be restarted, or *reactivated*, under certain circumstances. In particular, for each marking in which an activity may be activated, a set of *reactivation markings* can be defined. Then, if prior to completion, one of these markings is reached, the activity is *reactivated*, i.e., aborted and then immediately activated. This provides a mechanism for restarting activities, either with the same or a different activity time distribution.

Output gates, together with directly connected output places, are used to specify the action to be taken upon completion of an activity. The function associated with each output gate can be any computable function on the connected places. For example, as shown in table 3, output gate *intra*, represent an attempt to access the channel. The current status of the channel is encoded by the number of tokens in place *channel* (zero tokens, an idle bus; one token, a packet that has not yet propagated to all stations on the bus; two tokens, a propagated packet; and three tokens, a collision). Accordingly, the action taken depends on the current status of the bus, and may change the status of the bus. For more details on the execution of SANs, see [12].

Performance variables are defined on individual SANs, using the idea of a reward model. Reward models consist of three components: a stochastic process, a reward structure, and

Table 3: Output gate functions for station submodel

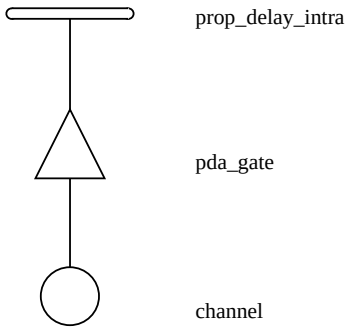
Gate	Function
intra	$if\ (MARK(channel) == 0)\ \{$ $\quad MARK(channel) = 1;\ MARK(B) = 1;\ \}$ $else\ if\ (MARK(channel) == 1)\ \{$ $\quad MARK(channel) = 3;\ MARK(C) = 1;\ MARK(A)++;\ \}$ $else\ if\ (MARK(channel) == 2)\ \{$ $\quad MARK(C) = 1;\ MARK(A)++;\ \}$ $else\ if\ (MARK(channel) == 3)\ \{$ $\quad MARK(C) = 1;\ MARK(A)++;\ \}$

a performance variable defined in terms of the reward structure and the stochastic process. Rates are normally assigned to states, and impulses assigned to state transitions. We have extended this idea to SANs by assigning impulse rewards to activity completions, and rate rewards to having particular numbers of tokens in places. For more information, see [18].

Complete models (called *composed models*) are constructed using the SAN formalism discussed above, by "composing" one or more SAN submodels, together with their defined reward structures, using "replicate" and "join" operations. The *replicate* operation replicates a submodel a certain number of times, duplicating all activities, input gates, output gates, and some subset of the set of places of the submodel. The subset of places that is not duplicated (and hence held common to all replicas), is called the set of *distinguished* (also known as "common") places for the operation. Thus the replicate operation allows one to construct composed models that consist of several identical component submodels, keeping some subset of the places common, to allow interaction between the submodels.

The combination of several different submodels is accomplished using the *join* operation. The join operation produces a composed model which is a combination of the individual submodels. In this case, the join operation associates a list of distinguished places with each submodel. The operation merges the first place in each list to form a single place, the second place to form anther place, and so on. Particular elements on the lists can be null, allowing certain places to be created from a proper subset of the joined submodels. The result of both the replicate and join operation is a *composed SAN-based reward model* (SBRM), and is a submodel on which additional operations can be performed.

For example, a model of a CSMA/CD type network (Figure 3) can be constructed using



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: Network Submodel

the station submodel considered earlier (Figure 1) and a SAN model which represents the bus (Figure 2). In Figure 3, the letters C and R refer to the impulse rewards and rate rewards, respectively, associated with a SAN. As can be seen from the figure, the complete model of the network is constructed by taking the station submodel, replicating it n times (resulting in an n station network), and making place *channel* a distinguished place. This is shown in Figure 3, using an R to denote the replicate operation, the superscript n to denote the number of times the submodel is replicated, and $\{channel\}$ to denote the set of places that is distinguished. This SBRM is then joined to the submodel representing the bus of the network (Figure 2), holding the place *channel* common between the two submodels. This operation is denoted, in the figure, by showing the two lists of distinguished places, each consisting of the place *channel*.

III Traditional Method

Our goal is to make use of the composed model and SAN structure to reduce the number of checks that need to be done per state change. However, for comparison purposes, we first describe a simple approach to simulation, in which each activity in the model is an event type and every activity completion is an event [12]. Procedure III.1 is such a procedure.

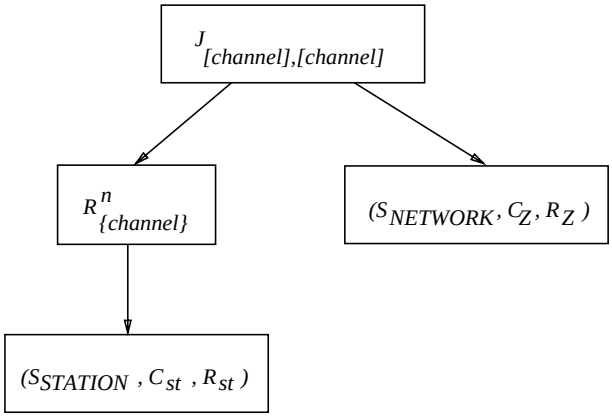


Figure 3: Example Composed SBRM for CSMA/CD LAN

During execution of the procedure, at each event, the model will behave according to its event type. Future events are scheduled based on sampled completion times for the enabled activities. The procedure **GenerateEvent** generates a sampled 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 (markings with no instantaneous activities enabled in them) generated by **ExecuteSAN**. This procedure executes a SAN submodel until all possible next stable markings are reached and the probabilities of reaching them are determined, given a particular stable starting 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. The events that remain enabled must be kept in the future event list, unless the current marking is a reactivation marking for the event. If it is, it is first removed from the list, and then added back to the list, with the new scheduled completion time. 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.

Procedure III.1, translated from Sanders [12], is as follows:

Procedure III.1 GenerateBehavior

Begin

(E is the set of activities that have been activated but have not yet completed)

$E = \emptyset$
μ = INITIAL MARKING
(EN_μ is the set of activities enabled in μ)
for each $a \in EN_\mu$
    e_a = **GenerateEvent**(a, μ)
    $E = E \cup \{e_a\}$
while $E \neq \emptyset$
    cur_a = **Earliest**(E)
    $E = E - \{e_{cur_a}\}$
    μ' = **ExecuteSAN**(cur_a,μ)
    for each $e_a \in E$
        if $a \notin EN_{\mu'}$
            $E = E - \{e_a\}$
    for each $e_a \in E$
        ($React_a$ is the set of reactivation markings of a, for the marking it was activated in.)
        if $\mu' \in React_a$
            $E = E - \{e_a\}$
    for each $a \in EN_{\mu'}$
        if $a \notin E$
            e_a = **GenerateEvent**(a, μ')
            $E = E \cup \{e_a\}$
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 sampled 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, replicas 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 procedure ends with another check on all activities to find the ones that were not enabled in μ but are enabled in μ'. As the composed model grows in size, the computation time spent on these operations can make solution by simulation inefficient.

IV Making Use of the Composed Model Structure

To avoid the inefficiencies pointed out in the previous section, we make use of the SAN and composed model structure. This is done by developing a less-refined notion of state,

which does not distinguish between replicate submodels, and using this notion of state to develop a mapping between activities and event types in the simulation program. To illustrate this, the LAN example introduced in Section II is used.

A State Representation

A definition of "state" is needed to make the description of a composed model execution procedure possible. Recall that a composed SBRM is a result of operations on SBRMs that themselves may be composed models or individual SBRM submodels. We can represent this composition graphically by a tree, as described in Section II. A notion of "state" can then 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 replicas in each existing submodel "state" is kept at the replicate node representing the operation. Finally, at the lowest level, the "state" of a SAN model is its normal marking. The complete state is then the impulse reward due to the last activity 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 one level to those at the next lower level based on the operation done at that level. Each replicate operation is represented as a bag [11] of states of the replicated submodels. Join operations are represented by a vector of states of the joined submodels. At the lowest level of operation, the elements of the vectors or bags are the projected markings of the global marking on places of individual SAN submodels.

For example, consider the SAN-based reward model of Figure 4. The composed state for that composed SBRM can be represented as, using the notation of [19]:

$$
\begin{aligned}
State &= (\mathcal{C}(a), V) \\
V &= (B_1, \mu^2) \\
B_1 &= < V_{11}, V_{12} > \\
V_{11} &= (B_{111}, B_{112}) \\
V_{12} &= (B_{121}, B_{122}) \\
B_{111} &= < \mu^{1111}, \mu^{1112} > \\
B_{112} &= < \mu^{1121}, \mu^{1122}, \mu^{1123} > \\
B_{121} &= < \mu^{1211}, \mu^{1212} > \\
B_{122} &= < \mu^{1221}, \mu^{1222}, \mu^{1223} > .
\end{aligned}
$$

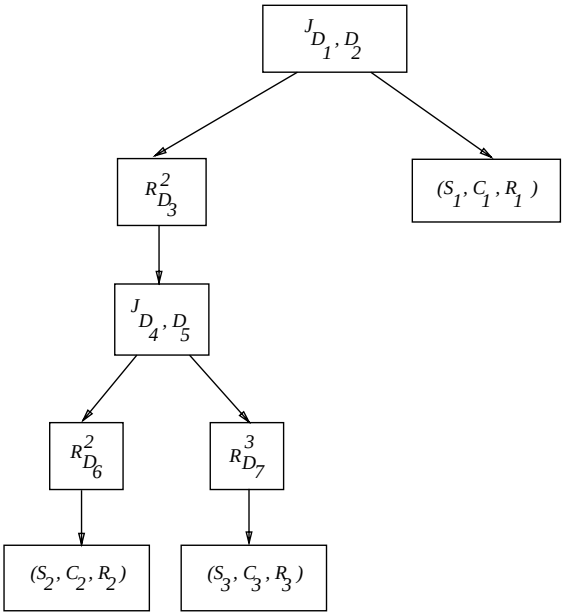


Figure 4: Example Composed SAN-based Reward Model

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 [11]. 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 of the operation at the fourth level below the highest.

B 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 are a graphical representation of the composed state described in the previous subsection, which make it easier to understand the mechanics of state change.

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 type 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 leaves of 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 at 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 nodes 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 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 a replicate or join, we denote its set of children nodes as C_i.

To illustrate the use of state trees, consider once again Figure 4. In this figure, the letters "D_i" denote sets of distinguished places at certain levels in the composed model. 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, in Figure 4, let $D_6 = \{e, g\}$, $D_7 = \{h\}$, $D_4 = \{e\}$ $D_5 = \{h\}$, $D_3 = \{h\}$, $D_2 = \{a\}$ and $D_1 = \{h\}$. A possible state tree for this composed SBRM could be as in Figure 5. Note that the number of children below a replicate node is equal to the number of submodels in distinct markings. 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

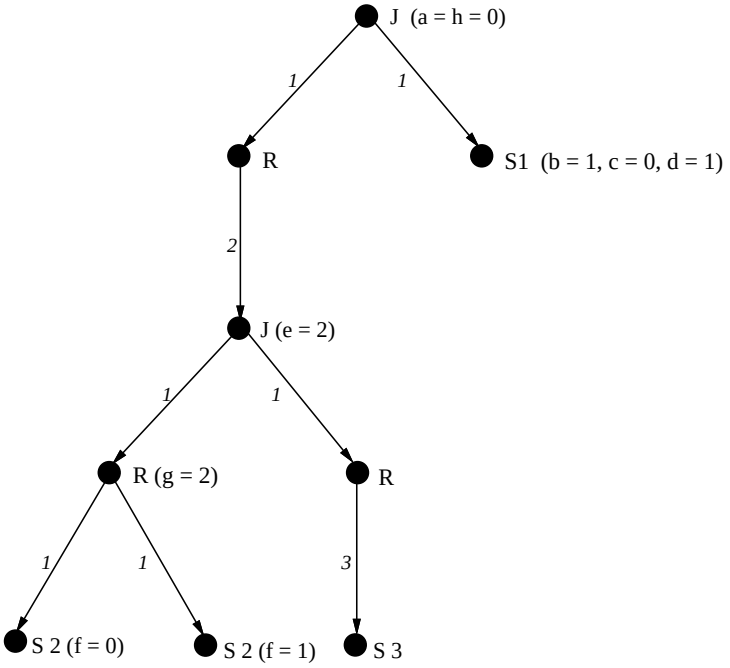


Figure 5: Example State Tree

with them. The places that are distinguished at the represented operation are also distinguished (in the SBRM) at all operations on all paths from that level to the root. 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 in Figure 5, the representation of state tree closely parallels the representation of the composed state using bags and vectors. The number on each arc depicts 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 for the SAN at the leaf node. 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.

C Compound Events and Multiple Future Events Lists

The procedures described in the next section 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 replicas of this activity which have their input places in the same marking as the input places of a will have the same status as a.

2. From one composed SBRM stable state to the next stable state, only those places connected (either directly, or through an input or output gate) to the timed activity and zero or more instantaneous activities that just completed could have a change in their marking. Furthermore, only those activities connected to this set of places can have a change in their status. 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 status of activities can be limited to that region.

This section will build upon the first fact. We define a *representative activity* as an activity that "represents" the set of replica activities $a_1 \in A_1$, $a_2 \in A_2$, ..., $a_i \in A_i$, ..., $a_n \in A_n$, where A_i is the set of activities of the i^{th} replica in a set of n replicas of a particular submodel 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 easy identification of sets of replicas in a particular marking, as well as the number in that marking. During simulation, for list management purposes, instead of having every replica activity checked for its status in the current marking, we use representative activities. These checks are then reduced to a single check per set of replica activities for a set of submodels in identical markings. The events for each of these replica activities can be grouped into a list related to the representative activity. We call this list of sampled completion times a "compound event." More formally, we define a *compound event* e_a for representative activity a as the list of sampled completion times $\{t_1, t_2, \ldots, t_n\}$, where n is the number of activities represented by a.

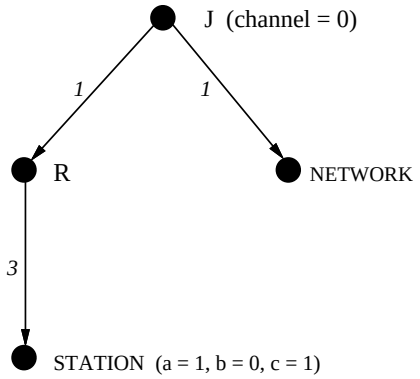


Figure 6: A Possible State Tree for LAN model

The number of submodels (n) in a particular marking 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 3, where *STATION* is the submodel of Figure 1. *STATION* is replicated three times holding place *channel* as distinguished. The result is then joined with the network submodel of Figure 2, holding the same place, *channel*, as distinguished.

Figure 6 illustrates a possible state tree for the LAN model. Note that the SAN *NETWORK* only 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 lead 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 7 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 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 t relates it to a particular set (e.g., E1) and the second identifies it with a representative activity in SAN submodel

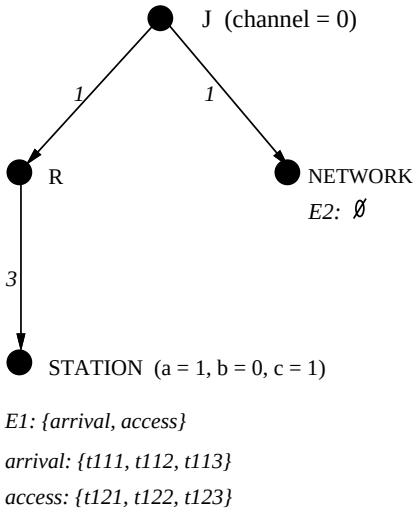


Figure 7: Future Events Lists for a Possible State Tree

of type *STATION*. Since there are three replicas 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 sampled 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 8 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. Note that, because of the completion of access, the marking of place *channel* has changed to one, and activity *prop_delay_intra* in *NETWORK* is now enabled.

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

V Composed SBRM Execution Procedures

Having introduced the notions of state trees and compound events, it is now possible to describe the procedures that perform the multiple future events list management. These procedures describe how to efficiently construct a new state tree, together with its multiple

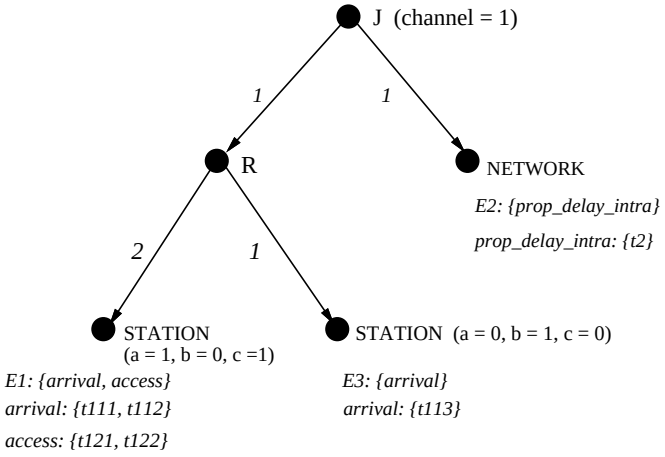


Figure 8: State Tree After Completion of Activity *access* in Figure 7

future events lists, from an existing state tree and future events lists. After the procedures are described, the LAN example is used to illustrate their use.

Figure 9 gives an overview of the organization of the procedures described in this section. Procedure **Main** is the main simulator loop. It calls **Initialize** to build the initial state tree, calls **BuildNewTree** repeatedly to build next state trees (given an existing tree and a representative activity to complete), and performs various performance variable collection and statistical analysis functions. Since the focus of this section is on the state change mechanism, procedure **Main** is not described in detail. Procedure **BuildNewTree** is the controlling procedure for the state change operation. As can be seen from the figure, it calls four other procedures, each which acts on a particular part of the old state tree to build the new tree.

Procedure **BuildNewLeafNode** acts on the SAN in which the activity just completed, and generates a new leaf node where each compound event has a single time, corresponding to the sampled completion time for each activity in the SAN. Procedure **BuildSubTree** is used to build a new subtree when the portion of the tree has changed in a manner such that only the future events lists of the SANs have changed, not the structure of the subtree. This will happen if there is no change in the marking of nodes below this point in the tree (signaled by the flag *difBelow* being FALSE in the following procedures). Procedures **BuildJoinNode** and **BuildReplicateNode** are used to generate new subtrees for the new state tree when a marking change has occurred, and hence the structure of the tree may

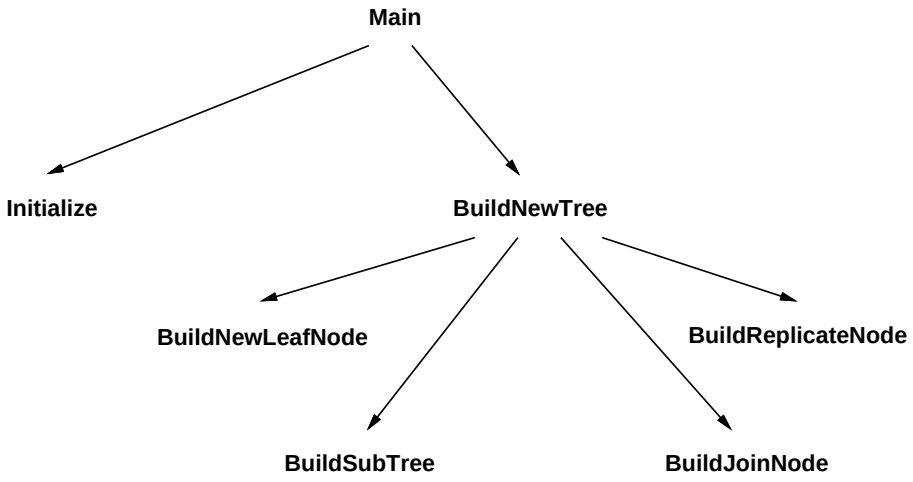


Figure 9: Organization of State Change Procedures

change. Each of these procedures is now discussed in detail.

Procedure V.1 Initialize(i, n, μ)
(Initializes all sets of compound events associated with leaves on the state tree)

Parameters :
i : a node on the state tree.
n : the number of submodels in same marking on a particular route on state tree.
μ : a projected marking.
Begin
$\mu = \mu \cup \mu_i$
if $type_i$= REP
(C_i is the set of child nodes of node i.)
for each $c \in C_i$
Initialize(c, $n * n_{i,c}$, μ)
if $type_i$= JOIN
for each $c \in C_i$
Initialize(c, n, μ)
if $type_i = SAN$
(E_i is the set of representative activities that are enabled at node i, but have not yet completed.)
$E_i = \emptyset$
for each $a \in EN_\mu$
for $m = 1$ to n
$e_a = e_a \cup$ **Generate**(a, m, μ)
$E_i = E_i \cup \{e_a\}$
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 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 an identical marking μ (a projection of the global marking on the submodel represented by a leaf). Compound events are then formed with n sampled 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 replicas of an activity in replica submodels in a particular marking will have the same status. Thus, only one check per representative activity is needed. Function **Generate**(a, m, μ) is used to generate a sampled 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 V.2 (**BuildNewTree**) generates a new state tree, and future events lists, given an existing tree and a list of events. The procedure recursively traverses the path on the tree defined by the route to a representative activity with the earliest 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 sampled 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 of the route to leaf j, **BuildNewTree** returns the topmost node for a newly created subtree. This node represents the child of the node at the current level in the state tree for the new state.

The projected marking for the replica submodels represented by the leaf (i.e., 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 the procedure **ExecuteSAN**. This procedure generates a set of possible next stable markings for a submodel, along with a probability distribution defined on the set, and then chooses a new marking probabilistically using the derived 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 placement of the set of places whose markings have changed 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 (done by procedure **Find HighestLevelOfDiff**, in the procedures that follow), we can identify the subtree that contains the leaf nodes related to the submodels that have changed in marking. All other submodels need only be checked for activity reactivations. The root of the subtree is 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, the new state tree built up to that level (using procedure **BuildSubTree**) 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 replicas 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 leaf 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 sampled 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, the state tree is not modified.

Procedure V.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.)

Parameters :

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 sampled completion times of compound event for activity that completed.
μ: a marking.
$level$: level on state tree.

```
Begin
    μ = μ ∪ μ_i
    if type_i = JOIN
        (c', μ') = BuildNewTree(route_j[level + 1], j, a, k, μ, level + 1)
        if difBelow = FALSE
            if μ'_i ≠ μ_i
                if hld ≤ level
                    difAbove = TRUE
                i' = BuildSubTree(i, difAbove, 1, μ')
                difBelow = TRUE
                return (i', μ' − μ'_i)
            else return (i, μ' − μ'_i)
        else
            if hld > level
                difAbove = FALSE
            i' = BuildJoinNode(i, c', difAbove, μ', route_j[level + 1])
            return (i', μ' − μ'_i)
    if type_i = REPLICATE
        (c', μ') = BuildNewTree(route_j[level + 1], j, a, k, μ, level + 1)
        if difBelow = FALSE
            if μ'_i ≠ μ_i
                if hld ≤ level
                    difAbove = TRUE
                i' = BuildSubTree(i, difAbove, 1, μ')
                difBelow = TRUE
```

```
                return (i', μ' − μ'_i)
            else return (i, μ' − μ'_i)
        else
            if hld > level
                difAbove = FALSE
            i' = BuildReplicateNode(i, c', difAbove, μ', route_j[level + 1])
            return (i', μ' − μ'_i)
    if type_i = SAN
        μ' = ExecuteSAN(a, μ)
        if μ'_i ≠ μ_i
            P = Differ(μ', μ)
            hld = FindHighestLevelOfDiff(P)
            i' = BuildNewLeafNode(i, P, μ', k)
            difBelow = TRUE
            return (i', μ' − μ'_i)
        else
            difBelow = FALSE
            return(i, μ' − μ'_i)
End.
```

Procedure V.3 (**BuildNewLeafNode**) directly takes 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 projected markings 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 sampled completion times, within the compound event, for the activity with the earliest sampled completion 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. These include moving a sampled 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.

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

Procedure V.3 BuildNewLeafNode(j, P, μ', k)

(Returns a leaf for the new state tree.)

Parameters :

j: leaf node on original state tree.

P : set of places that had a change in marking.

μ': the new projected marking on the places at j and nodes above.

k : the component of the compound event in which the activity completes.

Begin

$j' =$ **CreateNode**(j)

$A =$ **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, μ')

else

$e'_a = e'_a \cup t_k$

$e_a = e_a - \{t_k\}$

else

$e'_a = e'_a \cup$ **Generate**(a, 1, μ')

$E_{j'} = E_{j'} \cup \{e'_a\}$

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, μ')

else

$e'_a = e'_a \cup \{t_k\}$

$e_a = e_a - \{t_k\}$

$E_{j'} = E_{j'} \cup \{e'_a\}$

End.

Procedure V.4 (**BuildSubTree**) copies a subtree and does future events list management. It performs basically the same operations as Procedure V.3, when the difference between the new marking and the old marking is only 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 node-by-node while recursively traversing the subtree.

Every operation on a compound event is done n times, where n is the number of replica submodels represented by the leaf in the new state tree if no "merging" of leaves occur later in Procedure **BuildReplicateNode** (to be explained later). 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 only checked 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 above and at the level at which the procedure was called on the route to the leaf associated with the activity that completed. The root node of the new subtree is returned.

Procedure V.4 BuildSubTree(i, *difAbove*, n, μ')

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

Parameters :

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.
μ': 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*, $n * n_{i,c}$, μ')
$n_{i',c'} = n_{i,c}$
$C_{i'} = C_{i'} \cup \{c'\}$
if $type_i = JOIN$
$C_{i'} = \emptyset$
for each $c \in C_i$
$c' =$ **BuildSubTree**(c, *difAbove*, n, μ')
$n_{i',c'} = 1$
$C_{i'} = C_{i'} \cup \{c'\}$
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 procedure described by Procedure V.5 (**BuildJoinNode**) creates a join node on the new state tree. It makes use of Procedure V.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*). 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 on the route to the leaf associated with the activity completion. A join node for the new state tree is returned.

Procedure V.5 BuildJoinNode(i, c', *difAbove*, μ', *oldChild*)

(Returns a join node for the new state tree)

Parameters :
i: a node on state tree.
c': the child that will go in place of old child.
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.
μ': the new projected marking on places at i or at nodes above.
oldChild: the child subtree where the activity completed.

Begin
  $i' =$ **CreateNode**(i)
  $C_{i'} = \emptyset$
  for each $c \in C_i$
    if $c \neq oldChild$
      $c'' =$ **BuildSubTree**(c, *difAbove*, 1, μ')
      $n_{i',c''} = 1$
      $C_{i'} = C_{i'} \cup \{c''\}$
    else
      $n_{i',c'} = 1$
      $C_{i'} = C_{i'} \cup \{c'\}$
  return (i')
End.

Finally, Procedure V.6 (**BuildReplicateNode**) 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 **BuildJoinNode**, 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 V.6 BuildReplicateNode(i, c', *difAbove*, μ', *oldChild*)

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

Parameters :
i: a node on state tree.
c': the child that will go in place of old child.
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.
μ': the new marking.
oldChild: the child subtree where the activity completed.

```
Begin
    i' = CreateNode(i)
    C_i' = ∅
    found = FALSE
    for each c ∈ C_i
        if c ≠ oldChild
            if CompareSubTree(c, c') = NOT_SAME_MARKING
                c'' = BuildSubTree(c, difAbove, n_i,c, μ')
                n_i',c'' = n_i,c
                C_i' = C_i' ∪ {c''}
            else
                n_i',c' = n_i,c + 1
                MergeEvents(c, c')
                C_i' = C_i' ∪ {c'}
                found = TRUE
        else
            if n_i,c > 1
                c'' = BuildSubTree(c, difAbove, n_i,c − 1, μ')
```

$n_{i',c''} = n_{i,c} - 1$
$C_{i'} = C_{i'} \cup \{c''\}$
if $found$ = FALSE
$n_{i',c'} = 1$
$C_{i'} = C_{i'} \cup \{c'\}$
return (i')
End.

A Example State Generation

We now illustrate how the procedures just presented can be used to generate new state trees and future events lists. The LAN model, composed of three replicas of the station submodel of Figure 1 joined with the network submodel of Figure 2, is taken as an example. We assume an initial marking with one token in places A and C, and zero tokens in all other places. The state tree for the composed model is as given in Figure 6. 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.

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 $n = 3$ and μ being 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 sampled completion times for each activity of this submodel enabled in marking μ. The activities enabled are $arrival$ and $access$. These sampled completion times are inserted in compound events with the 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, n is now equal to 1 and μ is the projected marking on the places of submodel *NETWORK*. Ac-

tivity *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 7.

Assume now that the earliest sampled completion time is t_{123}. This is a completion time for the activity of type *access* in one of the replicas 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 sampled 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 to the leaf. At this 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 upon the completion of *access*. Then a marking (μ') is chosen probabilistically from this set to be the new marking for the third replica submodel. In this case, there is only one possible next marking, which is $\mu(channel)$=1, $\mu(A)$=0, $\mu(B)$=1, and $\mu(C)$=0.

Since there is a difference between the old marking and the new marking in places at the leaf node, a set 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 thus be created for *arrival* and the sampled 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 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 empty 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 third sampled completion time (t_{123}) deleted since the activity *access* is no longer enabled in μ'.

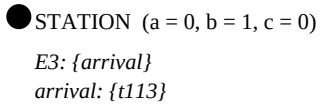


Figure 10: Node for Station In New Marking

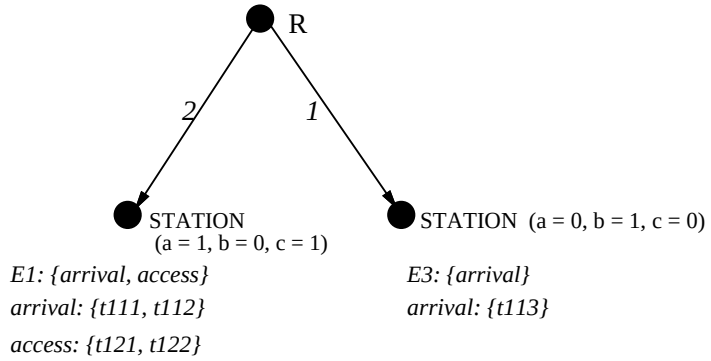


Figure 11: Replicate Node for New State Tree

Figure 10 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 an iteration through the set of child nodes. **BuildSubTree** is called to build the subtrees for the new replicate node. This procedure finds the projected marking for the submodels represented by the other leaf node. 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. The set of activities connected to the places where a change in marking occurred is thus generated. 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. Both leaves are then inserted in the set of children. Figure 11 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

been processed 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*. The marking of *channel*, in this submarking, is now 1.

The set of activities for the *NETWORK* submodel connected to places that changed in marking consists of the single element *prop_delay_intra*. This activity is in $EN_{\mu'}$, so a new event is created for the activity, 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 11 node. When **BuildSubTree** is finished, we have the state tree and future events as in Figure 8.

VI Implementation and Conclusions

The example of a state generation just presented, although simple, is sufficient to illustrate the advantages of the new procedures over the traditional method (as in procedure III.1) 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 be the same number as in the method presented in this paper, since reactivations require that all activities which were enabled and remain 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 replicas 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 of the three 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. Even for a model with a small degree of replication (3 times), there is a significant 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.

The procedures described in the previous section of this paper have been implemented, and are part of a larger performance-dependability evaluation package known as *UltraSAN* [3]. In addition to implementing the simulation procedures described in this paper, *UltraSAN* provides for analytic (numerical) solution of SAN models when activity time distributions are exponential, and the state space is not too large. In this case, a reduced base model [17, 18] is constructed, and this is solved numerically using known Markov process solution techniques. LU-decomposition and Successive Over Relaxation are used for steady-state solution, and uniformization is used for transient solution.

In *UltraSAN*, a model of a system consists of a hierarchical (or composed) SAN-based reward model, of the type described in Section II. Input to UltraSAN is graphical, via X-window based interfaces. Model specification is done via three editors: sanedit, compedit, and varedit. SANs are input to the system by drawing their constituent components, and supplying the necessary attributes for each component using sanedit. Compedit is used to create a composed SAN-based reward model of the system being studied, using replicate operations, join operations, and the SAN models created earlier using sanedit. Finally, varedit is used to specify the variables desired, and, when doing simulation, the desired relative confidence interval width and level. The iterative batch means method is then used to obtain steady-state solutions and the iterative replication method is used to obtain transient solutions using the procedures described in this paper as the state change mechanism.

The implementation of the simulation procedures is in C, and is roughly 10,000 lines of code. The implementation works well, and has been in use at several industrial and academic sites for over two years. It has been used in production evaluation studies in the areas of computer and telecommunication networks [14, 15], multiprocessor systems [4], system software, manufacturing systems, and dependable computing [13]. The models that have been solved have been large and varied, including composed models consisting of approximately 150 SANs, and many-level-deep composed models. These studies show that the procedures are sound and can produce results for large composed SAN-based reward models.

VII Acknowledgment

We would like to acknowledge the hard work and support of Doug Obal, member of the *UltraSAN* development team. His help in implementing the procedures described in this paper was invaluable in proving their worth, and making them useful to others. We would also like to thank the reviewers for their helpful comments on both the presentation and content of the paper.

REFERENCES

[1] G. Chiola, "Simulation framework for timed and stochastic Petri nets," *Int. Journal Computer Simulation*, 1, 1991, pp. 153-168.

[2] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "On well-formed coloured nets and their symbolic reachability graph," *Proc. 11th Int. Conf. on Appl. and Theory of Petri Nets,* Paris, France, June 1990.

[3] J. Couvillion, R. Freire, R. Johnson, W. D. Obal II, M. A. Qureshi, M. Rai, W. H. Sanders, and J. E. Tvedt, "Performability modeling with *UltraSAN*," *IEEE Software*, vol. 8, no. 5, Sept. 1991, pp. 69-80.

[4] E. Hokens and A. Louri, "Performance considerations relating to the design of interconnection networks for multiprocessing applications," submitted for publication, ICPP '93, August 1993.

[5] J. C. Kemeny and J. L. Snell, *Finite Markov Chains*, Princeton: D. Van Nostrand Co., Inc., 1969.

[6] M. Ajmone Marsan and G. Chiola, "Construction of generalized stochastic Petri net models of bus oriented multiprocessor systems by stepwise refinements," *Proc. 2nd International Conf. on Modeling Techniques and Tools for Performance Analysis*, Sophia Antipolis, France, June 1985.

[7] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application", in *Proc. International Workshop on Timed Petri Nets*, pp. 106–115, Torino, Italy, July 1985.

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

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

[10] J. D. Noe and G. J. Nutt, "Macro E-net representation of parallel systems," *IEEE Transactions on Computers,* C-31(9), August 1973, pp. 718-727.

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

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

[13] W. H. Sanders and L. M. Malhis, "Dependability evaluation using composed SAN-based reward models," in *Journal on Parallel and Distributed Computing,* special issue Petri Net Models of Parallel and Distributed Computers, vol. 15, no. 3, July 1992, pp. 238-254.

[14] W. H. Sanders, L. Kant, and A. Kudrimoti, "A modular method for evaluating the performance of picture archiving and communication systems," submitted for publication, *Journal of Digital Imaging.*

[15] W. H. Sanders, R. Martinez, Y. Alsafadi, and J. Nam, "Performance evaluation of a structured PACS using stochastic activity networks," to appear in *IEEE Transactions on Medical Imaging.*

[16] 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, pp. 74–84.

[17] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," in *Proc. Third International Workshop on Petri Nets and Performance Models*, Kyoto, Japan, Dec. 11-13, 1989, pp. 74–84.

[18] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *Dependable Computing for Critical Applications, Vol 4: of Dependable Computing and Fault-Tolerant Systems* (ed., A. Avizienis and J. Laprie), Springer-Verlag, 1991, pp. 215–237.

[19] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," in *IEEE Journal on Selected Areas in Communications,* special issue on *Computer-Aided Modeling, Analysis, and Design of Communication Networks*, vol. 9, no. 1, Jan. 1991, pp. 25-36.

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