UML2SAN: TOWARD A NEW SOFTWARE PERFORMANCE ENGINEERING APPROACH

Ihab Sbeity[1], Mohamed Dbouk[1] and Ilfat Ghamlouche[2]

[1]Faculty of Sciences, Lebanese University, Lebanon
[2]Faculty of Economical Sciences and Management, Lebanese University, Lebanon

ABSTRACT

Software Performance Engineering (SPE) has recently considered as an important issue in the software development process. It consists on evaluate the performance of a system during the design phase. We have recently proposed a new methodology to generate a Stochastic Automata Network (SAN) model from a UML model, to consequently obtain performance predications from UML specifications. In this paper, we expand our idea to cover more complex UML models taking in advantage the modularity of SAN in modeling large systems. A formal description of the generation process is presented. The new extension gives rise to a serious approach in SPE that we call UML2SAN.

KEYWORD

Software engineering, Performance software engineering, UML, Stochastic Automata Network.

1. INTRODUCTION

Software engineering has traditionally focused on functional requirements and how to build software that has few bugs and can be easily maintained. Most design approaches include non-functional requirements among the elements of the analysis of a system, but little attention has usually been paid to how these requirements can be dealt with through the development life-cycle [9].

Over the two last decades, the performance analysis of software systems during the design process is becoming widely suitable. The benefit comes from getting quantitative predications about the system before being implemented. That reduces the possibility of unexpected shortcoming on the system functionality. .Moreover, an important methodology, initially introduced by Smith [11], starts today to cover a large place in the software engineering area. This methodology, called *Performance Software Engineering* (SPE), consists essentially on introducing some techniques allowing obtaining performance predictions of the system basing on the design model.

Several approaches have been introduced to provide SPE techniques. Some of them propose to derive from a UML (Unified Modeling Language) model a separate performance model [5, 8,

DOI : 10.5121/ijsea.2012.3304

10]. In addition to the classification of UML as a universal modeling language widely used by designers, it is apparently possible to attach performance analysis tools to UML notations in a relatively straightforward manner. Our recent work in [10], initiates a preliminary methodology to derive from UML a Stochastic Automata Network (SAN) model, a Markov-based model used to generate performance predictions. The SAN formalism [7] is usually quite attractive when modeling a system with several parallel cooperative activities. In addition, SAN permits to represent a system in modular way. A SAN model is a state-transition graph having a strong likeness with the UML state-chart diagram.

The purpose of this work is to make a step forward toward proposing a general heuristic to derive a SAN model from complex UML model. In [10], our methodology is based on the UML state-diagram, with transitions' edges only labeled by *unconditional* triggers and actions. In this paper, we show how to deal with *conditional* triggers. In addition, we show the impact of the collaboration diagram in the derivation process. The demarche is also informally presented, however, this works enlarge the possibility to propose a formal demarche.

The rest of this paper is organized as follows. Section 2 recalls the definition of the SAN as modular formalism. Section 3 presents a UML producer/consumer example with conditional triggers in the stat-chat diagram. This section shows the essential UML diagrams need for the derivation of the SAN model. Section 4 explores how the UML model maps into SAN basing on our case study. Section 5 concludes our paper and describes our ongoing works.

2. STOCHASTIC AUTOMATA NETWORKS

Stochastic Automata Networks (SAN) is a structured Markovian formalism, i.e., it describes continuous-time Markovian models not as a flat system, but as a structured (modular and organized) collection of subsystems. The basic modeling principle of SAN is to describe a whole system by a collection of subsystems with an independent behavior and occasional interdependencies. Each subsystem is described as a stochastic automaton, i.e., an automaton in which the transitions are labeled with probabilistic and timing information. Hence, one can always build a continuous-time stochastic process related to SAN [2, 12].

The *global state* of a SAN model is defined by the cartesian product of the local states of all automata. There are two types of events that change the global state: local and synchronizing events. ***Local events*** change the SAN global state passing from a global state to another that differs only by one local state. ***Synchronizing events*** can change simultaneously more than one local state, i.e., two or more automata can change their local states simultaneously. In other words, the occurrence of a synchronizing event forces all concerned automata to fire a transition corresponding to this event.

Each event is represented by an identifier and a rate of occurrence, which describes how often a given event will occur. Each transition may be fired as result of the occurrence of any number of events. In general, non-determinism among possible different events is dealt with according to Markovian behavior, i.e., any of the events may occur and their occurrence rates define the relative frequency with which each of them will occur. However, if, from a given local state, the occurrence of a given event can lead to more than one state, then an additional routing probability

must be informed to each possible destination state. The absence of routing probability is tolerated if only one transition can be fired by an event from a given local state.

The other possibility of interaction among automata is the use of ***functional rates***. Any event occurrence rate may be expressed by a constant value or a function of the state of other automata. By contrast with synchronizing events, functional rates are one-way interaction among automata, since it affects only the automaton where it appears.

Figure 1 presents a SAN model with two automata, one synchronizing event (*e2*) with a constant rate, and four local events, being three with constant rates (*e3*, *e4* and *e5*) and one with a functional rate (*e1*). In this model the rate of the event *e1* is a functional rate *f* semantically explained below, and described inside Figure 1 using the SAN notation. The interpretation of such a function can be viewed as the evaluation of an expression of non-typed programming languages, e.g., C language, where each comparison is evaluated to value 1 (true) or value 0 (false).

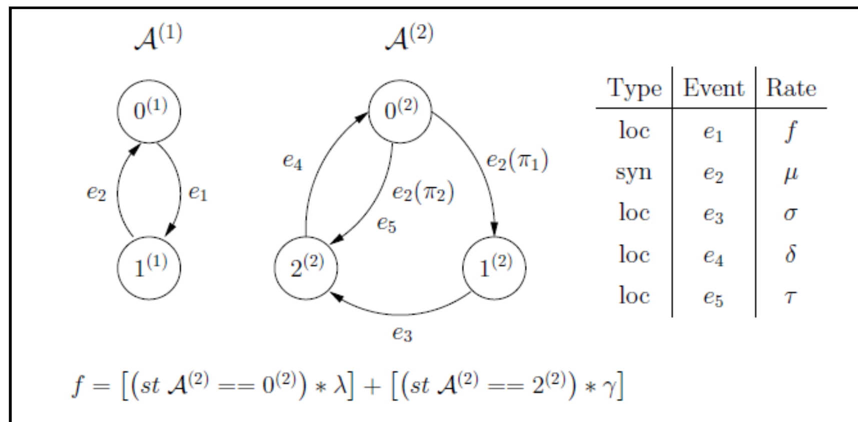


Figure 1 : Example of a SAN model

$$f = \begin{cases} \lambda \text{ if automaton } \mathcal{A}^{(2)} \text{ is in the state } 0^{(2)} \\ 0 \text{ if automaton } \mathcal{A}^{(2)} \text{ is in the state } 1^{(2)} \\ \gamma \text{ if automaton } \mathcal{A}^{(2)} \text{ is in the state } 2^{(2)} \end{cases}$$

The firing of the transition from states $0^{(1)}$ to $1^{(1)}$ occurs with rate λ if automaton $A^{(2)}$ is in state $0^{(2)}$, or γ if automaton $A^{(2)}$ is in state $2^{(2)}$. If automaton $A^{(2)}$ is in state $1^{(2)}$, the transition from states $0^{(1)}$ to $1^{(1)}$ does not occur (rate equal to 0). It is important to observe that the use of functions allows a compact and flexible way to describe in one single event (local or synchronized) alternative behaviors [2]. *The advantage of using functions will be obviously clear in our demarche of derivation a SAN model from UML model.*

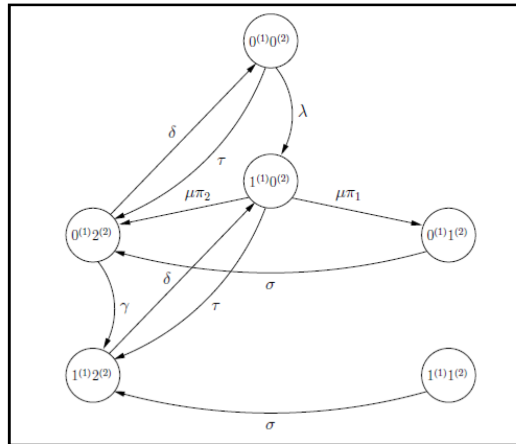


Figure 2: Equivalent Markov chain

Figure 2 shows the equivalent Markov chain model of the SAN model in Figure 1. Assuming the state $0^{(1)}0^{(2)}$ is an initial state, only 5 of the 6 states in this Markov chain model are reachable. In order to express the reachable global states of a SAN model, it is possible to define a (*reachability*) function. The reachable states could also be computed by analyzing all possible firing sequences, starting from a given reachable initial state. For the model in Figure 1, the reachability function excludes the global state $1^{(1)}1^{(2)}$, thus:

$$Reachability \; = ! \left[\left(st\, \mathcal{A}^{(1)} == 1^{(1)} \right) \&\& \left(st\, \mathcal{A}^{(2)} == 1^{(2)} \right) \right]$$

One of greatest advantages of the SAN formalism in comparison with straightforward Markov chain, and even other structured formalism is to have since its first definition [7] a compact form to store the infinitesimal generator of the equivalent Markov chain. Instead of storing an (usually) huge matrix, the SAN formalism defines a storage based on a tensor formula of considerably smaller matrices. Tensor, or Kronecker, algebra [3,4] is defined as a set of multi-dimensional structures (tensors) and algebraic operations. It usually allows the very compact description of quite large and complex matrices. Also, computation can be handled without ever generating extensively the equivalent Markov chain.

3. A UML PRODUCER/CONSUMER MODEL

The Unified Modeling Language (UML) [6] is a graphically based notation, which is being developed by the Object Management Group as a standard means of describing software oriented designs. It contains several different types of diagram, which allow different aspects and properties of a system design to be expressed. Diagrams must be supplemented by textual and other descriptions to produce complete models. For example, a use case is really the description of what lies inside the ovals of a use case diagram, rather than just the diagram itself.

Our concern in the approach we propose is basically UML models where the use of time has significance. In fact, it is evident to say that the need to predict performance is applied only to *time-driven* application, i.e. applications where objects actions are triggered by time progress.

Thus, here we present an example of such time-driven applications which describes a producer/consumer model. The model we propose is a more detailed extension of the model seen in [8]. In our model presentation we only focus of diagrams that describe the structure and the behavior of the system, i.e. the class diagram (section 3.1), and collaboration and state-chart diagrams (section 3.2).

3.1. The class diagram

A UML class model defines the essential types of object available to build a system; each class is described by a rectangle with a name. This can be refined by adding compartments below the name which list the attributes and operations contained in each instance of (object derived from) this class. Classes are linked by lines known as *associations* which indicate that one of the classes knows about the other. The direction of this knowledge is known as the *navigability* of the association. In an implementation an association typically corresponds to one class having a reference variable of the type of the other class. Sometimes navigability has to be two ways, but it more often one way. This can be shown by adding arrow head to the end(s) of the association [8].

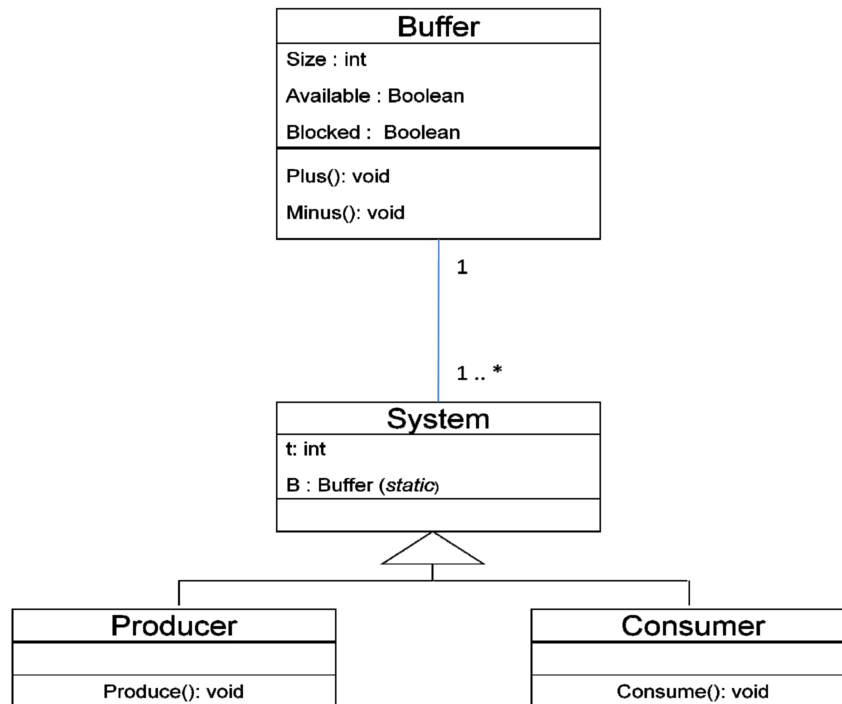


Figure 3: Producer/Consumer model - the class diagram

The class model of our producer/consumer application is presented in Figure 3. We assume a system with two kinds of processes, producer and consumer that communicate via a buffer.
A producer spends time producing a message before sending it to the buffer due to the method *Plus()*.

A consumer picks a message from the buffer, via the method *Minus()*, and then spends some time consuming it.

The variable *t* of the super-class *System* represents the producing, respectively consuming, time of a producer, respectively a consumer. Of course, the value of this variable may be different for each process (producer and consumer).

The buffer has some capacity, described by its variable *Size*; when the buffer is not full, this implies that it may to receive new message from the producer. The value of the attribute *Blocked* is set to true when the buffer is full. On the other hand, the attribute available is set to false when the buffer is empty, meaning that the consumer cannot pick messages from the buffer.

We also assume the existence of the two methods *setAvailable* and *setBlocked* that gives values to the attributes *Available* and *Blocked* respectively. These two methods are important in our demarche intending to derive a SAN model as it will be discussed in section 4.

3.2. Collaboration and State-chart diagram

Collaborations describe the way the objects interact *externally*. Statescharts describe how instances of classes behave *internally*. In a complete design they should provide between them a full description of how the system works.

3.2.1. Collaboration Diagram

Collaborations are collections of objects, linked to show the relevant associations between their classes. Here "time" is not represented explicitly. Instead the emphasis is on showing which objects communicate with which others.

Figure 4 represents the collaboration diagram of producer/consumer model. It acts of a communication between a family of *M* producers and another family of *N* consumers via one buffer. In the model of Figure 4, we suppose that there are (*M=3*) producers and (*N=5*) consumers.

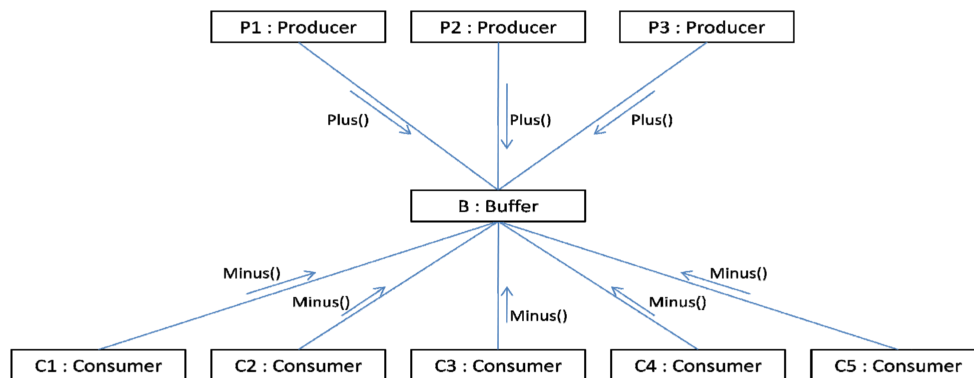


Figure 4: Producer/Consumer model - the collaboration diagram

As it was mentioned before, the collaboration diagrams show how objects are interacting with each other. Each produce communicates with the buffer via the *Plus()* message, and each consumer communicates with the same buffer via the *Minus()* message. Each communication message is the result of the corresponding method previously described in the class diagram.

3.2.2. State-chart Diagram

On the other hand, state-charts describe how objects behave internally. In a complete design they provide a full description of how the system works. We insert the state-chart for each object into its box in the collaboration. At any point in the lifetime of this system, each object must be in one, and only one, of its internal states. Each time a message (event) is passed, it may cause a state change in the receiving object and this may cause a further message to be passed between that object and another with which it has an association.

Figure 5 presents the state-chart diagram of our producer/consumer model.

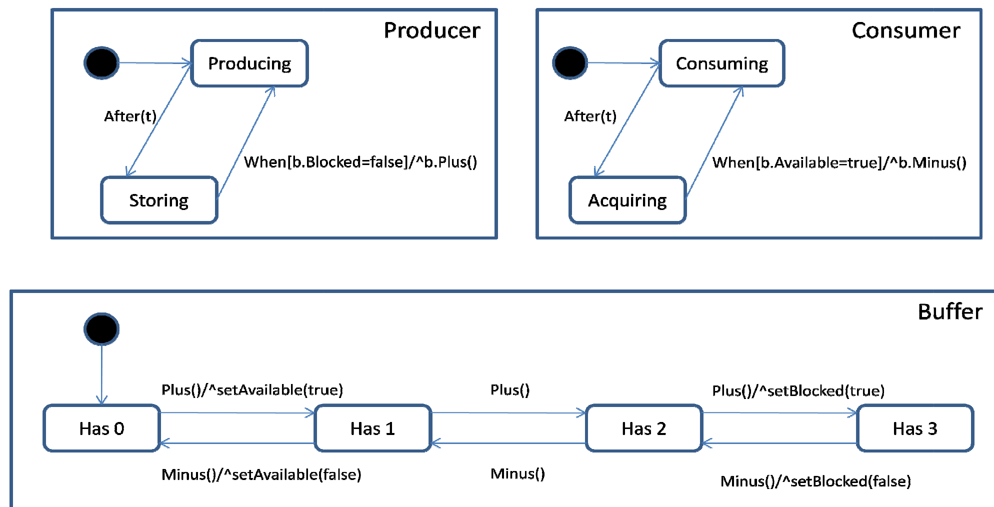


Figure 5: Producer/Consumer model - the state-chart diagram

Each chart describes the internal behavior of the concerned object. Briefly, a chart is composed of states, transitions, triggers and actions.

States are shown here as rounded rectangles; the initial state as a black filled circle.
Transitions are the arrows between states, labeled with a trigger.

Remark that states and transitions of UML state-chart are similar to the SAN's states and transitions.

Triggers represent the reason for an object to leave one state and follow the corresponding transition to another state; typically a trigger is an incoming message shown by one of the two usual ways for messages:

- A change in a condition; represented by the word *when* followed by the condition enclosed in brackets. This kind of triggers is called conditional triggers.
- An elapsing of time – shown by the word *after* enclosing the duration. This is an unconditional trigger.

Triggers may also invoke actions.

Actions are prefixed with a forward slash and carried out before entering the new state. These actions may be local method of the object where the trigger is fired, or a method of another object. In the second case, the invoked action represents then a trigger in the chart corresponding to the concerned object.

In order to simplify the process of derivation in section 5, we propose some definitions.

Definition 1

*A trigger is called of type **T1**, if it does not fire an action.*

Definition 2

*A trigger is called of type **T2**, if it fires one or more actions.*

Back to Figure 5, the behavior of each object type is described by a chart. For example, a producer alternates between two states: the state where it is producing a message and the state where it is storing the message in the buffer. The transition from the *Producing* state to the *Storing* state is labeled by the trigger *after(t)*, e.g. *t* is the class variable. This trigger does not fire any action, it is of type ***T1***. The trigger *when[b.Blocked = false]*, labeling the reverse transition, invoke the action *b.Plus()* in the chart corresponding to buffer. This trigger is of type ***T2***.

It is important to notice that triggers of type T1 look like *local events* in a SAN model. A trigger of type T2 seems resembling SAN synchronizing event.

Definition 3

*Let t be a trigger of type **T2**, we call Seq(t) the set of different action sequences that may be fired by t.*

Card(Seq(t)) is the cardinality of Seq(t) and it is equal to the number of sequences in Seq(t).

According to definition 3, consider the type T2 trigger *t = when[b.Blocked = false],* the set *Seq(t)*, is identified by taking into consideration the different sequence of actions that the trigger may fire. In our example, Seq(t) is composed of three sequences S1, S2 amd S3 such that:

Seq(t) = { S1: *when[b.Blocked = false]* → *b.Plus()* → *setAvailable(True)*
S2: *when[b.Blocked = false]* → *b.Plus()*
S3: *when[b.Blocked = false]* → *b.Plus()* → *setBlocked(True)* }

In fact, when the event *when[b.Blocked = false]* is invoked, it fires the action *b.Plus().* However the firing of this action may also fires another action depending on the state of the the state-chart *Buffer*. If the local state is Has0, *setAvailable(True)* is fired, if the local state is Has1 no other action is fired, and if the local state is Has2, the *setBlocked(True)* is fired. Card(Seq(t)) is equal to 3.

Moreover, states of the buffer represent the number of messages available in the buffer. The number of these states is equal to the buffer size (here, size = 3). Triggers labeling transitions of the chart buffer, i.e. *Plus()* and *Minus()*, are actions invoked by a producer or a consumer. We underline that the action *setAvailable()*, respectively *setBlocked(),* change the value of the Boolean variable *Available*, respectively *Blocked*, that affects the firing of the trigger *when[b.Available= true]* in the chart consumer, respectively the trigger *when[b.Blocked = false]* in the char producer. This means that at any time, we need to know the value of the variable in order to enable or disable the firing of the trigger. Recalling a SAN model, this phenomenon gives the impression to be similar to a functional rate that depends on the variable's value. We call such variables, i.e. which affect the firing of a trigger, ***affecting variables***.

The overall state of a system will be the combination of all the current internal state of its objects, plus the current values of any relevant attributes, i.e. *Blocked* and *Available*. As we mentioned before, intuitively, readers may sense a strong similarity to the stochastic automata networks behavior. A SAN automaton is a set of states, transitions and events. The global state of a SAN is a combination of the local states of its automata. The powerful point of mapping the state-chart diagram into a SAN model is that the mapping process will not produce fundamental changes in the graph structure, only some information needed to represent relevant attributes and the time spent in a state is required. That may help designers to better understand the SAN performance model which is an emphasized advantage of our approach.

4. Generating the SAN model

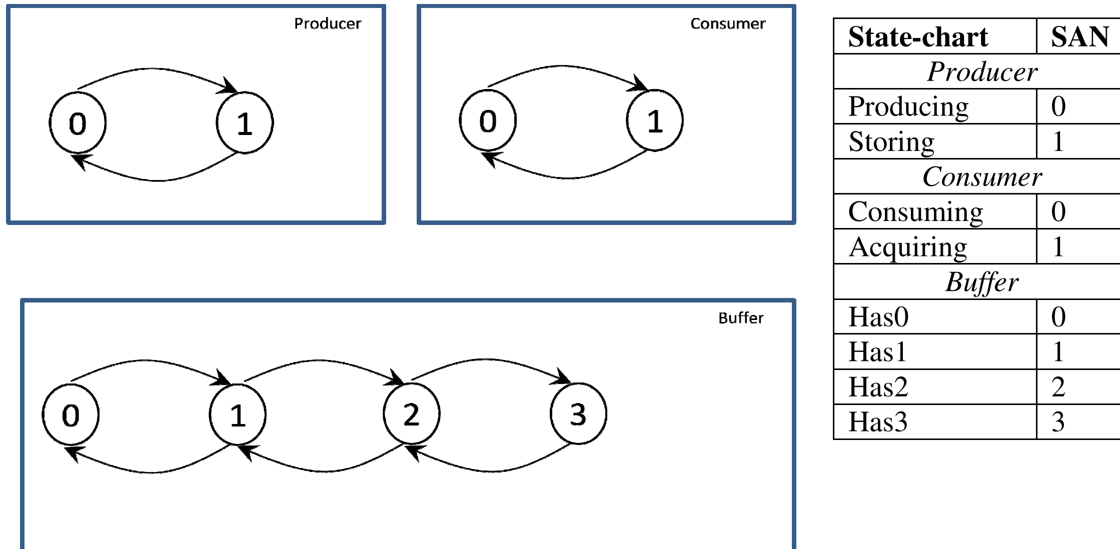
In the previous section, we highlighted the basic elements on which our generation process is principally based. In this section, we present how a SAN model is directly generated for the UML producer/consumer, giving a sense for a systematical generation process. The SAN automata are these corresponding to UML state-charts and these necessary to represent the *affecting variables* value. The generation process is composed of multiple steps. As it is shown in [10], it is important to see that there is an intuitive mapping of the UML state-chat into SAN. A state-chart automaton is mapped into SAN automaton (states and transitions). The state-chart events are translated as SAN events. However, there is still some work to achieve in order to take into consideration the affecting variables. In the following, we formally describe the procedure of the generation of the SAN model and we illustrate each step of the procedure basing on our producer/consumer model. In the following, each subsection represents a step in our generation process. Some of these steps are formally described; however others are presented basing on intuitive observations.

4.1. Step 1: States/transitions mapping

The first step is to create SAN automata that correspond to state-charts. Only states and transitions are created in this step. The labels of SAN transitions, i.e. events, are created in step 3.

Step 1: Statechart -- States/Transitions Mapping
For each automaton X of the UML state-chart, create a SAN automata Y as following: *Omit the initial state.* *For each state I_X in X, create the state I_Y in Y.* *For each transition in X from state I_X to state J_X, create a transition from I_Y to J_Y in Y.*

Applying the first step to our producer/consumer model gives raise to the following *intermediate incomplete* SAN graph. The table enclosed at right shows the correspondence between states of UML model and the new states of the *intermediate incomplete* SAN model.



State-chart	SAN
Producer	
Producing	0
Storing	1
Consumer	
Consuming	0
Acquiring	1
Buffer	
Has0	0
Has1	1
Has2	2
Has3	3

Figure 6: Generation Process - Step 1 – state-chart automata

4.2. Step 2: Affecting variables mapping

The second step of the generation process is related to *affecting variables*. Recall that the values of such variables are used as conditions for the firing of triggers in the state-chart diagram. Thus it is important to represent the modification of these values in order to enable/disable the firing of SAN events, as it will be shown in step 3.

Step 2: Affecting variables Mapping

For each affecting variable v in the UML model, create a SAN automaton Y as following:

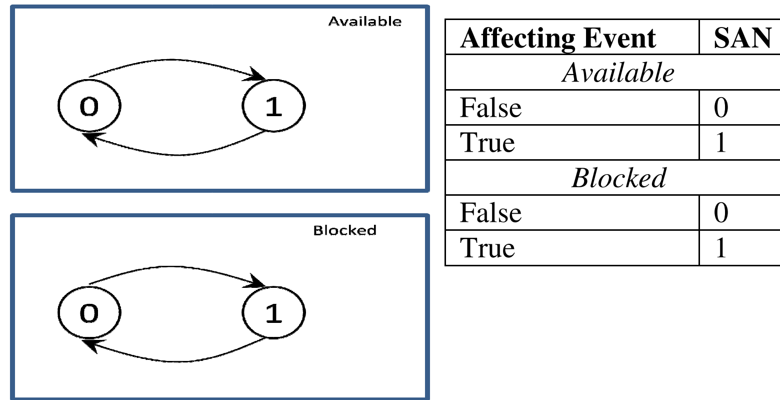
For each potential value k_v of v, create the state k_Y in Y.

For each potential value's modification from v= k to v =l (k # l), create a transition from k_Y tp l_Y in Y.

Indeed, building a SAN automaton that corresponds to an affecting variable is possible under some criteria. In fact, the number of states of the SAN automaton is equal to the potential values the variable may have. This implies that the number of potential values should be finite. Dealing with affecting variables with infinite value range is a serious challenge. An intuitive idea to approach this problem is to decompose the values range into two sub-ranges: the sub-range where the variable's value enables the firing of the trigger, and the sub-range where it disables the trigger's firing.

Here, we restrict our study to variables with finite potential values, i.e. Boolean variables.

The second step consists to map the potential values that an affecting variable may have into SAN states. In our producer/consumer model, there is two affecting Boolean variables, i.e. *Available* and *Blocked*. Applying Step 2, generate the following incomplete SAN automaton. Again, SAN events are not represented here as they will be generated in step 3.



Affecting Event	SAN
Available	
False	0
True	1
Blocked	
False	0
True	1

Figure 7: Generation Process - Step 2 - affecting variables

For each of the two Boolean affecting variables, an automaton is associated. The states of each automaton represent the different values the variable may have, i.e. either true or false. There is a transition from each state to the other state. In case of no Boolean affecting variable, from each state, we have to draw a transition to all other states, taking into consideration that the value of the variable may change without being aware of the new value. In addition, we need to identify the UML method that produces the change. We call this method ***affecting method***. In our model there is two affecting methods: *SetBlocked()* and *setAvailable()*, related respectively to the affecting variable *Blocked* and *Available*.

Affecting method should be predictable in the UML model because they will drive the transitions in the SAN automata related to affecting variables, as it will be described in step 3.

4.3. Step 3: Events Generation

The purpose of this step is to generate the list of events that drive the transitions of the SAN automata. As it is previously presented, there are two types of events in the SAN model: local event that affects only the state of one automaton and synchronizing event that can change simultaneously the state of more than one automaton. Each trigger of type ***T1*** of the UML state-chart will give raise to a SAN local event (Step 3.a). One Trigger *t* of type ***T2*** of the state-chart will give raise to several SAN synchronizing events according to the cardinality of *Seq(t)* (refer to Definition 3).

Step 3.a : Local events Generation

For each trigger t of the UML state-chart, create a SAN event e as following:
For each automaton X of the UML state-chart (Y is SAN automaton created in step 1)
For each transition in X from state I_X to state J_X, labeled by t
Add e to the transition from state I_Y to state J_Y
Add e to the transition from state I_Y to state J_Y

Step 3.b : Synchronizing events Generation

For each trigger t of the UML state-chart
For each sequence of actions S in Seq (t)
create a SAN event e_S as following:
For each automaton X of the UML state-chart and Y its corresponding SAN automaton created in step 1)
For each transition in X from state I_X to state J_X, labeled by t
Add e_S to the transition from state I_Y to state J_Y

For each action a in S
If a is an affecting method then
Add e_S to the corresponding automata (of the affecting variable)
Else
For each automaton X of the UML state-chart and Y its corresponding SAN automaton created in step 1)
For each transition in X from state I_X to state J_X, labeled by a
Add e_S to the transition from state I_Y to state J_Y

Applying the third step to our producer/consumer example gives raise to the graph of figure 8. The automata presented in this figure are the set of automata created in step 1 and 2.

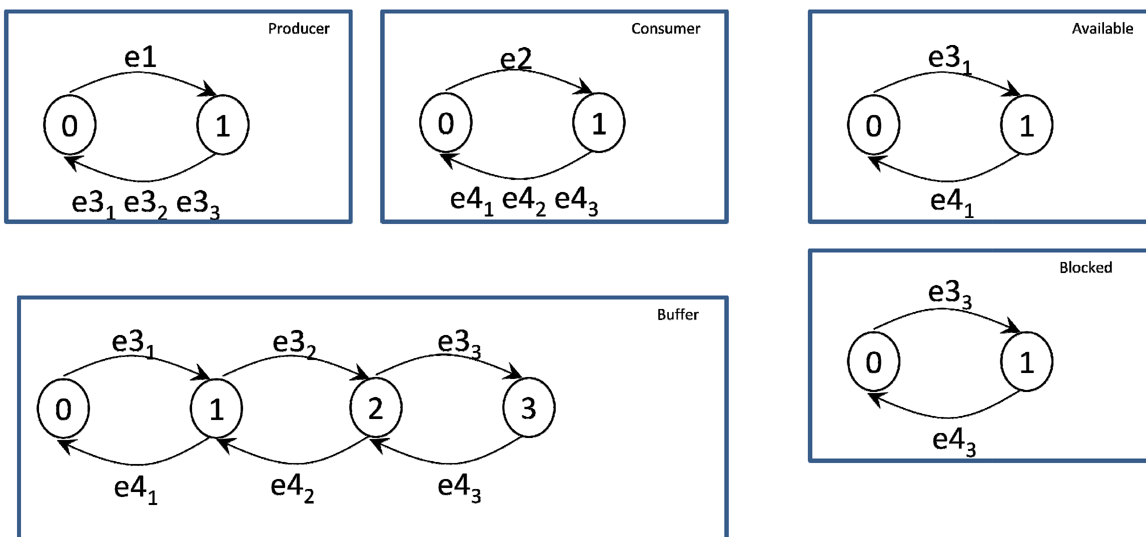


Figure 8: Generation Process - Step 3 - events generation

For example, the local event *e1* of the SAN automaton *Producer* is the corresponding of the trigger *after(t)* of the Producer state-chart. This event is generated according to Step 3.a, as the trigger is of type **T1**. The trigger *t = when[b.Blocked = false]*, of type **T2**, gives raise for three SAN events, i.e. $e3_1$, $e3_2$ and $e3_3$. Recall that the set *Seq(t)* contains three different sequences of actions (see section 3.2.2). Each event corresponds to a sequence. For example, The event $e3_1$ corresponds to the sequence S1: *when[b.Blocked = false]* → *b.Plus()* → *setAvailable(True)*. This sequence involves methods in the state-chart *Producer*, i.e. *when[b.Blocked = false]*, and the state-chart *Buffer*, i.e. *b.Plus()* , and a method that invoques a change in the value of the affecting

variable *Available*. Thus, the event $e3_1$ synchronizes the three SAN automata Producer, Buffer and Available, and it labels the corresponding transitions.

4.4. Step 4: Events' rates

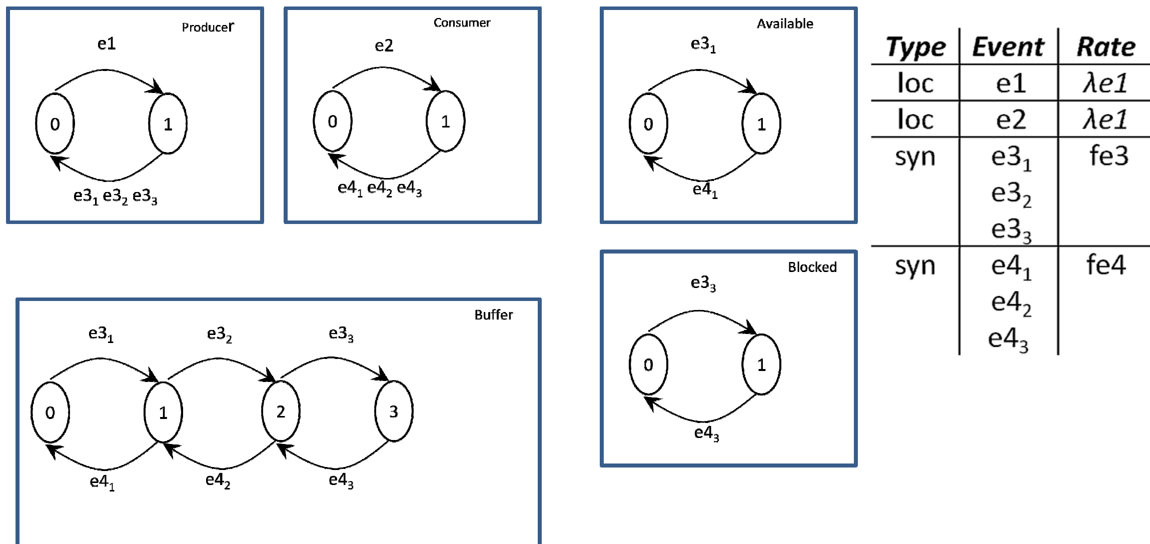
Basing on Step 3, it may be noticed that two categories of events may be underlined: events corresponding to the "when" methods, e.g. the event $e3_1$, and events that do not correspond to the "when" method, e.g. the event *e1*. Recall that the method *when* has a condition and it controls the occurrence of a sequence of methods.

For each event *e* that does not correspond to the *when* method, the rate λe is assigned. However, an event corresponding to the *when* method is assigned a functional rate. Let us reconsider the event $e3_1$ that corresponds to the sequence *when[b.Blocked = false]* → *b.Plus()* → *setAvailable(True)*. The event $e3_1$ should be eligible to fire only if the automaton Blocked is in its local state 0, i.e. the state that corresponds to the value False of the affecting variable Blocked. Thus the rate assigned to the event $e3_1$ is the function rate *fe3* described as following:

$$fe3 = \begin{cases} \boldsymbol{\lambda e3} & if\ automaton\ Blocked\ is\ in\ state\ 0 \\ \mathbf{0}\ Otherwise \end{cases}$$

It is obvious to see that the concept of functional rate is a powerful point of using stochastic automata network.

Finally, recall that events $e3_1$, $e3_2$ and $e3_3$ are basically related to the same trigger, they should be assigned the same rate, i.e. *fe3*. Thus, the typical SAN model corresponding to the UML Producer/Consumer model is given by figure 9.



Type	Event	Rate
loc	e1	λe1
loc	e2	λe1
syn	$e3_1$ $e3_2$ $e3_3$	fe3
syn	$e4_1$ $e4_2$ $e4_3$	fe4

Figure 9: Generation Process - Step 4 - events rates

4.5. Step 5: Effect of collaboration diagram

The SAN model presented in figure 9 is a complete SAN model in the sense it respects the formal definition of a SAN model. However, this SAN model does not yet correspond to the UML model as it does not take into consideration the presence of different instances of each UML class type. For example, the automaton Producer of figure 9 describes the state of only one producer; however, it is possible to have more than producer in the system.

As it has been shown, the collaboration diagram presents the interaction between all objects of a system; thus, it reflects the number of each instances of each class type acting in the system.

The last step of the generation process is to take into consideration the number of objects acting in the system basing on the UML collaboration diagram. For each object, a separate SAN automaton is created and which is a duplicate of the original automaton giving in step 4. For example, the producer/consumer, there are three Producer objects, and then three automata should be created as a duplicate of the automaton Producer of figure 9, as it is shown in figure 10.

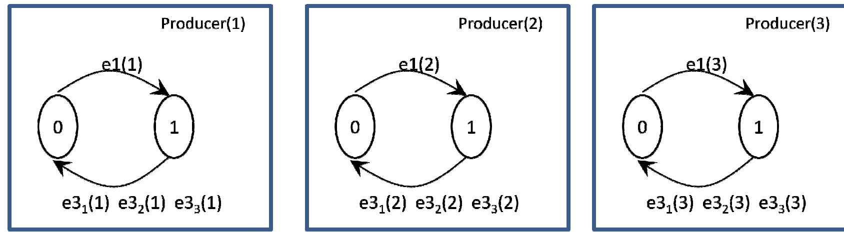


Figure 10: Generation Process - Step 5 - effect of collaboration diagram

In each of duplicate automaton, events type (local or synchronizing) is the same as original events type. For example, the event $e_1(1)$ is a local event of the automaton Producer(1). In addition, a synchronizing event synchronizes the same automata as the original event. For example, events $e3_1(1)$, $e3_1(2)$ and $e3_1(3)$ should also synchronize automata Buffer and Available, as the original automata $e3_1$, i.e. they replace $e3_1$ on the corresponding transitions label.
Moreover, each duplicate event has its own rate. Furthermore, if the original event has a functional rate, then the duplicate event should also have the same functional rate. For example the rate of the duplicate event $e3_1(1)$ is given by:

$$fe3(1) = \begin{cases} \boldsymbol{\lambda e3(1)} & if\ automaton\ Blocked\ is\ in\ state\ 0 \\ \mathbf{0}\ Otherwise \end{cases}$$

As a conclusion of our generation process, by combining the five steps, the SAN model corresponding to a UML model may now be systematically generated, and performance predictions may easily be analyzed using the PEPS software tool [1].

5. Conclusion

This paper is the continuity of our work that proposes a new methodology in the performance software engineering. We have presented a *formal* process that allows generating a stochastic

automata network model from a UML model. The generation process is systematic and can be easily implemented.

This work has double interests. The direct interest is that it proposes a new methodology that allows designers to predicate the performance of their applications before proceeding to the implementation phase. The indirect interest is that it offers a visual interface in order to construct SAN models by benefiting from UML tools facilities. In fact, SAN model can be constructed as a UML state-chart and the SAN model is then generated using our generation process.

It remains for us to implement the *UML2SAN* generation process in a tool. This is a perspective of our work. Exploring and testing more case studies is also an ongoing work.

REFERENCES

[1] L. Brenner, P. Fernandes, B. Plateau, and I. Sbeity. PEPS2007 - Stochastic Automata Networks Software Tool. QEST 2007: 163-164

[2] L. Brenner, P. Fernandes and A. Sales. The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations, in: 20th Annual UK Performance Engineering Workshop, Bradford, UK, 2004, pp. 48–60.

[3] M. Davio. Kronecker Products and Shuffle Algebra, IEEE Transactions on Computers C-30 (1981), pp. 116–125.

[4] P. Fernandes, B. Plateau and W. J. Stewart. Efficient descriptor - Vector multiplication in Stochastic Automata Networks, Journal of the ACM 45 (1998), pp. 381–414.

[5] P. King and R. Pooley. Using UML to Derive Stochastic Petri Net Models, UKPEW' 99, Proceedings of the Fifteeth UK Performance Engineering Workshop, 1999

[6] Object Management Group, Response to the OMG RFP for Schedulability, Performance, and Time, OMG Document ad/2001-06-14, June 2001, http://www.omg.org.

[7] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms, in: Proceedings of the 1985 ACM SIGMETRICS conference on Measurements and Modeling of Computer Systems (1985), pp. 147–154.

[8] R. Pooly and P. King. Using UML to derive stochastic process algebra models, Proceedings of the Fifteeth UK Performance Engineering Workshop, 1999

[9] R. Pooley. Software Engineering and Performance - a roadmap, in Finkelstein Ed The Future of Software Engineering, IEEE International Conference on Software Engineering, Limerick, July 2000, pp189-200.

[10] I. Sbeity, L. Brenner and M. Dbouk. Generating a Performance Stochastic Model from UML Specifications, International Journal of Computer Science Issues (January 2011), v. 8, issue. 1, pp. 13-21.

[11] C. U. Smith. Performance Engineering of Software Systems. Addition-Wesley, Reading, Massachusetts, 1990.

[12] W. J. Stewart. "Introduction to the numerical solution of Markov chains," Princeton University Press, 1994.