Performance Evaluation from Statecharts representation of Complex Systems: Markov Approach

Nandamudi Lankalapalli Vijaykumar¹, Solon Venâncio de Carvalho¹, Carlos Renato Lisboa Francês², Vakulathil Abdurahiman³, Ana Silvia Martins Serra do Amaral⁴

¹Laboratório Associado de Computação e Matemática Aplicada (LAC)
Instituto Nacional de Pesquisas Espaciais (INPE) – São José dos Campos, SP – Brazil

²Departamento de Engenharia Elétrica – Universidade Federal do Pará (UFPA)
Belém, PA – Brazil

³Divisão de Ciências de Computação – Instituto Tecnológico de Aeronáutica – ITA
São José dos Campos, SP - Brazil
(vijay, solon, anasil)@lac.inpe.br, rfrances@ufpa.br, rahiman@comp.ita.br

Abstract. While developing a system, manufacturing systems or computational systems, nowadays it is natural to study its behavior to understand the bottlenecks in order to improve it during the implementation. Mathematical modeling is one of the most used for this purpose. Usually evaluation is obtained either through simulation or through analytical approach. When considering analytical approach, Markov chains are used as they have well founded theory and at the same time they can be handled computationally to determine performance measures. A specification technique Statecharts have become quite popular in representing complex systems that are reactive by nature and manufacturing and computing systems fall under this category. Statecharts have their origin from state-transition diagrams to which notions of hierarchy, parallelism and synchronization are included. This paper discusses the process of modeling a complex system by using Statecharts as well as how a Markov chain is generated from this specification so that by making use of analytical approach, performance measures can be obtained.

1. Introduction

Generally speaking, specific features need to be considered for representing a system in a clear manner with the objective of evaluating the performance of the system. These features include hierarchy (systems consisting of subsystems or components), parallelism (more than one subsystem working or functioning at the same moment), and interdependence (synchronization and communication of subsystems). Systems with these characteristics have been called complex systems although other definitions are also available [Siljak, 1991]. Nowadays, performance information of complex systems has been an important topic for evaluating and analyzing their behavior before their release so that improvements can be implemented. Systems such as production job shops, flexible manufacturing systems, computers, communication networks, etc. fall under the category of complex systems and they are reactive by nature. Complex
systems, according to the definition used in this paper, are reactive. The main characteristic of a reactive system is its whole behavior is based on reaction to events received by internal and external media [Harel, 1987a]. These complex systems are heavily based on controls or events. The reaction to different kinds of events takes place in complex ways. In fact, a vast majority of the systems in day to day applications are rather reactive in nature [Harel, 1987b].

Due to their nature of randomness, complex systems for which performance is to be evaluated are, in general, represented by stochastic models. In these models, the system behavior can be represented by a state-transition diagram. In this diagram each state represents a physical state of various components of the system (idle, busy, failure, etc.) and the transitions among the states take place through the events that correspond, for example, to the termination of a job or to some disruption in the system such as a failure. In order to determine the performance analysis of such systems solutions make use of simulation techniques and analytical approaches. The focus of the paper is on analytical approach. Usually, analytical approach is associated with Markov chains as their theory is well established and moreover they can be handled computationally.

One more interesting factor related to these state-transition diagrams is that they are considered as Markov Chains, if the events that lead to transition among states follow an exponential distribution. The reason is that if a stochastic model is represented by a state-transition diagram and if it is a Markov chain, then it can be solved by using an analytical approach [Silva & Muntz, 1992]. This solution results in steady-state probabilities which are probability functions denoting the occupation of states during a certain period of time or in a long horizon. The question still remains as to how to represent a complex system in a high-level fashion so that performance measurements can be obtained.

The paper describes an approach of representing a complex reactive system by means of a high-level specification technique Statecharts and convert that representation into a Markov chain. Then, based on analytical solutions, steady-state probabilities are determined for the corresponding Markov chain and these are the basis for calculating performance measures. A software has been developed for this purpose and the name PerformCharts has been given to it. The paper is organized as follows: a very brief description of Statecharts is given in Section 2. Section 3 is devoted to showing an algorithm of how to convert a Statecharts representation into a Markov chain. An example is also presented to illustrate the algorithm. Section 4 describes algorithms for extensions such as probabilistic transitions in Statecharts and how to deal with representation of “memory” in Statecharts while generating a Markov chain. Some implementation details of PerformCharts as well as interface based on XML are briefly explained in Section 5. Section 6 ends the paper with some comments.

2. Statecharts
Statecharts are graphical-oriented and are capable of specifying reactive systems. They have been originally developed to represent and simulate real time systems [Harel, 1987a]. Moreover Statecharts have a defined formalism [Harel et al. 1987] and [Harel & Politi, 1998] and their visual appeal along with the potential features enable one to represent complex logic as a given reactive system’s behavior. They are an extension of
state-transition diagrams by including notions of hierarchy (depth), orthogonality (representation of parallel activities) and interdependence (broadcast-communication).

States are clustered to represent depth. It is possible to combine a set of states with common transitions into a macro-state also known as super-state. State refinement is achieved by means of XOR and AND decompositions. The former may be used whenever an encapsulation is required. When a super-state in a high level of abstraction is active, one (and only one) of its sub-states is indeed active. The latter is used to represent concurrency. In this case when a super-state is active, all of its sub-states are active at the same time. One more type is BASIC which means that there no more further refinements from this state.

In Statecharts global state of a given model is referred to as a configuration that is the active basic states of each orthogonal component. Definitions of each element and the main features are shown in [Harel, 1987a], [Harel et al, 1987], [Harel & Namaad, 1996] and [Harel & Politi, 1998].

By definition, when modeling a given system, there must always be an initial state also known as default state, which is the entry point. Another way to enter a system is through its history, i.e. when a system (or a sub-system) becomes active, the state most recently visited is activated. In order to use history, symbol H is provided. It is also possible to use the history all the way down to the lowest level (H^x) [Harel, 1987a]. Other approach to deal with the feature entry by history to influence only certain levels of the hierarchy can be done by using the appropriate number of H symbols applied to the desired level.

A brief discussion of events considered for performance evaluation [Vijaykumar, 1999] and [Vijaykumar et al, 2002a] is in order. Events are fundamental within a reactive system to change its behavior so that states (configurations) move to other states (configurations), i.e. transitions occur from one configuration to another. Within the scope of application to performance models, events have been classified into two categories: internal and external. Internal (or immediate) events are those that take zero time when they are enabled as the transition is fired immediately. Statecharts have such in-built events: true(condition), false(condition), entered(State), exit(State). The basic element action as an event that can influence some other orthogonal component is also considered as an immediate event. This means that whenever an event is associated as action, a reaction to this event is immediate. External events are stochastic events (where time between their activation and their occurrences follow a stochastic distribution) that have to be externally stimulated. In order to make the association of a Statecharts model with a Markov chain, events have to be stochastic, i.e. time between activation and occurrence of events follows a stochastic distribution. In particular, for Continuous-Time Markov Chains, this distribution has to be exponential. The original notation along a transition arc is given as event[condition]/action. The interpretation of this is that when an event is enabled and the associated condition is satisfied, only then the transition takes place by moving from one state to another. Once the transition is fired, action is performed by continuing the reaction moving from one state (in another parallel component) to another. An extension has been made to this notation by adding probability event[condition]/probability/action which will be described in Section 4.
3. On Constructing a Markov Chain from Statecharts representation

A Continuous-Time Markov Chain, consisting of transition rates among states is solved through numerical methods [Silva & Muntz, 1992] to determine the steady-state probabilities. Therefore, the problem is solved if the model represented in Statecharts generates a Markov chain that corresponds to the behavior of the specified model [Vijaykumar et al, 2002a].

Once the model is specified in a Statecharts representation, the first step is to check which events are to be triggered for the initial configuration. Recalling the categories of events explained earlier, internal (or immediate) events are the ones that are automatically triggered. As long as such events are active for the resulting configurations, reactions continuously take place by changing one configuration to another until a configuration is reached from which no more internal events can be triggered. The next step is to explicitly stimulate stochastic events. Based on the resulting configuration from internal events, the algorithm lists which stochastic events are to be triggered so that transitions are fired to generate new configurations. In order to make the association of a Statecharts model with a Markov chain, events have to follow an exponential distribution. By considering each stochastic event from the list, a reaction is performed, yielding a new configuration for each event of this list. Once a configuration is obtained, internal events, if enabled, are triggered, firing transitions to yield new configurations. In both the cases, actions also have to be considered if they are associated within a transition. In this case a reaction occurs immediately as action is considered as an internal event. This process continues until all the configurations have been expanded. The result of all this process is a list of a structure that contains a source configuration, stimulated stochastic event (along with its rate), and the target configuration. This information is a Markov chain with which steady-state probabilities can be determined.

Before showing an example, algorithms to deal with the reaction are shown. The main algorithm is responsible to generate a graph (state-transition diagram, a Continuous-Time Markov chain). The other two are auxiliary routines invoked by the main algorithm to react to immediate events and to stochastic events respectively.

generateGraph ( Statecharts stch )
{
  get initial configuration conf_init for stch
  // conf_set holds configurations that have to be expanded
  conf_init = reactToInternalEvents ( conf_init )
  add conf_init to conf_set
  WHILE (configuration conf belongs to conf_set AND there are events for conf that have to be stimulated) DO {
    make a list of external events for conf
    FOR (each external event ev from the list) DO {
      // conf_aux “remembers” the original source configuration to continue
      // stimulating the rest of the events of the list
      conf_aux = conf
      conf_new = reactToStochasticEvent ( conf_aux, ev ) // conf_aux reacting to ev
      WHILE (there are internal events) DO {
        conf_new = reactToInternalEvents ( conf_new )
      }
    }
  }
  // markov_chain holds the state-transition diagram consisting of
  // source configuration, target configuration and the stochastic event
In order to illustrate the whole process, from the specification to the generation of the Markov chain, with an example, consider a system with three parallel components that correspond to two machines (E1 and E2) and a supervisor (Supervisor) to repair any eventual failure of the machines. In case of failure of both the machines, a priority is provided to repair E1. This priority is described by the event \( \text{tr}[\text{in(B2)} \land \neg \text{in(B1)}] \) meaning that the conditions \textit{in state B2} (in(B2) – machine E2 is down) and \textit{not in state B1} (not in(B1) – machine E1 is not down) have to be satisfied in order to fire the transition to repair E2. More details about the events and conditions in Statecharts can be seen in [Harel et al., 1987]. The list of stochastic events include a1, r1, f1, s1 a2, r2, f2 and s2. Internal events are \( \text{tr}[\text{in(B1)}], \text{tr}[\text{in(B2)} \land \neg \text{in(B1)}] \). Actions c1 and c2, also considered as internal events, are triggered after the events s1 and s2 are executed. For example, once \( s1 \) is triggered, a transition from state C1 to WS (within the Supervisor component) is fired and this is followed by another transition associated to action c1.
moving from state B1 to state W1 (within E1 component). This model is shown in Figure 1.

![Statecharts representation of equipment with a repairer](image)

**Figure 1. Statecharts representation of equipment with a repairer**

When the specification is complete, a reaction takes place by first checking the internal events that may be enabled for the initial configuration and then by stimulating these enabled events. For example, internal events such as tr[in(X)], where X is a State, are automatically stimulated if the system’s initial configuration has an active State X. In the case of the example presented in Figure 1, no such internal events are active for the initial configuration. Therefore, taking the initial configuration (W1, W2, WS) the enabled events are stochastic events, a1 and a2. Therefore, these events will be stimulated to yield new configurations.

When the system is in the initial configuration and a1 is stimulated, the next configuration is (P1, W2, WS). Suppose that during the process of stimulating the events, a resulting configuration is (B1, B2, WS). In this case the active events are the so called immediate events tr[in(B1)] and tr[(in(B2) \land \neg in(B1))]. As noted earlier, these are the events that have to be checked and enabled before the stochastic events, i.e. even though there are enabled stochastic events, the immediate events (true(condition) and false(condition)) are the events that have to be automatically stimulated. The state-transition diagram is shown in Figure 2. Note that this diagram consists of only stochastic events that follow exponential distribution. The states and arcs with events following exponential distribution compose a Markov chain with which numerical methods can be applied to determine steady-state probabilities, the basis for calculating performance measures.

![Markov chain of the example in Figure 1](image)

**Figure 2. Markov chain of the example in Figure 1**
Given that Figure 2 is a Markov model, steady-state probabilities are determined by invoking the appropriate numerical methods to solve the Markov chain. As mentioned in the Introduction, a software called PerformCharts has been developed in C++ with all the extensions that will be discussed in the following section.

According to the algorithms and the example already shown, it should be clear that the translation from a Statecharts representation into a Markov chain is not a “blind” AND product. Otherwise, the Statecharts in Figure 1 would result into a 27-state diagram instead of 10-state diagram (Figure 2). However, one must consider the well known problem of state-space explosion that still exists and depends very much on the number of components and number of sub-states within each component. The advantage is that the Statecharts specification of the model is represented in a “cleaner” fashion instead of a web of arcs traversing among several hundreds of states.

Questions may arise in respect to the process of reaction. Especially, what happens when more than one stochastic event in separate components become active. As briefly explained in the algorithm, the process is somewhat put into a sequence. Once a configuration is obtained, it basically has a set of active states from each parallel component. Therefore, for this particular configuration a list of stochastic events that can be stimulated is created. This means that the events in this list can be triggered at the same time. However, the algorithm takes one event at a time and performs the reaction thus generating a new configuration. The order the events are placed in the list is the order they have been created during the specification phase. However, if the order of stimulation of the events in the list is changed, the resulting state machine should be the same. This can be seen in Figure 2. By taking any configuration, at which more than one exponentially distributed stochastic event exist, the resulting configurations are the same regardless of the order the events are stimulated. The authors expect that this is usually the case in most of the complex systems; however. If the logical behavior of a complex system depends on the order, then a time parameter (associated to an event) has to be associated or some type of alternative ways of synchronization must be made.

4. Useful extensions for representing Performance Models

Many a time there are situations in which an event might lead to more than one different destination states. Usually this is considered as a conflict as it is not determined which destination the transition should move to. However, it was already suggested and briefly mentioned in [Harel, 1987a] that, instead of considering such situation as non-deterministic, the conflict could be resolved (if required) by including a probabilistic aspect, thus creating “probabilistic Statecharts”.

Moreover, performance models are probabilistic by nature, and it is more than natural if one decides to add a probability whenever such conflict takes place. In this case the transition is fired based on the probability provided to each of the destination states. Therefore, the proposal of a solution to resolving this case is presented here [Vijaykumar, 1999] and [Vijaykumar et al., 2006]. The specified transition rate is multiplied by the probability specified in each destination arc. The interpretation of associating a probability to an event is similar to associating a condition to an event. In the case of conditioned event, the event is “protected” by the condition and the transition is only fired if the condition is satisfied. Whereas, in the case of probability, it is rather
associated to an event, i.e., the transition is fired “probabilistically” based on the specified probability. Formally, the probabilistic transition is defined as a decomposition of a Statechart transition: given a discrete set of probabilities \( P = \{p_1, p_2, \ldots, p_n\} \), such that (a) \( 0 \leq p_i \leq 1, \forall i = 1, 2, \ldots, n \); and (b) \( \sum_{i=1}^{n} p_i = 1 \).

A Statechart transition labeled by \( \text{event}[\text{condition}]/\text{action} \) can be decomposed into a set of transitions \( T = \{t_1, t_2, \ldots, t_n\} \), where each \( t_i \) \((i=1,2,\ldots,n)\), labeled as \( \text{event}[\text{condition}]/\text{probability}/\text{action} \). Therefore, a transition is such that when the \text{event} is triggered and \text{condition} satisfied, the reactive system changes from a source state \( S \) of the original transition to a target state \( S_i \) with probability \( p_i \).

Consider Figure 3a representing a machine that, when idle (W) through event \( a \) may provide some type of processing (P). Once the processing is finalized, the event \( s \) is triggered and the machine returns to W. Now, consider that the machine may be required for three types of services T1, T2 or T3 with probabilities p1, p2 and p3 \((p1+p2+p3 = 1)\) respectively. Here, P is decomposed into three sub states. Figure 3b shows the transition labeled by \( a \) decomposed into three transitions representing a probabilistic service assignment. As a consequence, the transition labeled by \( s \) must also be decomposed, in this case, in a deterministic manner.

![](image1.png)

(a)

![](image2.png)

(b)

**Figure 3. Example of Entry by Probability**

However, it is possible to simplify this notation just as in the case of Entry by Condition (already proposed in Statecharts) through a similar connector Entry by Probability shown in Figure 4. This alternative provides a cleaner specification of the model. Now, with the addition of the probability feature the new notation is \( \text{event}[\text{condition}]/\text{probability}/\text{action} \) [Vijaykumar, 1999] and [Vijaykumar et al., 2006].

![](image3.png)

**Figure 4. Example of Entry by Probability using a connector**
While producing the graph, all the target configurations are considered with the corresponding transition rate multiplied to the probability assigned to that particular transition. A slight change is made to `reactToInternalEvent`, `reactToStochasticEvent` and `generateGraph`. The algorithm that reacts to a stochastic event has to return a list of target configurations as well as a list of transition rates in which each value corresponds to each element of the list of target configurations. The transition rates used in the transition matrix are those multiplied with probability of moving to the target configuration. In Figure 4 transition rates for moving from W to T1, T2 and to T3 are respectively a*p1, a*p2 and a*p3. The following blocks show the modifications for both the algorithms.

```c
generateGraph ( Statecharts stch )
{
  // Similar to the original generateGraph with changes to
  // consider probabilistic transitions to avoid non-determinism
  get initial configuration conf_init for stch
  // conf_list is a list of configurations
  // This list is required as probabilistic transitions lead to
  // several destination states
  conf_list = reactToInternalEvents ( conf_init )
  // conf_set holds configurations that have to be expanded
  add conf_list to conf_set
  WHILE (configuration conf belongs to conf_set AND there
    are events for conf that have to be stimulated) DO {
    make a list of external events for conf
    FOR (each external event ev from the list) DO {
      // conf_aux is necessary to “remember” the original source
      // configuration to continue stimulating the rest of the
      // events of the list
      conf_aux = conf
      // markov_chain holds the state-transition diagram consisting of source configuration,
      // target configuration and the stochastic event along with the transition rate
      // conf_aux reacting to ev
      conf_list_aux = reactToStochasticEvent ( conf_aux, ev, markov_chain )
      empty conf_list
      FOR ( each conf_laux from conf_list_aux ) DO {
        WHILE (there are internal events) DO {
          conf_list = reactToInternalEvents ( conf_laux, ev, markov_chain )
        }
      }
    }
    remove conf from conf_set
  }
}

reactToInternalEvents ( Configuration conf, Event ev, Markov_Chain mcaux )
{
  conf_list is a list of configurations
  transition_rate_list is the list of transition rates times probability on the arc
  FOR (all transitions) DO {
    IF (source state in transition == state in conf) THEN {
      IF (event in transition is INTERNAL and ACTIVE) THEN {
        add {sourcestate, ev*probability on the transition arc, destination_state} to mcaux
        add destination_state to conf_list
      }
    }
  }
}
reactToStochasticEvent ( Configuration conf, Event ev, Markov_Chain mcaux )
{
    conf_list is a list of configurations
    transition_rate_list is the list of transition rates times probability on the arc
    FOR ( all transitions ) DO {
        IF (source state in transition == state in conf) THEN {
            IF (ev == stochastic event in the transition) THEN {
                add {source state, ev*probability on the transition arc, destination state} to mcaux
                add destination state to conf_list
            }
        }
    }
    return conf_list
}

Another interesting feature already defined in the original Statecharts language by Harel is the capability of “remembering” the last visited state. In order to illustrate this feature, consider an equipment A that when idle can be requested to process two types of jobs T1 or T2. Event a1 triggers the job of type T1 whereas a2 triggers the T2. Events s1 and s2 represent end of service for jobs T1 and T2 respectively. Both the types of jobs undergo through two sub-processes T11 and T12 (in case of T1) and T21 and T22 (in case of T2). While the equipment is busy, it may fail and moves to F (Failure) through event f. Note that the condition not in(W) is attached to the event to guarantee that the equipment may fail only when not idle.

When the failure is corrected the last state visited becomes active (represented by symbol H). In Figure 5(a) when A is entered, most recently visited state (W, T1 or T2) will become active bypassing the provided default state. In case T1 is the most recently visited state, the sub-state T11 will become active as it is the default state within T1 while in case of T2, sub-state T21 (default state within T2) will become active. In Figure 5(b) as H* is used the state to be active within T2, for example, may be T21 or T22 (whichever was the most recently visited state). Other approach to deal with the feature entry by history to influence only certain levels of the hierarchy can be done by using the appropriate number of H symbols applied to the desired level.

![Figure 5. Entry by History](image_url)

However, due to the memoryless property in the Markov chains, - future behavior of the process does not depend on the past behavior given the present state of
the chain - Entry by History seems, at a first glance, not to be compatible with Markov models. But, in fact, memory is frequently introduced in Markov models just by adding the necessary past information into the state space as an additional component. By introducing this feature, representation of performance models in Statecharts can be very much enhanced. Therefore, a solution to automatically cope up with this feature is provided [Vijaykumar, 1999] and [Vijaykumar et al, 2002b]. The main idea consists of adding the history information in each possible “present” state configuration by creating a new orthogonal component to the root state. In order to “synchronize” the return to last visited state (affected by the history symbol H or H*), internal events \textit{entered (state)} and \textit{exit (state)} are used. These events become (“immediately”) true if the parameter \textit{state} becomes active and when it is not any more active (an event causes transition from it) respectively.

In order to illustrate this idea, observe Figure 6. It illustrates an equipment that processes three types of services T1, T2 or T3. Event a1 takes the equipment to process service T1, event a2 service of type T2 and event a3 service of type T3. Equipment may fail but it is assumed that the failure occurs only while processing. Event f takes the equipment to a failure state F. After the repair it must return to the last active service (T1 or T2 or T3) via history (symbol H). In order to cope up with this situation, the solution has to “memorize” the last active state among T1, T2 or T3 before a failure occurs so that this active state is reached after returning from F. Therefore, a dummy component, named for example, History(P) is created and it consists of the states that are essential to history plus one more state Active. This extra state is used whenever the P component is active. Whenever the event f takes the system from P to F, the state in which the P component was before occurring a failure is “memorized” by making the corresponding state in the dummy component active.

\begin{center}
\includegraphics[width=0.5\textwidth]{figure6.png}
\end{center}

\textbf{Figure 6. Entry by History feature}

One can notice that in the dummy component, events already defined in the Statecharts formalism such as \textit{en(X)} and \textit{ex(X)} are used. This situation is shown in Figure 7. In case P is not any more active while in T1, immediately the dummy component History(P) would make the HT1 sub-state active as the event \textit{ex(P) \land ex(T1)} becomes true. As one can notice regardless of the destination state from P (either F or W), T1 is “memorized”. However, the “memory” aspect will be used only when returning from F.
Note that in the proposed solution, a dummy state History(E) must be created for each state E that contains Entry by History. In the general case, these dummy states must be the direct offspring of the root state i.e. they must be on the same level of the other orthogonal components of the root state. History is eliminated using this solution, i.e. past information becomes part of the present configuration. Just for illustration purposes of using entry by history all the way down within an hierarchy, example of Figure 5b is considered. Now, Figure 8 shows the addition of the orthogonal component to deal with Figure 5b where H* is used. In this case it is necessary to “remember” all the sub-states within T1 and T2. Therefore, it is essential to memorize T11, T12, T21 and T22 so that it is possible to get back to the state that was active. It is essential to make it clear here that both H and H* symbols are applied only to states of XOR type.

The following blocks describe algorithms to deal with H and H*.

1. H symbol is detected in State S
2. create an orthogonal state (as a son of Root) and name it as History(S)
3. // creating sub-states within History(S)
4. create one sub-state with name Active
5. create as many number of sub-states as in S maintaining the same original name but with prefix H_
6. make the first one of these to be chosen as an initial default state
7. // creating transition arcs with the immediate events within History(S) – sub-state Active is target state
8. for each sub-state H_ss within History(S) except Active
9. create a transition arc from H_ss to Active
10. event on this transition arc must be entered (S)
5. PerformCharts and its Interface

As mentioned elsewhere, the idea of using Statecharts to specify a reactive system and generate its performance information by associating the specification with a continuous-time Markov chain has turned into a software product, PerformCharts. This section reports the elements that were taken into account for the implementation and briefly explains the interface developed based on XML.

Harel proposed Statecharts with some basic elements to specify a complex reactive system. These elements are states, events, actions, conditions, variables, expressions and transitions. Initially, PerformCharts did not take into consideration
variables and expressions as reactive systems that could make use of such elements for performance evaluation did not come into picture. However, they have been recently added to the tool in order to deal with generating test sequences [Amaral, In Press] of a reactive system and this will be commented in Section 6.

Events that are considered in PerformCharts are: true condition event (true when condition is satisfied), primitive event (stochastic with a transition rate and internal), entered state (enabled when a given state becomes active), exit state (enabled when a given state is no more active) and conditioned event. Conditions considered for PerformCharts are: in state condition (becomes true if a given state is active), composed condition and condition related to comparing variables. Actions that were implemented so far are event triggering action which is an event and this event is considered to be internal that responds automatically to the environment without having to explicitly stimulate it. Two more actions have been added recently in order to deal with test sequence generation: action as output – prints an output message when the event is triggered and the transition fired; and action dealing with expressions.

The interface to deal with the specification as well as the necessary procedures to generate the Markov chain and the steady-state probabilities is by means of writing a main program in C++ language. An ideal interface is a graphical one and it is under development. However, by using XML interoperability language, PcML (PerformCharts Markup Language) was proposed and it is being used as an interface for the time being. Basically, the specification of a reactive system in Statecharts is written in PcML and scripts in perl or java can be used to convert this language into a main program in C++ language. The functionality of PerformCharts can be described by Figure 9. The specification in Statecharts is converted into a Markov chain. This Markov chain is then resolved by the numerical methods implemented in Modesto to generate the steady-state probabilities.

![Figure 9. PerformCharts](image)

6. Example

Consider a flexible manufacturing system with two machines Mₐ and Mₘ operating in series and continuously producing a single product, an in-process inventory S, a robot Rb and an operator Op [Vijaykumar et al, 2002a]. Inventory is used to store the products
finished by the first machine so that the waiting time of the machines is reduced. Maximum size of the inventory is \( N \geq 0 \). Each product goes through a process by \( M_a \) and is followed by another process by \( M_b \). Robot \( R_b \) loads and unloads products on the machines. Machines and the robot are subject to failure and the operator \( O_p \) corrects them. The involved times, i.e., time to failure of the machines or the robot, the corresponding times to repair the failures, times to process each product and times to load and unload the products in the inventory are considered to be exponentially distributed.

A Statecharts representation of the system is shown in Figure 10. Machines \( M_a \) and \( M_b \) are modeled with sub-states \( W \) (“waiting to be loaded”), \( P \) (“processing a product”), \( WU \) (“waiting to be unloaded”) and \( B \) (“failure”), \( W \) being the initial state. An event \( c_1 \) (\( c_2 \)) generated by the robot corresponds to loading a product on \( M_a \) (\( M_b \)) and fires a transition from \( W \) to \( P \) state. It is important to note that the two machines operate at the same time but on different products.

Machines exit from \( P \) state either when a product has been processed or a failure has occurred. Machines are subject to failure only while in operation. Event \( \beta_1 \) in \( M_a \) (and \( \beta_2 \) in \( M_b \)) corresponds to a termination of service (production). Event \( \lambda_1 \) in \( M_a \) (and \( \lambda_2 \) in \( M_b \)) corresponds to a failure.

After processing a product, machines move to \( WU \) state. If a failure occurs, machines switch to \( B \) and in this case, a product in process can be lost with probability \( p_1 \) for \( M_a \) (\( p_2 \) for \( M_b \)). While in \( WU \), machines have to wait for the robot to unload the product. Once the robot unloads the product, it triggers an event \( d_1 \) or \( d_2 \) (\( M_a \) or \( M_b \)) and a transition from \( WU \) to \( W \) is fired. In failure state \( B \), the operator repairs the machines. Termination of a repair corresponds to an event \( r_1 \) or \( r_2 \) (\( M_a \) or \( M_b \)) triggered by the operator; then, switching to state \( W \) if the product being processed is lost or switching to state \( P \) if the product being processed is not lost (the product continues to be processed).

Each state of the inventory \( S \) represents the number of products it contains. Events \( s_i \) and \( s_r \) represent a product to be placed in or removed from the inventory, respectively.

Robot \( R_b \) consists of three sub-states \( W \) (“waiting”), \( P \) (“processing”) and \( B \) (“failure”). The robot automatically operates loading (or unloading) a product on the machines or in the inventory. Priorities established for the robot are: (1) unload a product from \( M_b \), (2) unload a product from \( M_a \) and immediately load it on \( M_b \), (3) load a product on \( M_b \) from the inventory, (4) unload a product from \( M_a \) and load it in the inventory and (5) load a product to be processed on \( M_a \). Events \( \gamma_1 \) and \( \gamma_2 \) correspond to loading \( M_a \) and \( M_b \) respectively whereas events \( \delta_1 \) and \( \delta_2 \) correspond to unloading \( M_a \) and \( M_b \) respectively. Event \( \alpha \) makes the robot to unload a product from \( M_a \) and loads it directly into \( M_b \).

The initial state is \( W \). State \( B \) becomes active when the robot fails. After the correction the robot must finish the interrupted task. Therefore, the return to state \( P \) is made by an Entry by History. State \( P \) is divided into 5 sub-states representing the different tasks that can be executed and are presented by the order of priority:

- U2 (unloading \( M_b \)) is active whenever \( M_b \) is in \( WU \) state, and returns to \( W \) after generating event \( \delta_2 \) and triggering event \( d_2 \);
• UL (unloading $M_a$ and directly loading $M_b$) is active whenever $M_a$ is in WU state and $M_b$ is in W, and returns to W after generating event $\alpha$ and triggering events $d_1$ and $c_2$;

• L2 (loading $M_b$ from inventory S) is active whenever inventory S is not empty and $M_b$ is in W state, and returns to W after generating event $\gamma_2$ and triggering events $c_2$ and $s_i$;

• U1 (unloading $M_a$ to the inventory S) is active whenever inventory is not full and $M_a$ is in WU state, and returns to W after generating event $\delta_1$ and triggering events $d_1$ and $s_i$;

• L1 (loading $M_a$) is active when $M_a$ is in W, and returns to W after generating event $\gamma_1$ and triggering event $c_1$.

---

**Figure 10. Statecharts representation of a Flexible Manufacturing System**

Notice that the robot task priority was given to minimize the number of products in the system. So machine $M_b$ has a larger priority than $M_a$; more priority is provided to unloading a finished product from $M_b$ and a lower priority is assigned to load a product produced in $M_a$.

As shown in Figure 10, once the robot is repaired it returns from state B to the last active state (among U2, UL, L2, U1, L1) within the P macro-state by Entry by History (H).

Operator Op consists of W (“waiting”), R1 (“repairing $M_a$”), R2 (“repairing $M_b$”) and Rr (“repairing the robot”), initial state being W. The highest priority is given to repair $M_b$ followed by a lower priority to repair $M_a$ and the lowest priority is to repair
the robot. Once in state W, the operators reacts immediately to any failure in the system and switches to the corresponding repairing state. Events $\mu_1$, $\mu_2$, and $\mu$ correspond respectively to repair $M_a$, $M_b$ and the robot. After repairing, the operator generates an event ($r_1$ for the machine $M_a$, $r_2$ for the machine $M_b$ and $r_r$ for the robot) and returns to the initial state W.

Input parameters for the model are shown in Table 1. The model was tested based on considerations of the following cases: slower rhythm of $M_a$ than $M_b$ ($\beta_1 = 8$); $M_a$ with same production capacity rhythm as $M_b$ ($\beta_1 = 10$); faster rhythm of $M_a$ than $M_b$ ($\beta_1 = 12$). In each case, the inventory size varies between 0 and 15.

<table>
<thead>
<tr>
<th>Table 1. Input parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production rate</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Rate of loading $M_a$</td>
</tr>
<tr>
<td>Rate of unloading $M_a$</td>
</tr>
<tr>
<td>Rate of loading $M_b$</td>
</tr>
<tr>
<td>Rate of unloading $M_b$</td>
</tr>
<tr>
<td>Rate of moving from $M_a$ to $M_b$</td>
</tr>
</tbody>
</table>

Figures 11(a) and 11(b) show respectively variation of average productivity of the system (products/unit of time) and the average size of the inventory. Figures are plotted in terms of the inventory maximum size. The average productivity of the system corresponds to the average rhythm of the production by $M_b$. Figure 11(a) shows that the average productivity increases with the average size of the inventory.

Productivity is always limited by the production rhythm and reliability of the machines. Figure 11(b) shows that the system tends to use more the capacity of the inventory with a greater production rhythm of $M_b$. If $M_a$ is faster than $M_b$ ($\beta_1 = 12$), inventory size might overflow without any effective gain in the productivity. In order to obtain an optimal maximum size or a policy to control the inventory, it would be necessary to add a cost structure such as stocking costs, repairing costs and costs for machine idleness.
Figure 11. Average productivity of the system and average size of the inventory in terms of its maximum size

Performance measures can be obtained for each set of values provided to the model. In the implementation, average rate of production and the loss of products of each machine, the average size of the inventory and the availability of each component of the system are considered. For example, for $\beta_1 = 10$ and for the maximum size of the inventory equal to 7 (point P in Figures 11(a) and 11(b)), average size of the inventory is 2.82 and the rest of the performance measures are shown in Table 2.

Table 2. Performance measures

<table>
<thead>
<tr>
<th></th>
<th>Machine M_a</th>
<th>Machine M_b</th>
<th>Robot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average rate of production (products/time)</td>
<td>6.91</td>
<td>6.81</td>
<td></td>
</tr>
<tr>
<td>Average rate of losing the products</td>
<td>0.35</td>
<td>0.10</td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td>92.9%</td>
<td>97.5%</td>
<td>97.5%</td>
</tr>
</tbody>
</table>

With the average rate of production and loss of products of each machine, one can establish a flow chart as shown in Figure 12. The horizontal flows correspond to the input and output while the vertical flows correspond to the loss of products in the system.

Figure 12. Product flow in the system

Considering the present example, there are interesting applications that take advantage of including a policy to control the inventory. By including some costs such
as stocking costs, repairing costs, etc. and making use of Markov Decision Processes, one can come out with decision support systems. Statecharts will be also considered for representing Markov Decision Processes and this project has an important role to play in one of the research streams within stochastic processes.

7. Final Remarks

Due to their visual appeal and enabling the representation of many complex features of reactive systems, Statecharts have been used in modeling a wide variety of systems. Based on this aspect, as shown in the paper, they have also been considered to represent performance models. It was also shown that the represented model can be associated to a mathematical solution, in particular, with continuous-time Markov chain. From Markov chains, steady-state probabilities are obtained and these probabilities are the basis for determining the performance measures.

In this paper the main algorithm presented showed how a Statecharts representation is converted into a continuous-time Markov chain. In this case, stochastic events are explicitly stimulated and immediate (or internal) events are automatically triggered.

By nature performance models are probabilistic. Therefore an option is provided to include probabilistic transitions where non-determinism feature is not desirable. A probabilistic transition may occur (with same event) to more than one target state. This feature was not included in the original Statecharts formalism but was suggested by Harel that this could be a good option in cases where non-determinism was not desirable. One more feature in Statecharts is to “remember” last visited state. Although there is a conflict of this feature with Markov theory (next state depends only on the present), a solution to bring the past to the present has been adopted. However, it is important to bear in mind that one has to pay the price for representing as many “dummy” states as required. This software has been called as PerformCharts and written in C++. In the initial stage, the interface used to be written in the main program by calling the necessary methods to specify the model. Recently, a textual interface based on XML (eXtensible Markup Language) has been developed to deal with the specification. Scripts written in perl and java interpret the interface and generates the main program in C++ and when linked with other classes and run, steady-state probabilities are generated.

A simulation approach has also been independently developed based on Statecharts representation. At the moment, the simulation solution is being integrated into PerformCharts. Graphical interface is being developed. The final objective is to come up with a web-based PerformCharts.

Another research in progress deals with PerformCharts in generating test cases. Several methods have been developed to generate test cases once the software is specified as a state-transition diagram (in fact Markov chain is a state-transition diagram). A software specification will be represented in Statecharts and PerformCharts will generate not exactly a Markov chain but a state-transition diagram with minor changes. This output will be associated with another tool that based on the state-transition diagram can generate test cases. As an experiment, the association was made by integrating PerformCharts to yet another tool CONDADO [Martins et al, 2000]
developed at UNICAMP. CONDADO tool generates test sequences using switch cover method as long as the specification is given as a Finite State Machine.

Acknowledgement

The authors duly acknowledge CAPES (Coordenação de Aperfeiçoamento Pessoal) for the financial support to the project 0226050 within PROCAD (Programa de Cooperação Acadêmica) program.

References


