Model-Driven Performability Analysis of An Event Demultiplexing Pattern

Aniruddha S. Gokhale
Dept of EECS
Vanderbilt Univ.
Nashville, TN 37235
a.gokhale@vanderbilt.edu

Swapna S. Gokhale*
Dept. of CSE
Univ. of Connecticut
Storrs, CT 06269
ssg@engr.uconn.edy

Jeffrey G. Gray
Dept. of CIS
Univ. of Alabama
Birmingham, AL
gray@cis.uab.edu

Abstract

The growing reliance on services provided by software applications places a high premium on the reliable and efficient operation of these applications. A number of these applications follow the event-driven software architecture style since this style fosters evolvability by separating the event handling from event demultiplexing and dispatching functionality. The event demultiplexing capability, which appears repeatedly across a class of event-driven applications, can be codified into a reusable pattern, such as the Reactor pattern. In order to enable performability analysis of event-driven applications at design time, a model is needed that represents the event demultiplexing and handling functionality that lies at the heart of these applications. The model must also consider failures in order to provide realistic performance or performability estimates. In this paper, we present a performability model of the Reactor pattern based on the well-established Stochastic Reward Net (SRN) modeling paradigm. We illustrate the use of the model for the performability analysis of the services provided by the applications residing on mobile handheld devices. Furthermore, we describe how the performability analysis process could be scaled and automated by a model-driven framework. The model-driven framework will make the analysis approach easily accessible to software architects and experts who may be non experts in the performability analysis techniques.

1. Introduction

Society is increasingly dependent on capabilities provided by distributed software applications that enable critical services (e.g., electric power grid, air traffic control, mobile communications). Due to the growing reliance of our daily activities on these services, efficient and reliable operation of the applications that provide these services is crucial. A number of these applications are event-driven software systems [8]. The event-driven style constitutes a provider/listener model [3], where the application listens for service requests or "events" and provides the necessary services in response to these requests or events. Event-driven systems provide many advantages, the most prominent advantage being evolvability, which is enabled by the separation of event demultiplexing and dispatching from event handling. Although the handling of events is specific to the services provided by the application, the event demultiplexing and dispatching functionality is more or less uniform across all the applications that follow the event-driven style, and can be codified into a reusable building block or a pattern [2]. The *Reactor* pattern [14] embodies the demultiplexing and dispatching capabilities that could be reused to facilitate the development of event-driven applications.

It is a well-known fact that it is important to analyze the performance of an application starting from the early phases of the life cycle [17]. Moreover, pure performance analysis, which does not consider application failures leads to optimistic performance estimates. Thus, it is important to consider application failures during performance analysis in order to provide realistic "performability" estimates [15]. Design-time performance analysis is almost invariably based on a model that represents application behavior. As a result, in order to enable model-based performance analysis of event-driven applications, it is necessary to build a model of the underlying event demultiplexing framework that is ubiquitous in such applications.

In this paper we present a performability model of the reactor pattern, which provides the event demultiplexing and dispatching capabilities. The model is based on the Stochastic Reward Net (SRN) [12] modeling paradigm. In addition to the event demultiplexing capabilities, the model also considers application failures during event

* This research was supported in part by a grant from the National Science Foundation, NSF-CNS #0406376

handling. We consider the applications which provide productivity enhancing services on mobile handhelds as the guiding examples of event-driven systems. Such applications can leverage the reactor pattern for their design and implementation. We illustrate how the SRN model of the reactor pattern could be used for the performability analysis of the service provided by such applications. We also discuss how the performability analysis process can be scaled using a model-driven framework. The framework also enables the automation of the process so that it can be easily used by software architects and designers who may be non experts in the performability analysis tools and techniques.

This paper is organized as follows: Section 2 describes the applications that may reside on mobile handheld devices that illustrate the event-driven features; Section 3 discusses the process building a performability model of the reactor pattern using the SRN modeling paradigm; Section 4 illustrates how the SRN model could be used for the performability analysis of the services provided by applications on the mobile devices; Section 5 describes how the performability analysis process could be scaled and automated using a model-driven framework and finally Section 6 provides concluding remarks outlining the lessons learned from this study, as well as directions for future research.

2. Overview of Event Demultiplexing Patterns

This section describes the event demultiplexing patterns found in real-world systems that are used for ubiquitous computing, such as wireless handhelds that support number of applications like email, calendar, web browsing and multimedia. Section 2.1 provides at a typical event demultiplexing software architecture in handhelds. Section 2.2 then presents an outline of the reactor pattern [14] found in many real-world event driven systems including the handhelds.

2.1. Event-Driven Software Architecture in Mobile Handhelds

Figure 1 illustrates a typical software architecture for event demultiplexing and dispatching in mobile handhelds. Handhelds, such as PDAs, have been at the forefront of ubiquitous computing and are getting increasingly complex due to the need to support to multiple applications, such as email, web browsing, calendar management, multimedia support, and games. Since these handhelds are used in many critical domains such as patient healthcare monitoring and emergency response systems, reliable and efficient operation of these handhelds is esssential. Further, since the software residing in these handhelds is primarily responsible for providing the necessary functionality, it is necessary to analyze the software applications for their performance and reliability characteristics. In particular, in this paper we describe a methodology for the performability analysis of such event-driven software applications, which are commonly used in ubiquitous computing.

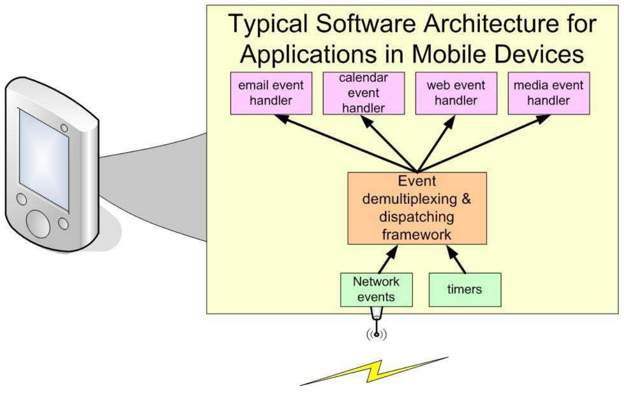
Typical Software Architecture for Applications in Mobile Devices
email event handler
calendar event handler
web event handler
media event handler
Event demultiplexing & dispatching framework
Network events
timers

Figure 1. Event Demultiplexing in Handhelds

The application mix available in the handhelds is aptly suited for event-driven systems, which requires demultiplexing and dispatching events to the correct event handlers in the handheld. For example, a user of the handheld could have set appointments in his/her calendar, which might raise an event while the user may be in the midst of web browsing. Similarly it is conceivable that email notification arrives while the user is in the midst of other applications like web browsing or listening to audio. In the context of handhelds that are tailored to provide service to emergency response personnel, one could conceive of a scenarios where sensor data from a phenomenon of interest is received by the handheld with other notifications, such as SMS notifications from the command and control, received by the handheld.

2.2. Description of the Reactor Pattern

Event demultiplexing, dispatching and event handling is a hallmark of event-driven systems exemplified by the application mix in handhelds. The development of these systems is increasingly relying on the use of well-documented patterns [2, 14], The use of patterns offers the potential of improving the reliability as well as enabling faster time-to-market by fostering reuse. Fundamental to the design and development of event-driven systems is the Reactor pattern [14]. Therefore, performability analysis of the event-driven aspects of systems, such as the handhelds, boils down to the performability analysis of the reactor pattern. This section therefore provides background information on the reactor pattern and its dynamics.

Figure 2 depicts a typical event demultiplexing and dispatching mechanism documented in the reactor pattern [14]. The application registers an event handler with the event demultiplexer and delegates to it the responsibility of listening to incoming events. On the occurrence of an event, the demultiplexer dispatches the event by making a callback to the correct application-supplied event handler. This is the idea behind the reactor pattern, which provides synchronous event demultiplexing and dispatching capabilities.

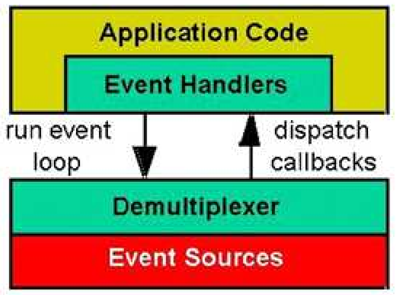

Figure 2. Event Demultiplexing Pattern

Figure 3 illustrates the reactor pattern dynamics. We categorize these dynamics into two phases described below.

1. **Registration phase:** In this phase all the event handlers register with the reactor associating themselves with a particular event type they are interested in. Event types usually supported by a reactor are input, output, timeout and exceptions. The reactor will maintain a set of handles corresponding to each handler registered with it.

2. **Snapshot phase:** Once the event handlers have completed their registration, the main thread of control is passed to the reactor, which in turn listens for events to occur. A snapshot is then an instance in time wherein a reactor determines all the event handles that are enabled at that instant. For all the event handles that are enabled in a given snapshot, the reactor proceeds to service each event by invoking the associated event handler. There could be different strategies to handle these events. For example, a reactor could handle all the enabled events sequentially in a single thread or could hand it over to worker threads in a thread pool. After all the events are processed, the reactor proceeds to take the next snapshot of the system.

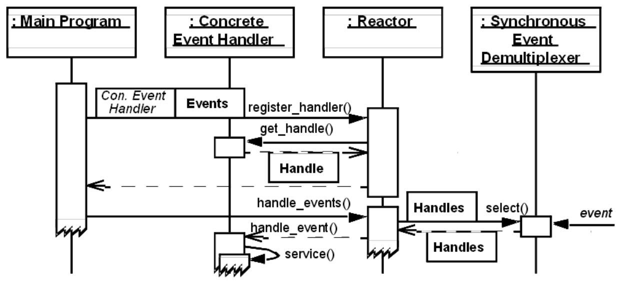

Figure 3. Reactor Pattern Dynamics

Many different variations of the reactor pattern are possible. These variations stem from the different event demultiplexing and event handling strategies used in a reactor. For example, in network-centric applications, networking events can be demultiplexed using operating system calls, such as `select` or `poll`. For graphical user interfaces (GUIs), these events could be due primarily to mouse clicks and can be handled by GUI frameworks like Qt or Tk. On the other hand, the event handling mechanisms could involve a single thread of control that demultiplexes and handles an event, or each event could be handled concurrently using worker threads in a thread pool or by thread on demand.

In this paper, we have developed a performability model of the reactor pattern, which uses the `select` system call for event demultiplexing and uses a single thread of control to demultiplex and handle events identified in a given snapshot. In the rest of the paper we describe this model and demonstrate the use of the model for performability analysis.

3. Performability Model of the Reactor Pattern

In this section we describe the process of constructing a SRN model for the reactor pattern. Towards this end, we first describe the characteristics of the reactor pattern and the relevant performance measures. A SRN model of the reactor pattern is then presented along with a discussion of how the performance measures can be obtained by assigning reward rates at the net level. We conclude the section with a discussion of possible extensions to the SRN model.

3.1. Characteristics of the Reactor Pattern

In our performability model of the reactor, we consider a single-threaded, select-based implementation of the reactor pattern with the following characteristics, as shown in Figure 4:

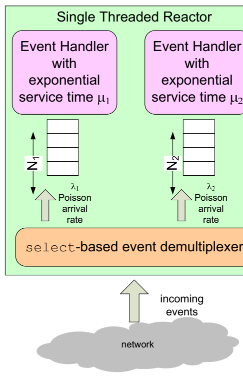
Single Threaded Reactor
Event Handler with exponential service time μ_1
Event Handler with exponential service time μ_2
N_1
N_2
λ_1 Poisson arrival rate
λ_2 Poisson arrival rate
select-based event demultiplexer
incoming events
network

Figure 4. **Characteristics of the Reactor**

- The reactor receives two types of input events with one event handler for each type of event registered with the reactor.
- Each event type has a separate queue, which holds the incoming events of that type. The buffer capacity for the queue of type #1 events is denoted N_1 and of type #2 events is denoted N_2.
- Event arrivals for both types of events follow a Poisson distribution with rates λ_1 and λ_2, while the service times of the events are exponentially distributed with rates μ_1 and μ_2.
- In a given snapshot, if the event handles corresponding to both the event types are enabled, then they are serviced in no particular order. In other words, the order in which the events are handled is non-deterministic.
- The failures in the reactor are assumed to occur only in the event handlers (e.g., due to memory corruption), in which case the event is simply purged from the system. The failure rates when handling events of type #1 and type #2 are γ_1 and γ_2, respectively. Other kinds of failures not considered here will cause the entire application to fail due to the single thread model.

The following performance metrics are of interest for each one of the event types in the reactor pattern described in Section 3.1:

- **Expected throughput** – which provides an estimate of the number of events that can be processed by the single threaded event demultiplexing framework. These estimates are important for many applications, such as telecommunications call processing.
- **Expected queue length** – which provides an estimate of the queuing for each of the event handler queues. These estimates are important to develop appropriate scheduling policies for applications with real-time requirements.
- **Expected total number of events** – which provides an estimate of the total number of events in a system. These estimates are also tied to scheduling decisions. In addition, these estimates will determine the right levels of resource provisioning required to sustain the system demands.
- **Probability of event loss** – which indicates how many events will have to be discarded due to lack of buffer space. These estimates are important particularly for safety-critical systems, which cannot afford to lose events. These also provide an estimate on the desired levels of resource provisioning.

3.2. SRN Model of the Reactor Pattern

SRNs represent a powerful modeling technique that is concise in its specification and whose form is closer to a designer's intuition about what a model should look like. Since a SRN specification is closer to a designer's intuition of system behavior, it is also easier to transfer the results obtained from solving the models and interpret them in terms of the entities that exist in the system being modeled. An overview of SRNs can be found in [12]. Stochastic reward nets have been extensively used for performance, reliability and performability analysis of a variety of systems [13, 4, 5, 16, 6, 10]. The work closest to our work is reported by Ramani *et al.* [13], where SRNs are used for the performance analysis of the CORBA event service. The CORBA event service is yet another pattern that provides publish/subscribe services.

Description of the net: Figure 5 shows the SRN model for the Reactor pattern described in Section 2.2. The net comprises of two parts. Part (a) models the arrival, queuing, service and failures of the two types of events. as explained below. Transitions $A1$ and $A2$ represent the arrivals of the events of types one and two, respectively. Places $B1$ and $B2$ represent the queue for the two types of events. Transitions $Sn1$ and $Sn2$ are immediate transitions which are enabled when a snapshot is taken. Places $S1$ and $S2$ represent the enabled handles of the two types of events, whereas transitions $Sr1$ and transition $Sr2$ represent the execution of the enabled event handlers of the two types of events. An inhibitor arc from place $B1$ to transition $A1$ with multiplicity $N1$ prevents the firing of transition $A1$ when there are $N1$ tokens in the place $B1$. The presence of $N1$ tokens in the place $B1$ indicates that the buffer space to hold the incoming input events of the first type is full, and no additional incoming events can be accepted. The inhibitor arc from place $B2$ to transition $A2$ achieves the same purpose for type two events.

Part (b) models the process of taking successive snapshots and non-deterministic service of event handles in each snapshot as explained below. Transition $Sn1$ is enabled when there are one or more tokens in place $B1$, a token in place $StSnpSht$, and no token in place $S1$. Similarly, transition $Sn2$ is enabled when there are one or more tokens in place $B2$, a token in place $StSnpSht$ and no token in place $S2$. Transition T_StSnp1 and T_StSnp2 are enabled when there is a token in either place $S1$ or place $S2$ or both. Transitions T_EnSnp1 and T_EnSnp2 are enabled when there are no tokens in both places $S1$ and $S2$. Transition $T\_ProcSnp2$ is enabled when there is no token in place $S1$, and a token in place $S2$. Similarly, transition $T\_ProcSnp2$ is enabled when there is no token in place $S2$ and a token in place $S1$. Transitions $Sr1$ and $TFail1$ are enabled when there is a token in place $SnpInProg1$, and transitions $Sr2$ and $TFail2$ are enabled when there is a token in place $SnpInProg2$. Table **??** summarizes the enabling functions for the transitions in the net shown in Figure 5.

Dynamic evolution of the net: We explain the process of taking a snapshot and servicing the enabled event handles in the snapshot with the help of an example. We consider a scenario where there is one token each in places $B1$ and $B2$. Due to the presence of tokens in places $B1$ and $B2$, and a token in place $StSnpSht$, transitions $Sn1$ and $Sn2$ are enabled. Both these transitions are assigned the same priority, and hence either one of them can fire. Without loss of generality, we assume that transition $Sn1$ fires first, which deposits a token in place $S1$. The presence of a token in place $S1$ and place $StSnpSht$ enables transition T_StSnp1. Also, transition $Sn2$ is already enabled. If transition T_StSnp1 were to fire before transition $Sn2$, the firing of transition $Sn2$ would be precluded. In order to prevent this from happening, transition $Sn2$ is assigned a higher priority than transition T_StSnp1, so that transition $Sn2$ fires before T_StSnp1 when both are enabled. Firing of transition $Sn2$ deposits a token in place $S2$ which enables transition T_StSnp2. Transitions T_StSnp1 and T_StSnp2 are both enabled, corresponding to the event handles of both types of events. If transition T_StSnp1 fires before T_StSnp2, then event handle for the first type of event will be executed prior to event handle for the second type of event. On the other hand, T_StSnp2 fires before T_StSnp1, then event handle for the second type of event will be executed prior to event handle for the first type of event. Both the transitions T_StSnp1 and T_StSnp2 have an equal chance of firing, and this represents the non-determinism in the execution of the enabled event handles. Without loss of generality, we assume transitions T_StSnp1 fires depositing a token in place $SnpInProg1$, which enables transitions $Sr1$ and $TFail1$. Additionally, firing of transition T_StSnp1 precludes the firing of transition T_StSnp2 and vice versa. Once transition $Sr1$ fires, a token is removed from place $S1$. If transition $TFail1$ fires prior to the firing of transition $Sr1$, it indicates that a failure occurred before the event handle could complete execution, and this also removes a token from place $S1$. Thus, either transition $Sr1$ and $TFail1$ fire and remove a token from place $S1$, after which transition $T\_ProcSnp2$ fires and deposits a token in place $SnpInProg2$. A token in place $SnpInProg2$ enables transition $Sr2$ and $TFail2$. $TFail2$ plays the same role in the service of the event type #2, as played by $TFail1$ in the service of the event type #1. Firing of $Sr2$ or $TFail2$ removes the token from place $S2$. Once $Sr2$ fires, there are no tokens in places $SnpInProg1$ and $SnpInProg2$, which enables transition T_EnSnp2. The firing of T_EnSnp2 marks the completion of the present snapshot, and the beginning of the next one.

Assignment of reward rates: The performance measures described in Section **??** can be computed by assigning reward rates at the net level as summarized in Table 2. The throughputs T_1 and T_2 are respectively given by the rate at which transitions $Sr1$ and $Sr2$ fire. The queue lengths Q_1 and Q_2 are given by the average number of tokens in places $B1$ and $B2$, respectively. The total number of events E_1 is given by the sum of the number of tokens in places $B1$ and $S1$. Similarly, the total number of events E_2 is given by the sum of the number of tokens in places $B2$ and $S2$. The loss probability L_1 is given by the probability of $N1$ tokens in place $B1$. Similarly, the loss probability L_2 is given by the probability of $N2$ tokens in place $B2$.

4. Performability Analysis of Mobile Services

In this section we illustrate how the SRN model of the reactor pattern is used for the performability analysis of the application mix consisting of two services, namely, calendar service and email service. The user of the handheld device expects that he be notified of the incoming

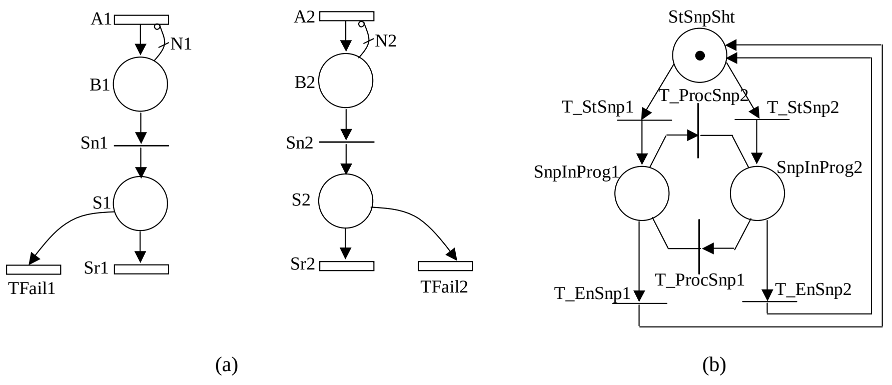

Figure 5. SRN model of the Reactor pattern

Transition	Guard function
$Sn1$	$((\#StSnpShot == 1)\&\&(\#B1 >= 1)\&\&(\#S1 == 0))?1:0$
$Sn2$	$((\#StSnpShot == 1)\&\&(\#B2 >= 1)\&\&(\#S2 == 0))?1:0$
T_StSnp1	$((\#S1 == 1)\|\|(\#S2 == 1))?1:0$
T_StSnp2	$((\#S1 == 1)\|\|(\#S2 == 1))?1:0$
$T\_ESnpSht1$	$((\#S1 == 0\&\&(\#S2 == 0))?1:0$
$T\_ESnpSht2$	$((\#S1 == 0\&\&(\#S2 == 0))?1:0$
$T\_ProcSnp1$	$((\#S1 == 1\&\&(\#S2 == 0))?1:0$
$T\_ProcSnp2$	$((\#S1 == 0\&\&(\#S2 == 1))?1:0$
$Sr1$	$(\#SnpInProg1 == 1)?1:0$
$Sr2$	$(\#SnpInProg2 == 1)?1:0$
$TFail1$	$(\#SnpInProg1 == 1)?1:0$
$TFail2$	$(\#SnpInProg2 == 1)?1:0$

Table 1. Guard functions

Performance metric	Reward rate
T_1	return rate($Sr1$)
T_2	return rate($Sr2$)
Q_1	return ($\#B1$)
Q_2	return ($\#B2$)
L_1	return ($\#B1 == N1?1:0$)
L_2	return ($\#B2 == N2?1:0$)
E_1	return($\#B1 + \#S1$)
E_2	return($\#B2 + \#S2$)

Table 2. Reward assignments to obtain performance measures

emails and the calendar events in a reasonable amount of time. Further, the probability of denying the service requests should be minimal. From the point of view of the service provider, the rate at which the service requests are processed or the throughput is of interest, since this will determine the revenue generated for the provider.

In order to implement the mix of these two services, the reactor pattern with the characteristics described in Section 3.1 can be used to demultiplex the events. The SRN model of the reactor pattern can thus be used for the performability analysis of the services. In order to use the SRN model, we designate the email notification requests as events of type #1, and calendar update events as events of type #2.

In the early stages of application development lifecycle, it is necessary to analyze the impact of design choices and configurations on the performance measures. Also, in these stages, it is rarely the case that the values of the input parameters can be estimated with certainty, which makes it imperative to analyze the sensitivity of the performance measures to the variations in the input parameters. Sensitivity analysis will enable the provider to determine the regions of operation during which service performance is acceptable. In the subsequent subsections we demonstrate how the SRN model can be used to explore alternative design configurations, as well as to facilitate sensitivity analysis.

4.1. Impact of Buffer Space

The buffer space available to hold the incoming events of each type is a configuration parameter of the reactor pattern and the services implemented by the pattern. This choice will have a direct impact on all the performability metrics. Most importantly, from the user's perspective, the buffer space will influence the probability of denying the service requests.

We analyze the impact of the buffer capacities on the performability measures. The values of the remaining parameters (except for the buffer capacities) are reported in Table 3. We consider two values of buffer capacities N_1 and N_2. In the first case, the buffer capacity is set to 1 for both types of events, whereas in the second case the buffer capacity of both types of events is set to 5. The performability metrics for both these cases are summarized in Table 4. Because the values of the parameters of the service requests from organization #1 (λ_1, μ_1 and N_1) are the same as the values of the parameters for the service requests from organization #2 (λ_2, μ_2, and N_2), the throughputs, queue lengths, and the loss probabilities are the same for both types of service requests for each one of the buffer capacities as indicated in Table 4. It can be observed that the loss probabilities are significantly higher when the buffer capacity is 1 compared to the case when the buffer space is 5. Also, due to the higher loss probability, the throughput is slightly lower when the buffer capacity is 1 than when the buffer capacity is 5.

For both buffer capacities, the total number of service requests from organization #2, denoted E_2, is slightly higher than the total number of requests from organization #1, denoted E_1. Because the requests from organization #1 are provided prioritized service over the requests from organization #2, on an average it takes longer to service a request from organization #2 than it takes to service a request from organization #1. This results in a higher total number of requests from organization #2 than the total number of requests from organization #1. This may result in a higher response time for service requests from organization #2 as compared with the service requests from organization #1. The reward rates to obtain the response times can be obtained using the tagged customer approach [9] and is a topic of our future research.

Parameter	Event #1	Event #2
Arrival rate	λ_1 = 0.400/sec.	λ_2 = 0.400/sec.
Service rate	μ_1 = 2.000/sec.	μ_2 = 2.000/sec.
Failure rate	γ_1 = 0.02/sec.	γ_2 = 0.02/sec.

Table 3. Parameter values

Measure	Buffer space	
	$N_1 = N_2 = 1$	$N_1 = N_2 = 5$
T_1	0.37/sec.	0.39/sec.
T_2	0.37/sec.	0.39/sec.
Q_1	0.064	0.12
Q_2	0.064	0.12
E_1	0.26	0.33
E_2	0.26	0.33
L_1	0.064	0.00024
L_2	0.064	0.00024

Table 4. Impact of buffer capacity on performability measures

4.2. Impact of Service Rates

Another design parameter that will impact the service performance is the service time of the event handlers. Obviously, a faster rate of service will improve performance, but from the hardware perspective, increasing the service rate may require higher levels of resources thereby increasing the cost of the device. Therefore, it is necessary to determine what service rate would be "good enough" to provide acceptable service performance. This can be accomplished by analyzing the sensitivity of the performability metrics with respect to the service rates, namely, μ_1 and μ_2.

The results reported in Section 4.1 indicate that the loss probability is significantly lower when the buffer space is 5 as compared to the loss probability when the buffer space is 1. As a result, for the purpose of sensitivity analysis with respect to μ_1 and μ_2, we set the buffer capacity for both service requests to be 5. We vary the service rates μ_1 and μ_2 in the range of 0.4/sec. to 2.0/sec. roughly in steps of 0.025, one at a time, to obtain the expected performance measures by solving the SRN model shown in Figure **??**. The remaining parameters are held at the values reported in Table 3.

The plots in Figure 6 show the variation of the performability measures with respect to μ_1. The left plot in the top column shows the variation of throughputs of service requests from organizations #1 and #2 with respect to μ_1. The figure indicates that as μ_1 decreases,

It can be noted that as μ_1 decreases, the performability measures degrade (loss probability, queue length

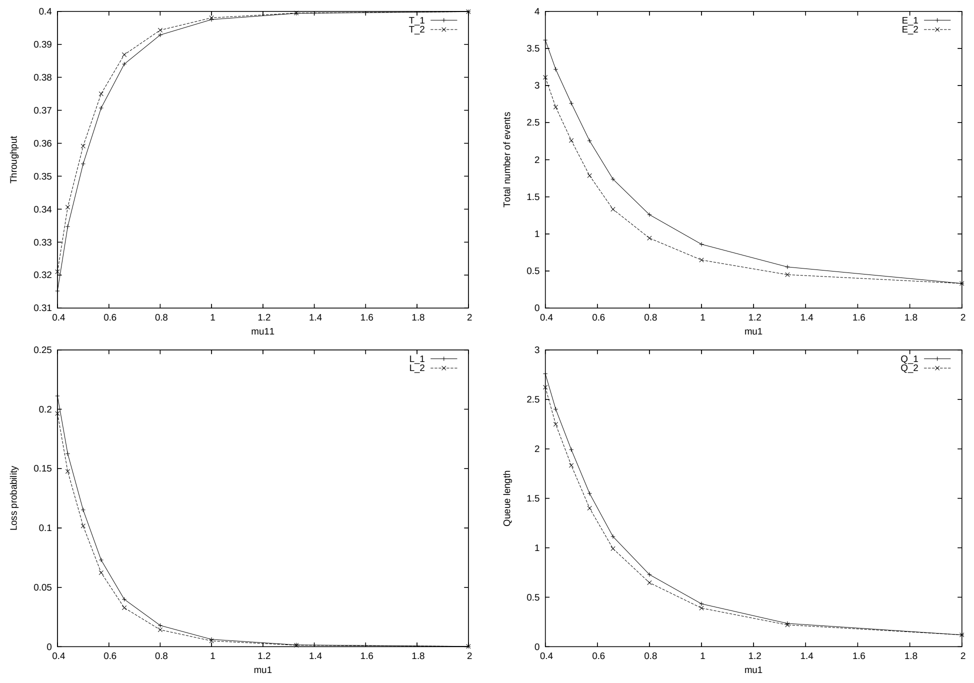

Figure 6. Sensitivity of performability measures to service rate μ_1

and total number of events increases, and throughput decreases). Also, as μ_1 decreases below 0.8/sec., the performability measures deteriorate rapidly. With decreasing μ_1, the time taken to service a request from organization #1 increases due to the additional time taken to complete each snapshot. As a result, indirectly, this also increases the time taken to service a request from organization #2 in each snapshot. Thus, the performance of requests from both organizations are adversely impacted, even though only the service time for the requests from only one of the organizations increases. The variation of the performability measures with respect to μ_2 shows the same trend, with the role of event types reversed. These results are not shown here due to space limitations.

4.3. Impact of Request Arrival Rates

The arrival rates of service requests, namely, λ_1 and λ_2, constitute the input parameters. In this section we analyze the the sensitivity of the performability measures to these input parameters.

Based on the results and analysis described in Section 4.1 and Section 4.2, we set N_1 and N_2 to 5, and we set μ_1 and μ_2 to 2.00/sec for sensitivity analysis with respect to λ_1 and λ_2. The parameters γ_1 and γ_2 were held at the values reported in Table 3. The parameters λ_1 and λ_2 were varied one at a time in the range of 0.4/sec to 2.0/sec. roughly in steps of 0.025 to obtain the expected performability measures by solving the SRN model shown in Figure **??**.

Figure 7 shows the performability measures as a function of λ_1. Referring to the left figure in the top row, it can be observed that initially, the throughput of service requests from organization #1 is nearly the same as the arrival rate of the requests, indicating that the requests are serviced at the same rate at which they arrive. However, as λ_1 increases, the throughput starts lagging the arrival rate, which indicates that the service rate is not sufficiently high to process the requests at the rate at which they arrive. This may cause the queue for requests from organization #1 to operate at full capacity for an extended period of time, which results in a rejection of the incoming requests from organization #1.

It can be observed that a decrease in the rate at which the throughput increases is marked by an increase in the loss probability of the requests (as shown in the third plot in the left column) and an increase in the queue length (as shown in the fourth plot in the left column) in Figure 7. The left plot in the second row represents the total number of service requests from each organization as a function of λ_1. The plot indicates that the total number of requests from organization #2 in the system increases as λ_1 increases. When λ_1 increases, the probability of having to service a request from organization #1 prior to servicing a request from organization #2 event in a given snapshot increases. Because requests from organization #1 have priority over requests from organization #2, requests from organization #2 tend to reside longer in the system as λ_1 increases. Thus, although the throughput of requests from organization #2 is unchanged with re-

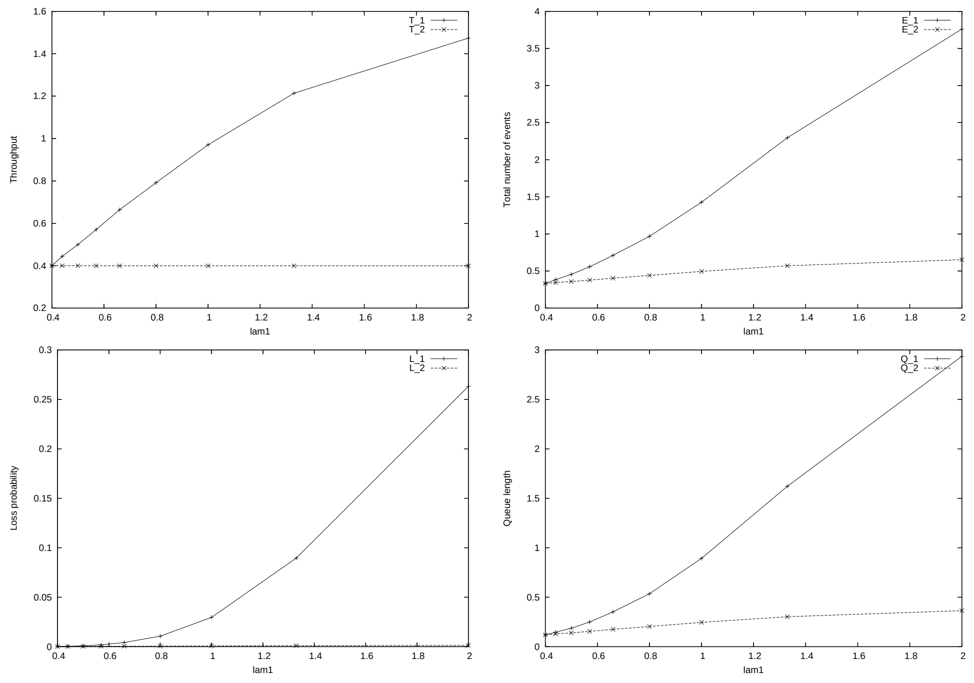

Figure 7. Sensitivity of performability measures to arrival rate λ_1

spect to λ_1, the response time of the requests from organization #2 may increase. The variation of the performability measures with respect to λ_2 shows the same trend, with the role of event types reversed. These results are not shown here due to space limitations.

5. Scaling and Automating the Performability Modeling Process

The previous sections described how a SRN model for performability of the desired application can be developed. The process described until now focusses on manually developing these models and the associated input scripts used by the model solvers, such as SPNP. As the application complexity grows, however, it becomes tedious and error prone to develop these models manually since the accidental complexities involved in modeling the application performability characteristics and the input script sizes for the model solver grow substantially. For example, the model described in Section 3 describes a performability model of a reactor, which handles only two types of events. In reality there could be several different event types that an event demultiplexing and handling framework may have to handle. Also, it should be possible for a software architect, who is a non expert in the tools and techniques of performability analysis to be able to use the model. To enable these dual objectives, it is necessary to encapsulate the performability modeling approach described in the earlier sections into user-friendly tools. In this section, we describe the model-driven approach which allows the user to scale the smaller base model and automate the process of performability analysis.

5.1. Modeling Languages for Performability Analysis

In this section we describe the ideas based on a model-driven [11] generative programming [1] framework we are developing to address the aforementioned challenges. Our modeling framework comprises a modeling language we have developed using the GME [7] metamodeling environment. The modeling language we have developed provides the visual syntactic and semantic elements that represent the artifacts of SRNs. Figure 8 shows a metamodel the represents our modeling language. This metamodel depicts a class diagram using UML-like constructs showcasing the elements of the modeling langauge and how they are associated with each other. Figure 9 illustrates part of the SRN model we have used in this paper, which has been modeled in our framework. The GME environment in which we have developed our modeling language also allows plugging in model interpreters that use generative programming to synthesize the input scripts for the SPNP model solver.

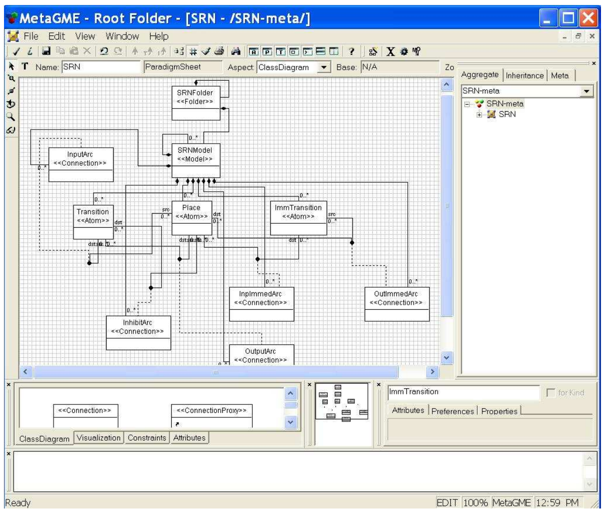

Figure 8. SRN Metamodel

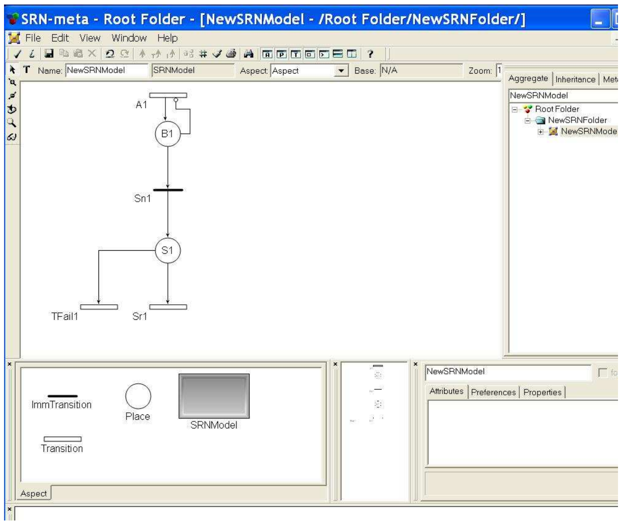
SRN-meta - Root Folder - [NewSRNModel - /Root Folder/NewSRNFolder/]
File Edit View Window Help
A1
B1
Sn1
S1
TFail1
Sr1
Root Folder
NewSRNFolder
NewSRNMode
ImmTransition
Place
SRNModel
Transition
Aspect

Figure 9. SRN GME Model of the reactor pattern

5.2. Scaling the Models via Model Replicators

5.3. Unresolved Challenges for Model-driven Frameworks

In the above sections, we describe how the base model which captures the essential concepts could be easily scaled using a model-driven framework. The scaled model essentially embodies the same characteristics of the original base model. In practice, however, it may be necessary to vary certain characteristics of the base model, in order to create "variants" of the pattern. For example, some applications that must provided differentiated levels of service must handle events in a prioritized manner, and the implementation of such applications will require a reactor pattern which provides prioritized handling. Another possibility is to handle events concurrently in multiple threads. In the future, we propose to develop the model-driven framework further, so that creating the variants of the base model will be facilitated, in addition to scaling the models with minimal manual intervention.

6. Conclusion and Future Research

In this paper we presented a performability model of the reactor pattern which codifies the event demultiplexing and dispatching capabilities that lie at the heart of event-driven software applications. The model is based on the well-established Stochastic Reward Net (SRN) modeling paradigm. It considers the failures of the application in order to provide realistic performance estimates. We demonstrate the use of the model for the performability analysis of a VPN service provided by a Virtual Router. Since the reactor pattern is central to the event-driven software applications, the model proposed in this paper would enable design-time performability analysis of these applications.

Our future research involves identifying widely used patterns similar to the reactor pattern, and developing performability models of these patterns. Developing strategies to compose models of several patterns, and mirroring their use in software systems is also a topic of future research.

References

[1] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[3] D. Garlan and M. Shaw. *Advances in Software Engineering and Knowledge Engineering, Volume 1, edited by V. Ambriola and G. Torotora*, chapter An Introduction to Software Architecture. World Scientific Publishing Company, New Jersey, 1993.

[4] O. Ibe, A. Sathaye, R. Howe, and K. S. Trivedi. Stochastic Petri net modeling of VAXCluster availability. In *Proc. of Third International Workshop on Petri Nets and Performance Models*, pages 142–151, Kyoto, Japan, 1989.

[5] O. Ibe and K. S. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, December 1990.

[6] O. Ibe and K. S. Trivedi. Stochastic Petri net analysis of finite–population queueing systems. *Queueing Systems: Theory and Applications*, 8(2):111–128, 1991.

[7] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[8] Jane W. S. Liu. *Real-Time Systems.* Prentice Hall, New Jersey, 2000.

[9] B. Melamed and M. Yadin. Randomization procedures in the computation of cumulative-timed distributions over discrete-state markov process. *Operations Research*, 32(4):926–944, July-August 1984.

[10] J. Muppala, G. Ciardo, and K. S. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability: An International Journal Published by SAE Internationa*, 1(2):9–20, July 1994.

[11] Object Management Group. *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.

[12] A. Puliafito, M. Telek, and K. S. Trivedi. The evolution of stochastic Petri nets. In *Proc. of World Congress on Systems Simulation*, pages 3–15, Singapore, September 1997.

[13] S. Ramani, K. S. Trivedi, and B. Dasarathy. Performance analysis of the CORBA event service using stochastic reward nets. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 238–247, October 2000.

[14] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, 2000.

[15] C.U. Smith. *Performance Engineering of Software Systems.* Addison Wesley, 1990.

[16] H. Sun, X. Zang, and K. S. Trivedi. A stochastic reward net model for performance analysis of prioritized DQDB MAN. *Computer Communications, Elsevier Science*, 22(9):858–870, June 1999.

[17] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications.* John Wiley, 2001.