Automated Transformation of Distributed Software Architectural Models to Finite State Process

Omid Bushehrian
IT and Computer Department
Shiraz University of Technology
Shiraz,Iran
Bushehrian@sutech.ac.ir

Hassan Ghaedi
IT and Computer Department
Shiraz University of Technology
Shiraz,Iran
Hassan.ghaedi@yahoo.com

Reza Ghanbari Baghnavi
IT and Computer Department
Shiraz University Of Technology
Shiraz,Iran
R_gh101@yahoo.com

***Abstract*—Software Performance Engineering (SPE) represents the collection of software engineering activities with the purpose of identification, prediction and also improvement of software performance parameters in the early stages of software development life cycle. Various models such as queuing networks, layered queues, Petri Nets and Stochastic Process Algebras are suggested for modeling distributed systems. Particular ability of a model is the prediction and estimation of non-functional characteristic of one system before it has been made. The main problem is a method by which we can easily transform architectural software models into formal simulate able models. In this paper a method for automatic transformation of UML deployment and sequence diagrams into FSP(finite state process) model is presented, so that we can analyze the resulting model through discrete event simulation tools from the performance perspective. In the proposed transformation algorithm, different aspects of a software system such as: communication model of software objects, synchronization and physical deployment of objects are considered.**

Keywords-performance engineering – distributed systems – finite state process – simulation – performance evaluation

I. INTRODUCTION

Performance is an important matter in software systems; everything affects it, including the software itself and all underlying layers, such as operating system, middleware, hardware, communication networks, etc [1]. Software Performance Engineering (SPE) includes the collection of software engineering activities and related analysis that it's purpose is identification, prediction of performance problems and also improvement of performance parameters in software development cycle in the early stages [2].

Nowadays, The various types of analyzable formal models are used for modeling a distributed system, including queuing networks, layered queues, Petri Nets and Stochastic Process Algebras. Particular ability of a model is the prediction and estimation of non-functional characteristic of one system, before it has been made. One of non-functional characteristics is performance. The main problem is a method by which we can easily transform software models such as deployment and sequence diagrams into simulate able models in architectural level.

Current methods in performance modeling and evaluation haven't represented an exact and complete algorithm for the transformation of UML models into simulate-able models. These methods have only sufficed at examples of how transformation can be preformed [3,4,5]. Specifically, in the previous methods, the communication models of objects(synchronous or asynchronous), object types(active or reactive), creation of threads and the deployment of the objects haven't been considered when transforming a UML model into its corresponding formal model.

In this article an algorithm for automatic transformation of software deployment and sequence diagrams into FSP[13] as a simulate-able model is presented. So that by simulating the resulting FSP model different performance factors such as response times, length of queues, objects population can be measured. To achieve this, we use UML sequence diagrams that are annotated with SPT profile[13]. As shown in Fig.1 in this profile, performance tags can be augmented to sequence diagrams which explain the performance aspects of the modeled software.

II. RELATED WORKS

There are many researchers in the field of transforming distributed system models to analyzable formal models. Mainly queuing and Petri nets are used in this direction.

Some have used Petri nets for modeling timed or stochastic analysis[6,7,8]. Some have extended Petri nets for modeling to analyzing safety, reliability, or performability[9,10]. In [11] an algorithm is presented for automatic transformation of software architectural models into Petri net models for performance analysis. Queuing networks are another approach for modeling. In[12] automatic algorithms are presented for transforming architectural models into queuing models for performing various analysis. In[13] FSP models are used as destination model in transforming software architecture models.

In summary we can say that in presented algorithms in previous researches for transforming UML models to Petri and queuing models, following shortcomings are observed:

- Deployment diagrams that are particularly important in distributed software are not considered.

- In the previous methods, the issues such as communication models of objects(synchronous or asynchronous), object types(active or reactive), creation of threads and the deployment of the objects haven't been considered when transforming a UML model into its corresponding formal model.

In this article by selecting FSP model as the destination model of the transformation, a 1-to-1 transformation algorithm is presented for transforming deployment and sequence diagrams to FSP's.

III. FSP AND SPT PROFILE

FSP's, are abstract machines used to model the behavior of concurrent and distributed systems[14]. Each process of distributed system can be modeled by a FSP that perform sequence of actions repeatedly. A distributed system, is a collection of such processes that some of their actions are synchronized with each other. For transforming sequence and deployment diagrams to FSP's, we must transform objects, CPUS and communication links to FSP's. Each FSP is indeed an automata with several states that transits between states[15]. Fig.2 illustrates automata corresponding to the server object shown in Fig.1. In this section, the transformation algorithm is explained with an example.

A simple sequence diagram that is annotated with SPT[1] profile for performance, is illustrated in Fig.1[5].

The <<PAcontext>> stereotype indicates that this diagram is a scenario involving some resources (software objects in this case) driven by a workload. The objects are a server (an active object, indicated by the heavy box), and a lock. The annotation on the lifeline of the user object has a <<PAopenLoad>> stereotype indicating that it is a workload, i.e. it defines the intensity of the demand made on the system by the users of this scenario; in this case the interval between requests is exponentially distributed with a mean of 40 ms. A requirement that the mean response time is 30 ms is given, along with a placeholder variable ($Resp) for the predicted value that will be determined by simulation. The server offers a single operation, which requires the lock to be acquired and released - each of these operations takes 10 ms on average.

The translation scheme generates an FSP process for each object specified in the sequence diagram. We use the order of operations shown on the timeline of each object to determine the events available in each state. The sequence of messages specified in the sequence diagram for each object are encoded as actions in the FSP, e.g. *enqueue, lock* and *unlock.response*. The open arrowhead on the request message in Fig. 1 implies that several users may be requesting the server at the same time and hence that there is an implicit queue (of undefined capacity) at the server. This is made explicit in the FSP via the auxiliary Queue process.

1 - Schedulability, Performance and Time

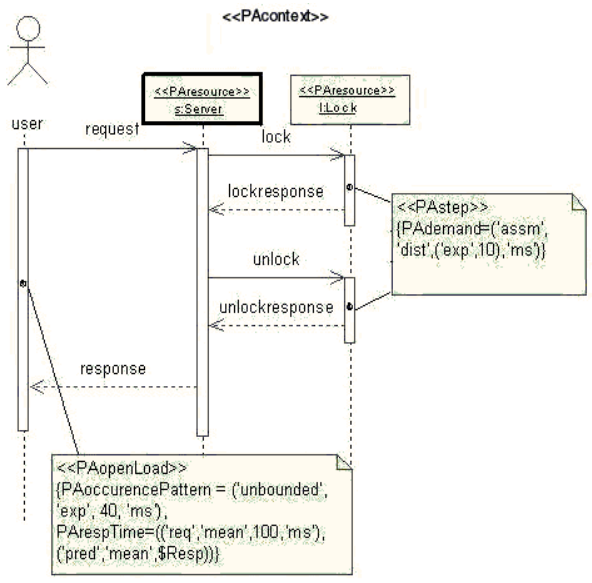

<<PAcontext>>
user
<<PAresource>> s:Server
<<PAresource>> l:Lock
request
lock
lockresponse
unlock
unlockresponse
response
<<PAstep>> {PAdemand=('assm', 'dist',('exp',10),'ms')}
<<PAopenLoad>> {PAoccurencePattern = ('unbounded', 'exp', 40, 'ms'), PArespTime=(('req','mean',100,'ms'), ('pred','mean',$Resp))}

Figure 1. A Simple Scenario

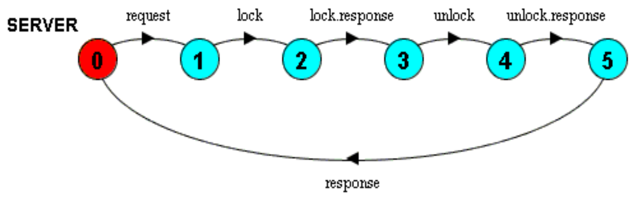


Figure 2. Automata Relevant To Server Process of Fig. 1

The process relevant to workload is expressed as following:

Requestworkload=(arrival<c:exp(1/40)>->?c?next->
Requestworkload).

The definition of queue for server active object is performed as following:

QUEUE= QUEUE [0],

QUEUE [i:0..N]=(when i<N enqueue-> QUEUE [i+1]

|when i>0 dequeue-> QUEUE [i-1]).

The process relevant to server active object is represented as following:

SERVER=(request->lock->lock.response->unlock->
unlock.response->response->SERVER).

The process relevant to lock passive object is as following:

Lock=(lock<c1:exp(1/10)>->?c1?lock.response->
unlock<c2:exp(1/10)>->?c2?unlock.response->Lock).

As shown in Fig.1., some method bodies include time delays and hence actions relevant to these methodse must wait.

to represent these delays, a concept named "clock" has been proposed in [14] . Within a process, an action can set a clock using the following instruction:

action<c:exp(parametr)>

This instruction sets clock c to an exponentially distributed value which the mean value of this distribution is denoted by *parameter*. The clock c can be checked for expiry by the following instruction:

?c?*action*

And it says that *action* is executable when the value of c reaches zero(i.e is expired). In Fig.1 there are two time delays for lock object defined as follows:

lock<c1:exp(1/10)>->?c1?lock.response->unlock<c2:exp(1/10)>

->?c2?unlock.response

For measuring the simulation time between two actions(events) the following instruction can be used:

timer W<variable1,variable2>

The first variable, is an action that starts the timer and the second variable, is an action that stops the timer and the resulted simulation time in between is kept in variable W. In Fig.1. the mean response times of the arrival requests is obtained by the following instruction and kept in Resp variable:

timer Resp<.request ,response>

after defining each object seperately, all of them should be simulated as a single concurrent system. We assume that the name of the system is *System*:

||System=(QUEUE||Resp||SERVER||Lock||Requestworkload)

For synchronizing object actions with each other , the following notation is used[14]:

/{arrival/enqueue,request/dequeue}.

The arrival workload is synchronized with the arrival of the queue and the queue output is synchronized with the arrival of the server object.

IV. Transformation Algorithm

Being available the sequence and deployment diagrams corresponding to one or more scenarios of a distribution software, for each scenario a collection of FSP's are created independently. Afterward, according to the deployment diagram of the system, a collection of processes corresponding to the computational nodes and network links represented in the deployment diagram is created as well. The whole distribution system is produced by the composition of created processes and appropriate synchronization of these processes. Deployment and Sequence diagrams may include active and passive objects, threads, synchronous and asynchronous invocations, creation of objects and threads, network links, CPU's and replicas that for each of them the the transformation method to FSP,s should be specified. Fig.3. indicates the general structure of our proposed transformation algorithm:

```
Algorithm FSPmodel UMLtoFSP(model m)
  Begin
    FSPmodel f=create FSPmodel();
    For each SequenceDiagram sd in m
      For each object o in sd
        f.AddObject(o);
    For each DeploymentDiagram d in m
      For each Node n in d
        f.AddNode(n);
      For each Link l in d
        f.AddLink(l);
    f.Synchronize(m),
end
```

Figure 3. Transformation Algorithm

A. Definitions

Sets O,L,M and C denote the set of objects, the set of links, the set of messages and the set of CPU's respectively. Each object *o* in O is specified by following multiplex that consists of name, type, node, first message, last message and the set of sent and received message by *o* respectively:

o:(Name, Type, Node, FirstMessage, LastMessage, MessageSet)

Each message *m* in M is specified by following multiplex that consists of name, sender object, receiver object, communication delay at the sender side, computational delay at the receiver side and message index respectively:

m:(Name,Sender,Receiver,Type,ComunicationDelay,
ComputationalDelay,Index)

Each link l in L is specified by following multiplex that consist of two various nodes:

l(N,M)

Each node *n* in N is specified by a node name and the number of CPU's installed on it.

n(Name,Number)

In the subsequent subsections, the transformation algorithm body shown in Fig.3 will be explained. Note that each function within the algorithm body generates a set of FSP codes and augments them to the final FSP model. *Emit()* method is used for this purpose.

B. AddObject() Function

For each object, if it is a thread or reactive one then we need a threadpool and if it is an active object, we need a queue:

```
AddObject(o):
  if(o.Type = active) then
//define Queue
    Emit(Queue«o.Name»=«o.Name»[0],
     Queue«o.Name»[index:0..QueueSize]=(when
    index<QueueSize «o.FirstMessage»[index]->
    Queue«o.Name»[index+1]|when index>0 «o.LastMessage»[index]->
    Queue«o.Name»[index-1]).)
```

Note that the parts enclosed in '<<' and '>>' symbols should be replaced with the values corresponding to the object for which the FSP is generated. For generating the FSP'code corresponding to an object like o, the sent and received messages by o, represented in the corresponding sequence diagram, are examined consecutively. Each message *m* can be described by an ordered pair <*messageType, messageDirection*> in which *messageType* can be synchronous(denoted by S), asynchronous (denoted by A) or return message(denoted by R) and *messageDirection* can be out-going (denoted by G) or incoming (denoted by I) . If *m* is an out-going message network link is acquired and if *m* is an incoming message CPU is acquired. The body of each object o is formed as a sequence of actions each corresponding to an out-going or incoming message. The getStatus() function in the following code returns the ordered pair <*messageType, messageDirection*> for each message m and object o:

```
// Object Body Transformation
  For each m in o.MessageSet
  Emit( «o.Name»=(
    switch( getStatus(m.Type,o) )
    case (S,I):
     Emit( «m.Name»->«o.Name»acquirecpu-> )
     case (R,I):
     Emit( «m.Name»->«o.Name»acquirecpu-> )
    case (S,G):
     Emit( «o.Name»acquirelink ->«m.Name»-> )
    case (R,G):
     Emit( «o.Name»acquirelink ->«m.Name»-> )
    case (A,G):
     Emit( «o.Name»acquirelink->«m.Name»-> Other[index],
    Other[index]=(otherwork[index]->Other[index]| )
    case (A,I):
    Emit(  «m.Name»->«o.Name»acquirecpu->Other[index],
    Other[index]=(otherwork[index]->Other[index]| )
    Else Emit(  «m.Name»->«o.Name»acquirecpu )
    Emit( «o.Name»). )
```

C. AddLink() and AddNode() Functions

For each node *n* a process is added to the FSP model as follows:

```
f.AddNode(n):
  Emit(Cpu«n.Name»=(getcpu«n.Name»->releasecpu«n.Name»->
  Cpu«n.Name»).)
```

Corresponding to each CPU installed on *n*, an instance of the above FSP is added once composing the whole system. For each communication link l an FSP is added to the model as follows:

```
  f.AddLink(l):
  Emit(Link«l.n»to«l.m»=(getlink«l.n»to«l.m»->
  releaselink«l.n»to«l.m» -> Link«l.n»to«l.m»).)
```

D. Synchronize() Function

In this section, the code generation algorithm for the synchronization among previously generated FSP's, is presented. First, the previously defined FSP's are composed together as a single concurrent system with '||' symbol:

```
  For all Links l in L
    Emit(Link«l.n»to«l.m»||)
  For all nodes n in N
    Emit( [1.. «.nnumber»]:Cpu«n.Name»||)
  For all Queues
    Emit(Queue«o.Name»||)
  For all objects o in O
    Emit( [T]:  «o.Name»||)
```

Variable T represents the number of object replicas corresponding to an active or re-active object. For synchronizing two FSP's P1 and P2 on action m1 and m2, where there are T replicas or threads available for P2 the action synchronization code is generated as follows:

```
    Emit(/{[T].m2/m1,)
```

Corresponding to each message *m* sent by object S and received by object R, the receiver object R should acquire CPU to perform the computation of *m*. Therefore there should be synchronization between *getcpu* action of the corresponding CPU process and *acquirecpu* action of R which is placed before action *m*:

```
«o.Name»acquirecpu/[i]getcpu «n.Name»
```

In the above synchronization, *[i]getcpu«n.Name»* denotes the *getcpu* action corresponding to the ith CPU process installed on node *n*. Node *n* is the node on which object *o* is deployed according to the deployment diagram. If a node n has more than one CPU installed on it, the allocation of CPUs to objects deployed on *n* should be specified before generating the synchronization codes.

For an object o deployed on Node *n* with T threads or replicas the code generation algorithm fairly divides the T threads among the number of CPU's *c* installed on *n*:

```
For each object o in O
    Emit(
     [0..T/c].«o.Name»acquirecpu/[1].getcpu«n.Name»,
    [(T/c)+1..2*T/c].«o.Name»acquirecpu/
    [2].getcpu«n.Name»,…
    [(c-1)/c*T+1..T].«o.Name»acquirecpu/
    [c]. getcpu«n.Name»
    )
```

The acquired CPU by an object *o* on receipt of message *m* should also release the CPU after completion of *m*. Therefore the FSP corresponding to o should contain *a releasecpu* action which follows action m. Moreover, there should be synchronization between *releasecpu* action of the corresponding CPU process and *releasecpu* action of object o. The generated code is similar to the one for *getcpu* synchronization (see the above code).

The sender object of each remote message *m* should acquire the connecting link between the node on which the sender resides and the node on which the receiver resides before sending *m*. Therefore the *acquirelink* action of the sender process of *m* and the *getlink* action corresponding to link process should be synchronized:

```
For each object o in O
  For each outgoing message m in o.MessageSet
    Emit(
     [T].«o.Name»acquirelink«m.Name»/
    getlink«m.sender.l»to«m.receiver.n»,
    [T].«m.Name»freelink«m.Name»/
    releaselink«m.sender.l»to«m.receiver.n»
    )
```

See the example sequence diagram shown in Figure 4:

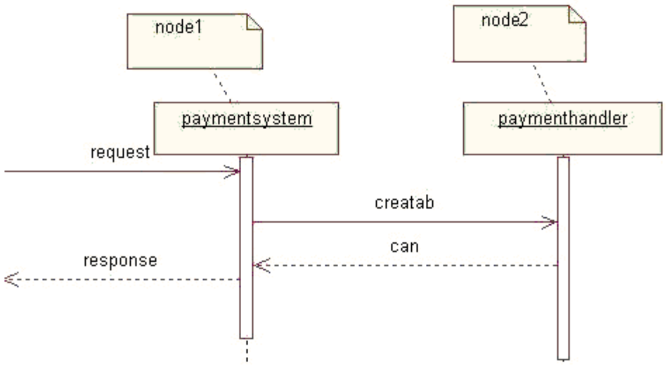


Figure 4. A Simple Sequence Diagram

Considering Fig.4, the *paymenthandler* FSP is defined as follows:

Paymenthandler=(createab->p_handleracquirecpu-> p_handleracquirelink -> can-> Paymenthandler).

With the assumption that there are two CPUS on the node which *paymenthandler* resides, and the link between the two nodes is Linkn$_2$ton$_1$, synchronizations are performed as follows:

[0..poolsize/2]. Paymenthandler /[1].getcpunode2

[poolsize/2+1..poolsize]. Paymenthandleracquirecpu/

[2].getcpunode2,

[0..poolsize/2]. Paymenthandleracquirelink/ [1].releasecpunode2,

[poolsize/2+1..poolsize]. Paymenthandleracquirelink/

[2].releasecpunode2

[T]. Paymenthandleracquirelink/ getlinkn2ton1,

[T].can/[T].releaselinkn2ton1

V. CASE STUDY

The case study is an electronic bill payment via the bank web site and for this case study we measure the population and throughput for each object and response time for each request. The Figs.6 illustrates some results. Sequence diagram of this system is illustrated in Fig.5. System objects are located on 4 nodes and each node has the number of CPUs. Bank, logger, payment_system and billing_systems' objects, are active objects of the system which require queue. Arrival rate is exponential with 20 value.

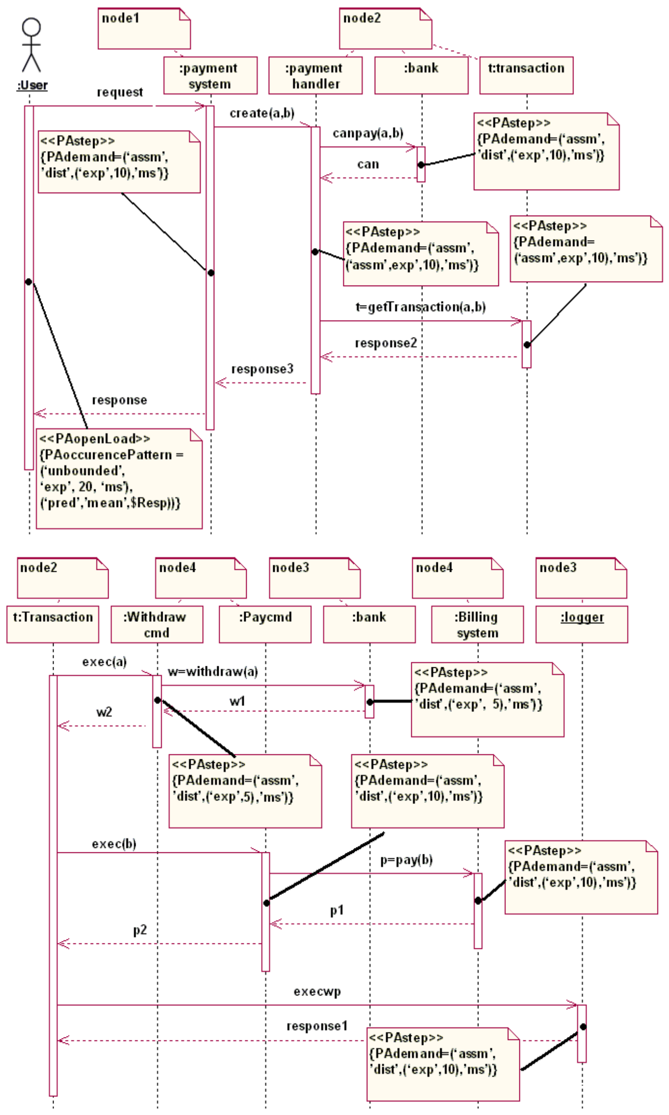
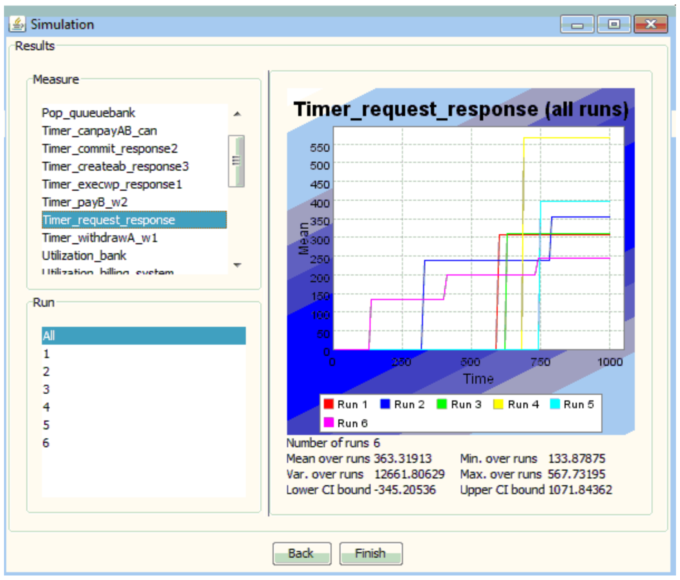

node1
node2
:User
:payment system
:payment handler
:bank
t:transaction
request
create(a,b)
canpay(a,b)
can
<<PAstep>> {PAdemand=('assm', 'dist',('exp',10),'ms')}
<<PAstep>> {PAdemand=('assm', ('assm',exp',10),'ms')}
<<PAstep>> {PAdemand= ('assm',exp',10),'ms')}
t=getTransaction(a,b)
response2
response3
response
<<PAopenLoad>> {PAoccurencePattern = ('unbounded', 'exp', 20, 'ms'), ('pred','mean',$Resp))}
node2
node4
node3
node4
node3
t:Transaction
:Withdraw cmd
:Paycmd
:bank
:Billing system
:logger
exec(a)
w=withdraw(a)
w1
w2
<<PAstep>> {PAdemand=('assm', 'dist',('exp', 5),'ms')}
<<PAstep>> {PAdemand=('assm', 'dist',('exp',5),'ms')}
<<PAstep>> {PAdemand=('assm', 'dist',('exp',10),'ms')}
exec(b)
p=pay(b)
p1
p2
<<PAstep>> {PAdemand=('assm', 'dist',('exp',10),'ms')}
execwp
response1
<<PAstep>> {PAdemand=('assm', 'dist',('exp',10),'ms')}

Figure 5. Electronic Bill Payment Sequence Diagram

In this case study, the number of requests that are performed simultaneously in system, are 15 requests. Time delays for each stage according to SPT profile are illustrated in Fig.5.



```
Number of runs 6
Mean over runs 363.31913      Min. over runs   133.87875
Var. over runs   12661.80629  Max. over runs  567.73195
Lower CI bound -345.20536     Upper CI bound 1071.84362
```

Figure 6. Mean Response Time Relevant To Arrival Request

VI. Conclusion and Future Works

In previous researches for transforming UML models to analyzable formal models, Deployment diagrams that are particularly important in distributed software are not considered. In the previous methods, the issues such as communication models of objects(synchronous or asynchronous), object types(active or reactive), creation of threads and the deployment of the objects haven't been considered when transforming a UML model into its corresponding formal model. In this paper with selecting FSP model as transformation destination model, an 1-to-1 transformation algorithm is presented for converting sequence and deployment diagrams of objects to FSP's. We applied this algorithm on our case study(Electronic bill payment) and measured factors that are important for performance. This case study has various characteristics such as synchronous and asynchronous calls, network communications(links), active and reactive objects. Though, FSP's can be simulated by LTSA, but there are cases in distributed systems that their modeling is not possible by FSP's particularly when several replicas of a resource(like CPU) exists. Modeling resource scheduling algorithms is not possible by FSP's as well. For the future work, our objective is to extend FSP models to overcome these deficiencies.

Refrence

[1] [1] Murray Woodside, Greg Franks, Dorina C. Petriu, The Future of Software Performance Engineering,2007

[2] [2] C.U. Smith, C. M. Lladó, V. Cortellessa, A.diMarco,L. Williams, "From UML models to software performance results: an SPE process based on XML interchange formats",in Proc WOSP'2005, Palma de Mallorca, 2005, . 87-98.

[3] S. Distefano,M. Scarpa, A. Puliafito, Software Performance Analysis in UML Models,2005

[4] Nima Kaveh and Wolfgang Emmerich, Deadlock Detection in Distributed Object Systems,2002.

[5] Andrew J. Bennett1, A. J. Field, and C. Murray Woodside." Experimental Evaluation of the UML Profile for Schedulability, Performance and Time",2005.

[6] G. Chiola, "Great SPN 1.5 software architecture," in Proc. 5th Int. Con$ Modeling Techniques and Tools for Comput. Perform. Eval., Torino,Italy, Feb. 1991,.117-132.

[7] G. Ciardo, J. Mu ala, and K. Trivedi, "SPNP: Stochastic Petri net package," in Proc. Conf Petri Nets and Performance Models, Kyoto,Japan, Dec. 1989, . 142-151.

[8] G. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon, "SARA (System Architect's A rentice): Modeling, analysis, and simulation su ort for design of concurrent systems," IEEE Trans. Software Eng.,vol. SE-12, no. 2, . 293-311, 1986.

[9] N. G. Leveson and J. L. Stolzy, "Safety analysis using Petri nets," IEEE Trans. Software Eng.,vol.SE-13,no.3, .386-397,1987.

[10] J. F. Meyer, "Performability modeling of distributed real-time systems,"in Mathematical Computer Performance and Reliability,Elsevier, 1984.

[11] Robert G.Pettit IV, Hassan Gomaa,Modelling Behavioral Pattern Of Concurrent Objects Using Petri Nets,international symposium on object and component-oriented real time distributed computing ,2006.

[12] F.Andolfi,F.Aquilani,S.Balsamo,and P.Inverardi. Deviving performance models of software architectures from message sequence charts. September 2000.

[13] Andrew J. Bennett and A. J. Field." Performance Engineering with the UML Profile for Schedulability, Performance and Time: a Case Study",2004.

[14] Alan Fekete, FSP Lectures, University of Sydney, 2004.

[15] Ashok Argent-Katwala, Allan Clark, Howard Foster,Stephen Gilmore, Philip Mayer, and Mirco Tribastone, Safety and Response-Time Analysis of an Automotive Accident Assistance Service,2008.