Proc 7th Workshop on Performability Modelling of Computer and Communications Systems (PMCCS7), Torino, Italy, Sept 2005, pp 89-92.

A Composable Performance Model for Service/Resource Systems

Murray Woodside
Dept. of Systems and Computer Engineering,
Carleton University, Ottawa K1S 5B6, Canada
cmw@sce.carleton.ca

Abstract

More and more systems have an architecture made up of resources which offer services at interfaces. An algebra with operators to compose services with each other and with resources, and to compose subsystems into systems, would make possible powerful compact descriptions of such systems, taking advantage of their particular structure. Practical modeling also often requires composition of submodels obtained from partial studies. Such an algebra can also support analysis models of many kinds; here we consider layered performance analysis. This paper outlines an algebra for composing layered queueing models.

1. Introduction and motivation

Modern complex systems are often built up by composing components and services, and a modeling language with corresponding power is essential. Process algebras have this capability, used by Hillston to create a compositional performance modeling framework PEPA [5] and initiate the fruitful field of Stochastic Process Algebras. These however have difficulty in solving for very large state spaces, which arise for instance in describing queueing of large numbers of concurrent entities.

Service systems with large state spaces include Web Services, systems with a Service-Oriented Architecture, Web Applications using J2EE or .NET or CORBA platforms, and legacy client-server applications. However, they have a more constrained class of behaviours, offering services in structured interactions between their components and their resources. Efficient performance solutions can be obtained by other means such as Layered Queueing Networks (LQNs), which are used here [2][3][11][13].

Two practical motivations for a compositional model for LQNs are, the need to mimic the system construction process, and the need to compose partial models made from measurements [6][7].

Software resource architectures were described in [12] as a combination of resources and operations, and the present work gives an algebra for these architectures. This paper sketches an algebraic system SRA (Service/Resource Algebra) for composing LQN models from simple elements and from LQN submodels. A more complete definition is in preparation. In particular, some special process types and operators are defined, with constraints on their application. They should be viewed as elements in a to-be-defined pattern system of process-algebraic definitions for service systems.

2. Service Systems

A System is made up of five types of process: Hosts, ServerTasks, Resources, Services, and Activities.

- a Host represents a device which executes the behaviour specified by a System,
- a ServerTask is a combination of a Resource and a set of Services that it offers, with a label defining a Host,
- a Resource accepts service calls into a queue, then uses a queueing discipline to assign resource units to calls, and dispatches its Services to serve them. It has labels defining a multiplicity, and a service discipline. Seen as a process, a resource has very constrained behaviour; it can only be claimed and released by requests,
- A Service is a process which is initiated by a call, with three patterns of response (as in LQNs);
  - an asynchronous response is concurrent with the calling process, and terminates;
  - a synchronous response generates a return to the calling process, as well as terminating;
  - a forwarding response forwards the request to another service, as well as terminating.
- an Activity is a sub-process of a Service.

To denote a process of a given type without stating its name, we will write :Type. Thus, :Service stands for some process of type Service.

Processes may be composed by the usual operators:

- sequential: P = AB indicates that the completion of process A triggers B;

- alternative: P = A | B means A or B. There may be a probability label (as also in PEPA) P = A (prob = pA) | B (prob= pB);
- parallel: P = A || B.

A process may have a repetition count, as in P = A(rep = n). A random number of repetitions may be indicated by a distribution, as in P = A(rep = Geometric(mean)). Repetition counts and probabilities are special cases of labels, as in P = A(labelName = value, ...). Label names are reserved words.

Every process defines a default interface, which is denoted by its name. A process may also have lists of offered and required services, as in P = [offered1, ...] A [required1, ...]. The elements of these lists are of type Service.

There are additional composition operators defined for composing Activities into Services, Resources and Services into ServerTasks, and ServerTasks into a System. These additional operators act as a shorthand for defined patterns of composition which embody the execution constraints of server system technology, and the performance model semantics of the LQN.

For simplicity the allocation of ServerTasks to Hosts is indicated not by composition but by a label of the ServerTask, as in Task = T(host = :Processor). The actual execution of any computational operation only begins when a Host executes it. Host processes are not described here.

To refer to properties of composed processes, a set of functions are defined:

- the processor of a given ServerTask, Service or Activity is host(element).
- the ServerTask of a given Service or Activity is the function task(element).
- the Service of a given Activity is service(:Activity).
- the set of serverTasks on a given :Host is tasks(:Host).
- the set of services offered or required by a ServerTask, System or Service is the function offServ(element) or reqServ(element). For Services, offServ( ) returns “self”.

3. Activities and Services

Activities and services are processes whose definition is constrained to suit the role.

- An *Activity* is a process with no internal parallelism, except parallelism which may be embedded in service call activities. It may have a label “demand” for its mean CPU demand, and a label “rep” for mean repetitions, as in act = A(demand = 1.5, rep = 2).
- A *Service request activity* or *call activity* is a placeholder for the behaviour of the requested service, up to a “return” event, which terminates the call activity. It can only be composed sequentially within an activity. It has labels
  - “rep” for the number of calls,
  - “callType” (with value = synchronous | asynchronous) for the type, and
  - “target” for the Service requested.
- A *ServiceProcess* is a process composed of activities. including parallel composition.

In a ServiceProcess, some activities may labeled as “return” activities, as anAct = A(return). They return control (and the result of the service) to the caller of the service, and this terminates the calling activity. However they do not terminate the service behaviour. If a Service receives a synchronous call and terminates without a “return” activity, control is returned by default at its termination. If a Service receives an asynchronous call it ignores return activities.

An alternative to “return” activities is “forwarding” activities, shown by a label as in Act = A(forward(aService, probF)), which forward the request (and the responsibility to return a result to the caller) to aService, with probability probF (or to return). They have the same constraints as “return” activities and can be mixed with them. Some examples of Service processes are:

```
aServiceP1 = act1
aServiceP2 = phase1(return) phase2
aServiceP3 = act1 act2 (act3(return) || act4) act5
aServiceP4 = act1 act2
  (act3(return) | act4(forward(exceptionService))) act5
```

If a process acquires a logical resource such as a semaphore, the execution within the resource context of the semaphore is regarded as a service of the semaphore. Since the semaphore must be released, the request is always synchronous.

A Service is defined by a serviceProcess and a list of labels giving properties of the service. The labels include its priority in the server:

aService = (:ServiceProcess)(list of service labels)

The set of Required Services (RS) for a Service are defined by the requests made by its Call Activities. Each call activity has a set RS with one element, the service it calls. Composing activities gives a set RS’ which is the union of their sets RS. The same is true for a serviceProcess and its Service. Required services can be displayed after a process name, as a list in square brackets, or returned as the function reqServ( ) (abbreviated here to rS( ). Suppose:

rS(act1) = [aService1]; rS(act2) = [aService2];
rS(act4 = [aService3];

then from the above definitions,

rS(aServiceP4) = [aService1, aService2, aService3]

4. Composition of Services

A ServerTask is composed of one Resource and one or more Services:

aServerTask = (:Service | :Service | ...)
|[ServT]| :Resource

where the special parallel composition operator |[ServT]| has these semantics:

- each :Resource enters only one such task in a system,
- a service operation requires a unit of the resource,
- the order of serving requests is determined by the queueing discipline of :Resource, and possibly by the priority of the services,
- the execution of sub-operations of any Service, relative to other services with the same host, is determined by the queueing discipline of host(:Resource).

The set of offered services of the task is just the set of Services that are composed. The set of required services is the union of the sets for the Services that are composed.

For a logical resource the ServerTask does not have a Host, and each of its Services may be executed on a different Host. This requires that each of these Services should immediately "call back" to a separate Service associated with the originating Host. This separate service (and its ServerTask) defines the part of the originating service executed within the context of the logical resource.

4.1 Merging ServerTasks

Submodels may be created that describe only some of the services of a task. To compose them, the tasks that occur in more than one are merged to one, with the union of their services. We define:

aServerTask1 = Merge(aResource) (list of :ServerTask)

such that the merged aServerTask1 has the union of services of the :ServerTask arguments, and aResource as its task Resource. In the union of services, duplicates are removed. A service is a duplicate of another if its service definition is the same and its properties are compatible (for example, if relative priorities can be adjusted to maintain the original relationships within the original subgroups).

The merge operator has been implemented for merging together tasks in two submodels created from different scenarios of the same system [6][7].

5. Composition of ServerTasks

A System is an assembly of ServerTasks composed by connecting required services to offered services, and assigning hosts to tasks. The assembly can be defined by listing the tasks, with the bindings of required to offered services, and of tasks to hosts.

System = |[Assemble]| ((list :ServerTask), bindings) (1)

A subtype of ServerTask in Eq (1) is a placeholder for a variety of component subsystems, called a Slot, which has only the offered and required interfaces.

For performance modeling purposes a system definition must include elements that define the sources of interactions (the system's users). These are defined by a distinct type of SourceTask that offers no services, and has just one Service element (rather mis-named in this case) which cycles forever and generates requests for service. The Resource of this task models the users or sources, with a multiplicity representing the number of users or sources that have this behaviour. Its Host is usually a set of hosts, one per source. Its required services are the users demands on the system. It enters Eq (1) as a subtype of ServerTask. In LQN literature, SourceTasks are described as "reference tasks".

Required services that are not bound form a list RS(:System). If it is non-empty the System is incomplete and must be treated as a "component subsystem". As a subsystem it also may have a list OS with a subset of the services offered within it, and can be written [OS]System[RS].

5.1 Composition of "component subsystems"

For LQNs, a sublanguage called CBML was defined to describe component submodels [14][15], which are models with offered and required services. CBML is compatible with UML component notation. In SRA a component subsystem can enter an Assembly in the same way that Slots and ServerTasks are, in Eq (1). Also, if a System has a Slot then a component subsystem can be fitted into it by the same Assembly operation applied to the Slot and the System:

System2 = |[Assemble]|(:Slot, aSystem; list of
bindings)

but for the argument :Slot, the offered and required services are interchanged (from the view inside the Slot, a Service it offers is required from the component, and vice versa).

6. Reconfigurations

When a processor or task fails the system configuration changes due to the failure and to a recovery subsystem. Das described a FTLQN (fault-tolerant LQN) [1] to capture some of these effects, and we will try to incorporate it. An augmented FTSystem model has additional parameters: each Host or ServerTask has a "status" label defined by status = active | ready | failed, and each Call Activity has a prioritized list of standby or candidate target Services (rather than just one target, as described above). Given values of the status variables, a

Configure operation can applied to an FTSystem definition to give a System:

FTSystem = System, (status attributes), (Service lists)
aSystem = Configure (FTSystem, strategy)

For a hot-standby strategy each Call Activity targets the first non-failed service in its ordered list. Failures of ServerTasks propagate to their clients, and failures of Hosts propagate to their tasks and Services. The configured System may offer fewer services, or service at reduced performance. Load-balancing and active replication strategies produce their own variants.

7. Relationship to LQN Notation

The elements defined here correspond closely but not exactly to the LQN notation used for the LQNS solver [3]. The following table describes the correspondences.

Service/Resource Algebra	**LQN**
Host	Processor (the same)
ServiceTask	Task (the same)
Resource	Task (an aspect)
SourceTask	Task (reference task)
Service	Entry (the same)
Activity	Activity (some differences)
Call	Call (the same)
Offered Services	Offered Services (CBML)
Required Services	Req. Services (CBML only)
Slot	Slot (CBML only)

In a layered system the tasks can be ordered into layers such that all required services are in a lower layer.

8. Conclusions

Related work includes Hillston's description of PEPA [5] and other work on process algebras. SRA has fundamental differences from PEPA, in particular time delays in SRA are deferred to analysis of the LQN. SRA is not a stochastic process algebra. The work [4] has some of the same goals as ours but uses different techniques and targets Stochastic Petri Nets for performance solutions.

Another related area is the algebraic definition of software architectures, by Garlan and Shaw, and others (e.g. [10]). This relationship is potentially important, since the intention of LQN is to model the performance of software systems from an architecture definition.

This extended abstract has described the beginning of a plan to create a formal framework for manipulating models of service systems, for performance and availability analysis using LQNs. Some features of LQNs have been incorporated, but others would be useful, such as parameterized definitions (for defining experiments over ranges of parameters) and replication by activities, by tasks and by subsystem (as in [8]).

9. References

[1] O. Das and M. Woodside, "Computing the Performability of Layered Distributed Systems with a Management Architecture," in *Proc. 4th Int. Workshop on Software and Performance (WOSP 04),* Redwood Shores, Calif., Jan 2004, pp. 174 – 185.

[2] G. Franks, A. Hubbard, S. Majumdar, J. Neilson, D.C. Petriu, J.A. Rolia and C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", *Performance Evaluation*, vol. 24, pp 117-136, 1995.

[3] Greg Franks, "*Performance Analysis of Distributed Server Systems*", PhD. thesis, Carleton Univ. Jan. 2000.

[4] V. Grassi, R. Mirandola, "Towards Automatic Compositional Analysis of Component Based Systems", *Proc 4th Int. Workshop on Software and Performance*, Redwoood Shores, CA, Jan. 2004, pp 59-63.

[5] J. Hillston, *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[6] C. Hrischuk, J. A. Rolia, and C. M. Woodside, "Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype", *Proc. 3rd Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '95),* Durham, NC, Jan 1995, pp. 399-409.

[7] T.A. Israr, D.H. Lau, G. Franks, M. Woodside, "Automatic Generation of Layered Queuing Software Performance Models from Commonly Available Traces", *Proc. 4th Int. Workshop on software and Performance (WOSP 05),* July 2005.

[8] T. Omari, G. Franks, M. Woodside, A. Pan, "Solving Layered Queueing Networks of Large Client-Server Systems with Symmetric Replication", *Proc. 4th Int. Workshop on Software and Performance (WOSP 05),* July 2005.

[9] J. A. Rolia and K. C. Sevcik, "The Method of Layers," *IEEE Trans. on Software Engineering*, v. 21, n. 8 pp. 689-700, Aug 1995.

[10] M. Shaw and D. Garlan, *Software Architecture*, Prentice-Hall, Inc., 1996.

[11] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", *IEEE Transactions on Computers*, v 44, n 1, pp. 20-34, January 1995.

[12] C. M. Woodside, "Software Resource Architecture", *Int. Journal on Software Engineering and Knowledge Engineering (IJSEKE),* v 11, n 4, pp. 407-429, 2001.

[13] M. Woodside, "*Tutorial Introduction to Layered Modeling of Software Performance*", Edition 3.0, May 2002 (Accessible from http://www.sce.carleton.ca/rads/ lqn/lqn-documentation/tutorialg.pdf)

[14] X.P. Wu and M. Woodside, "Performance Modeling from Software Components," in *Proc. 4th Int. Workshop on Software and Performance* (WOSP 04), Redwood Shores, Calif., Jan 2004, pp. 290-301.

[15] Erik Putrycz, Murray Woodside, and Xiuping Wu, “Performance Techniques for COTS Systems”, *IEEE Software*, v. 22, n 4, pp. 36–44, July-August 2005.