Activity Graph Reducer - agr

1.0 Getting Started with agr

agr is a java program and so must be interpreted by the java virtual machine. To reduce an activity graph, agr is invoked from the command line as follows:

java agr [-s substitutionfilename] [-h] [-l] < activityGraphFile

-s is used to specify a file containing the values of parameters used within an activity graph. For more information of parameter substitution, see section “1.6 Parameterization” on page 7.

-h tells agr to print the reduced activity graph in a human readable format

-l tells agr to print the reduced activity graph in a format which lqns can understand

In order for agr to work, the user must first set up his/her environment in order to run the java virtual machine. On Carleton’s SCE network, that requires two things:

1) Append /usr/local/jdk/bin to the executable path
2) Append /usr/local/jdk/lib to the library path

In addition, a CLASSPATH environment variable must be defined which points to the directory in which agr’s .class files reside. To define such a variable, add the following line to your .cshrc file:

setenv CLASSPATH <directory>

The simplest thing to do would be to set up your current directory as the CLASSPATH and invoke agr from the directory in which you installed it. , ie

setenv CLASSPATH .

And with that, you should be ready to run agr. If you find any bugs (make sure they are bugs and not syntax errors), please report them to me (Marc). email is:

courtois@sce.carleton.ca

2.0 Syntax

All activity graphs will have the following structure:

- Service Library Section
- Service Definition Section
- Graph Definitions Section

1.1 Notation

Throughout this manual, anything within angular brackets, <>, will refer to a parameter whose value must be provided by the user. Within any code fragment, words which are not in italics represent syntax which is defined elsewhere.

1.2 Service Library Section

This section is optional. It is used to specify the directories in which other service files and subgraph definitions can be found. If this section is not provided, the current directory will be used. To define the list of libraries in which other services and/or subgraphs are defined, the required syntax is:

ServiceLibsBegin
    <path name>
    <path name>
    ...
ServiceLibsEnd

1.3 Service Definition Section

This section is mandatory. It is used to specify the various services which will be used within the activity graphs defined in the next section. A service can be a primitive device (like CPU or DISK), an external service (like a file server), or a logical service which is made up of requests to other services. When defining the services, one service must be specified as the 'host'. The host refers to the processor on which the activities in the graph will run. The syntax to define the services is:

ServiceDefsBegin
    service defintion
    service defintion
    ...
ServiceDefsEnd

1.3.1 A Service Definition

The syntax for an individual service definiton is:

<a service>

or

HOST <a service>

or

FILE < a file name>

or

<a service> { <a service> <demand>; <a service> <demand>; ...}

To specify a non-host, primitive device/external service, simply provide its name. If the device is the host, include the keyword *HOST* before the name of the service.

If the services are defined in another file, include the keyword *FILE* followed by the name of the file. The parser will scan through the libraries provided in the Service Library Section, in the same order, and will use the first file it encounters with the same name. The file containing the service definitions must be of the same syntactical format as the activity graph file but need not include any graph definitions. The parser will only read in the service library and service definitions section found in that file.

A logical service can also be built on top of other services. To define a logical service, provide its name followed by a list within curly brackets, { }, of demands to other services. The name of the service used by a logical service can be defined anywhere in the Service Definition Section, and can refer to either a primitive device/external service, or to another logical service. The demand imposed on that service is a real number specifying the number of requests made to the lower level service per invocation of the logical service. Each service demand provided in the list must be followed by a semi-colon, ; .

1.4 Graph Definitions Section

At least one activity graph must be defined in this section. The first graph defined will be considered the 'main' graph. Other graphs defined beneath it can serve as subgraphs, or procedures, which are used within the main graph. An activity graph is defined using a script language containing a number of control structures found within other programming languages. In particular, the graph definition language allows the user to specify Switch constructs, Loop constructs, Procedure calls (though subgraphs) as well as Forks and Joins. The overall syntax for a graph definitions is:

GraphBegin <graph name>
  activity definition
  activity definition
  ...
GraphEnd

1.4.1 An Activity Definition

An activity can either be:

1) a simple activity
2) a reply activity
3) a procedure call ie a subgraph
4) a loop activity
5) a switch activity
6) a fork activity

- **1.4.1.1 A Simple Activity**

A simple activity is defined in the same way as a logical service. That is

<an activity name> service demands

where service demands is defined as:

{ <a service> <demand>; <a service> <demand>; ...}

The list of service demands defines the number requests made by the activity to the various services. Zero demands need not be specified.

- **1.4.1.2 A Reply Activity**

This activity exists for the purposes of integration with lqns. If the user's goal is simply to reduce the activity graph, then this type of activity is unnecessary. The syntax to define a reply activity is:

Reply <reply entry id> service demands

where *reply entry id* is the entry id of a task in an lqns model. Since the lqns context is often unknown at the time the activity graph is constructed, the *reply entry id* will typically be parameterized. Parameterization is discussed in section "1.6 Parameterization" on page 7.

- **1.4.1.3 A Procedure Call - Subgraph**

For the purposes of clearer activity graph definitions, activity graphs can be modularized by placing commonly used fragments into a subgraph. To invoke that subgraph within the main activity graph, the following syntax is required:

<a subgraph name> (G)

If the subgraph has not been defined beneath the main graph, the parser will scan through the libraries defined in the Service Library Section and will look for a file with the same name as the subgraph. It will then parse this file in. One of the graphs defined within the subgraph file must have the name of the required subgraph.

- **1.4.1.4 A Loop Activity**

To perform a sequence of activities repeatedly, a loop construct is used. The syntax of a loop activity is:

LoopBegin service demands *<iterations>*
activity definition
activity definition
...
LoopEnd service demands

The two list of service demands are optional, but can be used to specify the overheads of a loop construct. The service demands of the two overheads as well as those from the activities within the loop will be multiplied by the number of loop iterations to determine the overall service demand imposed by the loop.

- **1.4.1.5 A Switch Activity**

To model an if/else or a switch/case construct, a switch activity is used. Each case within the switch is assigned a probability of occuring. The syntax is as follows:

SwitchBegin service demands
CaseBegin <probability>
activity definition
activity definition
...
CaseBegin <probability>
activity definition
activity definition
...
...
SwitchEnd service demands

The overhead service demands of *SwitchBegin* and *SwitchEnd* are, once again optional. When reducing a switch activity, the service demands from the activities within each case will be summed and multiplied by the probability of the case. These demands, in turn, will be added to the overhead demands to provide the overall service demand impose by the switch.

- **1.4.1.6 A Fork Activity**

To model fork/join activity within an application, the fork activity is used. Its syntax is as follows:

ForkBegin service demands
    PartBegin
        activity definition
        activity definition
        ...
    PartBegin
        activity definition
        activity definition
        ...
    ...
ForkEnd service demands

As with the Switch and the Loop activities, the overhead demands for *ForkBegin* and *ForkEnd* are optional. The overall fork activity is not reducible. Only the sequence of activities within each part can be reduced.

1.5 Comments

Anything in between /* and */ will be ignored.

1.6 Parameterization

It may be desirable to parameterize your activity graph. That is, the user may wish to outline the *structure* of the activity graph but post pone the assignment of service times, device names, activity names or entry ids until run time. To do so, variables can be used in place of constant values. A variable is denoted as follows:

$<ANAME>

where *<ANAME>* is any alphanumeric string beginning with a letter. So, for instance, if the user wishes to parameterize the cpu and disk demand of activity1, the description of activity1 in the activity graph will look like:

actvity1 {cpu $cpuDemandOfActivity1; disk $diskDemandOfActivity1; network 1;}

Similarly, if the user wishes to parameterize the *reply entry id* in a reply activity, the syntax would look like:

reply $replyEntryId {cpu 10000;}

Prior to running agr, the user will then have to create a parameter substitution file where each line in the file contains a variable name followed by the value to use for that variable. If the two previous examples were in the same activity graph, the substitution file would look something like this:

$cpuDemandOfActivity1   15000
$diskDemandOfActivity1 2.5
$replyEntryId   clientRequest

To inform agr of the name of the substitution file, agr can be invoked with the -s option. That is, an invocation of agr with substitutions for a human readable output will look like:

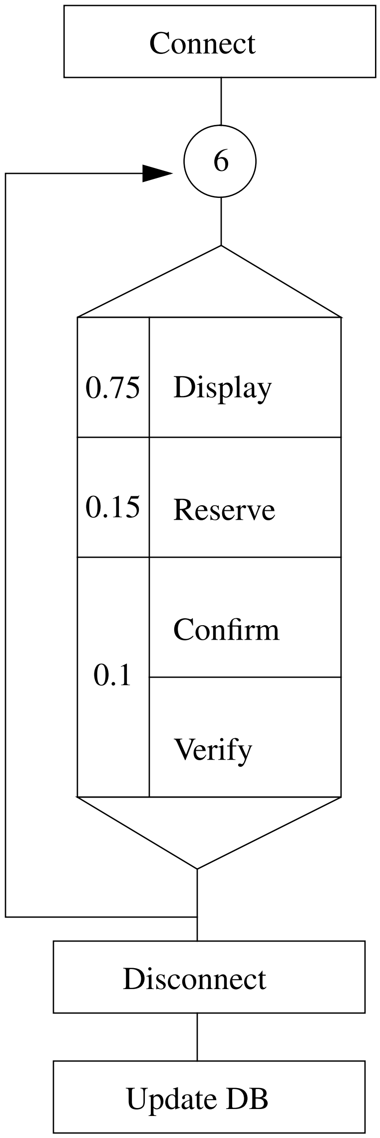
java agr -s substitutionFile -h < activityGraphFile

When the -s option is specified, agr reads in the substitution file and preprocesses the activity graph, transforming each variable in the graph to its corresponding value in the substitution file. If a variable is used in the graph but a value is not provided for it in the substitution file, the preprocessing step will simply remove the leading $ from the variable name. If that results in bad syntax, the actual agr parser will flag the problem.

1.7 Reduction

Provided there are no fork/joins within an activity graph, agr will roll the graph into a single activity. Any construct, ie switch, loop, subgraph, which contains a fork/join will be considered non-reducible. If a sequence of activities within a construct contains 1 or more non-reducible activities, the reduction process will collapse the reducible activities into 1 cumulative activity. The sequence of activities will then be updated to contain only the non-reducible activities and the 1 cumulative activity. The overall construct will be marked as non-reducible. In the case of a switch activity, if any case is non-reducible, the remaining cases will not be merged into the overall switch activity, though the sequence of activities within these cases may well be collapsed into cumulative activities.

3.0 Example - Ticket Reservation System



This example is an activity graph for a ticket reservation system, taken from Performance-Oriented Patterns in Software Design, Woodside, p 40. The service demands of each activity are shown in Table 1.

Table 1: Service Demands

Activity	CPU Million instrs	Database Server operations	Netware Server operations	CCReq Server operations
Connect	0.010		1	
Loop begin overhead	0.001			
Switch begin overhead	0.001			
Display	0.005	1	1	
Reserve	0.015	2	1	
Confirm	0.002		1	
Verify	0.004	1		1
Disconnect	0.001		1	
Update DB	0.007	1		

2.1 Activity Graph, No Logical Services, No Forks

Using the syntax described earlier, this activity graph can be described as follows:

```
ServiceDefsBegin
	HOST cpu
	databaseServer
	netwareServer
	ccReqServer
ServiceDefsEnd

GraphBegin TicketReservation
	connect { cpu 0.010; netwareServer 1;}
	LoopBegin {cpu 0.001;} 6
		SwitchBegin { cpu 0.001;}
			CaseBegin 0.75
				display { cpu 0.005; databaseServer 1; netwareServer 1;}
			CaseBegin 0.15
				reserve { cpu 0.015; databaseServer 2; netwareServer 1;}
			CaseBegin 0.1
				confirm { cpu 0.002; netwareServer 1;}
				verify { cpu 0.004; databaseServer 1; ccReqServer 1;}
		SwitchEnd
	LoopEnd
	disconnect { cpu 0.001; netwareServer 1;}
	updateDB { cpu 0.007; databaseServer 1;}
GraphEnd
```

2.1.1 Output

Now suppose this definition was stored in a file, tickrsrv.ag. To reduce the graph and to print it in both a human-readable and an lqns-readable format the user would type:

java agr -h -l < tickrsrv.ag

agr would produced the following output:

```
/*
 a39:
  connect
  SwitchBegin
  CaseBegin
  display
  CaseBegin
  reserve
  CaseBegin
  confirm
  verify
  SwitchEnd
  LoopBegin
  LoopEnd
  disconnect
  updateDB

*/
graphbegin TicketReservationSystem
 simpleactivity {ccReqServer 0.6;databaseServer 7.9;netwareServer 8.0;cpu 0.0696;} /*a39*/
graphend


y a39 ccReqServer 0.6
y a39 databaseServer 7.9
y a39 netwareServer 8.0
s a39 0.0696
:
```

The first section of output described what activities have been roled into the cumulative activities. Since there are no fork/joins, the entire graph is collapsed into 1 activity, *a39*. The second section of output displays the collapsed graph in the same syntax as the input. The third section describes the reduced graph in a format lqns can understand.

2.2 Activity Graph with Logical Services, No Forks

In the previous activity graph, the various servers had been treated as external services. Now suppose we know that databaseServer and NetwareServer run locally, and suppose we know the demand they impose on the host cpu and disk. Table 2, defines their service demands:

Table 2: Service demands of Database and Netware Servers

Service	CPU M-instr	Disk operations
Database Server	0.085	2
Netware Server	0.012	1.5

With this new information, we can modify the Service Definition Section of our activity graph to turn databaseServer and netwareServer into logical services. That is:

```
ServiceDefsBegin
        HOST cpu
        databaseServer { cpu 0.085; disk 2;}
        netwareServer { cpu 0.012; disk 1.5}
        ccReqServer
        disk
ServiceDefsEnd
```

2.2.1 Output

Running agr this time, produces the following output:

```
/*
 a45:
  connect
  SwitchBegin
  CaseBegin
  display
  CaseBegin
  reserve
  CaseBegin
  confirm
  verify
  SwitchEnd
  LoopBegin
  LoopEnd
  disconnect
  updateDB

*/
graphbegin TicketReservationSystem
 simpleactivity {disk 27.8;ccReqServer 0.6;cpu 0.8371;} /*a45*/
graphend


y a45 disk 27.8
y a45 ccReqServer 0.6
s a45 0.8371
:
```

As can be seen, the demands are reduced to those for primitive/external services.

2.3 Activity Graph with Logical Services and a Fork

One final twist would be to insist that the reserve activity forks off a logging activity which executes in parallel. By doing this, the switch activity and the overall loop become non-reducible. The graph definition would look like:

```
ServiceDefsBegin
    HOST cpu
    databaseServer { cpu 0.085; disk 2;}
    netwareServer { cpu 0.012; disk 1.5}
    ccReqServer
    disk
ServiceDefsEnd

GraphBegin TicketReservation
    connect { cpu 0.010; netwareServer 1;}
    LoopBegin {cpu 0.001;} 6
        SwitchBegin { cpu 0.001;}
            CaseBegin 0.75
                display { cpu 0.005; databaseServer 1; netwareServer 1;}
            CaseBegin 0.15
                ForkBegin { cpu 0.001;}
                  PartBegin
                     reserve { cpu 0.015; databaseServer 2; netwareServer 1;}
                  PartBegin
                     log { cpu 0.005; disk 1;}
                ForkEnd
            CaseBegin 0.1
                confirm { cpu 0.002; netwareServer 1;}
                verify { cpu 0.004; databaseServer 1; ccReqServer 1;}
        SwitchEnd
    LoopEnd
    disconnect { cpu 0.001; netwareServer 1;}
    updateDB { cpu 0.007; databaseServer 1;}
GraphEnd
```

2.3.1 Output

The agr output would look like:

```
/*
 a91:
  LoopBegin

 a98:
  SwitchBegin

 a105:
  display

 a136:
  ForkBegin

 a143:
  reserve

 a160:
  log

 a171:
  ForkEnd

 a174:
  confirm
  verify

 a205:
  SwitchEnd

 a207:
  LoopEnd

 a68:
  connect
  disconnect
  updateDB

*/
```

```
graphbegin TicketReservationSystem
 loopbegin {cpu 0.0010;} /*a91*/
 6.0
  switchbegin {cpu 0.0010;} /*a98*/
   casebegin 0.75
    simpleactivity {disk 3.5;cpu 0.102;} /*a105*/
   casebegin 0.15
    forkbegin {cpu 0.0010;} /*a136*/
     partbegin
      simpleactivity {disk 5.5;cpu 0.197;} /*a143*/
     partbegin
      simpleactivity {disk 1.0;cpu 0.0050;} /*a160*/
    forkend /*a171*/
   casebegin 0.1
    simpleactivity {disk 3.5;ccReqServer 1.0;cpu 0.103;} /*a174*/
  switchend /*a205*/
 loopend /*a207*/
 simpleactivity {disk 5.0;cpu 0.127;} /*a68*/
graphend
```

```
s a91 0.0010
s a98 0.0010
y a105 disk 3.5
s a105 0.102
s a136 0.0010
y a143 disk 5.5
s a143 0.197
y a160 disk 1.0
s a160 0.0050
y a174 disk 3.5
y a174 ccReqServer 1.0
s a174 0.103
y a68 disk 5.0
s a68 0.127
:
a91->a98
a98->(0.75)a105+(0.15)a136+(0.1)a174
a136->a143&a160
a143&a160->a171
a105+a171+a174->a205
a205->a207
6.0*a91->a68
```

2.4 Activity Graph with Logical Services and a Sugraph with a Fork

The same thing could have been achieved if we converted the Reserve activities into a sub-graph. That is:

```
ServiceDefsBegin
    HOST cpu
    databaseServer { cpu 0.085; disk 2;}
    netwareServer { cpu 0.012; disk 1.5}
    ccReqServer
    disk
ServiceDefsEnd

GraphBegin TicketReservation
    connect { cpu 0.010; netwareServer 1;}
    LoopBegin {cpu 0.001;} 6
        SwitchBegin { cpu 0.001;}
            CaseBegin 0.75
                display { cpu 0.005; databaseServer 1; netwareServer 1;}
            CaseBegin 0.15
                Reserve (G)
            CaseBegin 0.1
                confirm { cpu 0.002; netwareServer 1;}
                verify { cpu 0.004; databaseServer 1; ccReqServer 1;}
        SwitchEnd
    LoopEnd
    disconnect { cpu 0.001; netwareServer 1;}
    updateDB { cpu 0.007; databaseServer 1;}
GraphEnd

GraphBegin Reserve
    ForkBegin { cpu 0.001;}
        PartBegin
            reserve { cpu 0.015; databaseServer 2; netwareServer 1;}
        PartBegin
            log { cpu 0.005; disk 1;}
    ForkEnd
GraphEnd
```

2.4.1 Output

The output of agr is identical except that it indicates that a subgraph substitution has taken place:

```
/*
 a79:
  LoopBegin

 a86:
  SwitchBegin

 a93:
  display

 a126:
  ForkBegin

 a133:
  reserve

 a150:
  log

 a161:
  ForkEnd

 a164:
  confirm
  verify

 a195:
  SwitchEnd

 a197:
  LoopEnd

 a56:
  connect
  disconnect
  updateDB

*/
```

```
graphbegin TicketReservationSystem
 loopbegin {cpu 0.0010;} /*a79*/
 6.0
  switchbegin {cpu 0.0010;} /*a86*/
   casebegin 0.75
    simpleactivity {disk 3.5;cpu 0.102;} /*a93*/
   casebegin 0.15
    graphbegin Reserve
     forkbegin {cpu 0.0010;} /*a126*/
      partbegin
       simpleactivity {disk 5.5;cpu 0.197;} /*a133*/
      partbegin
       simpleactivity {disk 1.0;cpu 0.0050;} /*a150*/
     forkend /*a161*/
    graphend
   casebegin 0.1
    simpleactivity {disk 3.5;ccReqServer 1.0;cpu 0.103;} /*a164*/
  switchend /*a195*/
 loopend /*a197*/
 simpleactivity {disk 5.0;cpu 0.127;} /*a56*/
graphend

s a79 0.0010
s a86 0.0010
y a93 disk 3.5
s a93 0.102
s a126 0.0010
y a133 disk 5.5
s a133 0.197
y a150 disk 1.0
s a150 0.0050
y a164 disk 3.5
y a164 ccReqServer 1.0
s a164 0.103
y a56 disk 5.0
s a56 0.127
:
a79->a86
a86->(0.75)a93+(0.15)a126+(0.1)a164
a126->a133&a150
a133&a150->a161
a93+a161+a164->a195
a195->a197
6.0*a79->a56
```