DXM – Demos2k eXperiments Manager

Brian Monahan
HP Laboratories
HPL-2008-173

Keyword(s):
Demos2k, simulation, analytics, security

Abstract:
Applied models in Demos2k, a semantically well-founded process simulation language, are fundamentally stochastic in their nature. This means that a single run is at best a mere example of the possible range of behaviour - making it necessary to run these models many times over, Monte-Carlo style. This allows us to build a more statistically reliable picture of the overall systems behaviour and how varied it is. In this paper, we introduce DXM - the Demos2k eXperiments Manager - a software application that is designed to help applied analyst/modellers construct and manage repeated experimental simulation runs over a range of parameters. By using our DXM application, analyst/modellers can design their simulation experiments, run them and gather up the results for further analysis. Once analysts have obtained their experimental runs, they can use the DXV tool - Demos2k eXperiments Viewer - to help visually inspect and display these results. A number of Appendices present overall documentation for the current DXM and DXV packages.

External Posting Date: October 21, 2008 [Fulltext] - Approved for External Publication

Internal Posting Date: October 21, 2008 [Fulltext]

© Copyright 2008 Hewlett-Packard Development Company, L.P.

DXM – Demos2k eXperiments Manager

14th October 2008

Brian Monahan
Systems Security Lab
HP Laboratories, Bristol
BS34 8QZ, UK

Abstract

Applied models in Demos2k, a semantically well-founded process simulation language, are fundamentally stochastic in their nature. This means that a single run is at best a mere example of the possible range of behaviour – making it necessary to run these models many times over, Monte-Carlo style. This allows us to build a more statistically reliable picture of the overall systems behaviour and how varied it is. In this paper, we introduce DXM – the Demos2k eXperiments Manager – a software application that is designed to help applied analyst/modellers construct and manage repeated experimental simulation runs over a range of parameters. By using our DXM application, analyst/modellers can design their simulation experiments, run them and gather up the results for further analysis. Once analysts have obtained their experimental runs, they can use the DXV tool – Demos2k eXperiments Viewer – to help visually inspect and display these results. A number of Appendices present overall documentation for the current DXM and DXV packages.

1 Introduction

Demos2k is a semantically well-founded process simulation language (see Appendix E.2), together with a software application for executing models. Such models will typically rely upon several random variables sampled from a range of distributions, parameterised on the values of certain numerical parameters. The upshot of this is that a single run of the simulation is at best a mere example of the possible range of behaviour. Accordingly, analyst/modellers will need to run these models many times over, Monte-Carlo style, to get a statistically reliable picture of the overall aggregate behaviour and to see how broad it's variation is.

As a result, systems described by a Demos2k model often involve concurrent and secondly, tend to heavily depend upon fundamentally stochastic elements, such as random variates drawn from specific distributions. To get a reasonably accurate picture of the overall behaviour, we therefore need to run these repeatedly in general, given a set of parameters. Although DXM does provide some very basic statistical analysis capability, modellers will typically need to use external statistical packages to obtain deeper analysis of their simulation data.

We introduce DXM in terms of a small example, a three Tier service chain (with Customers), and show how Demos2x and DXM work together to define the experiments and to explore their outcomes.

1.1 Demos2k - A Very Brief Introduction

Systems descriptions written in Demos2k tend to be high-level, pleasingly short and to the point. The modelling approach supported is very much akin to 'extreme modelling', where the systems analyst/modeller can rapidly construct high-level models representing the customer's focus of interest. A key part of this comes form the way that probability distributions are used to abstract away from extraneous detail.

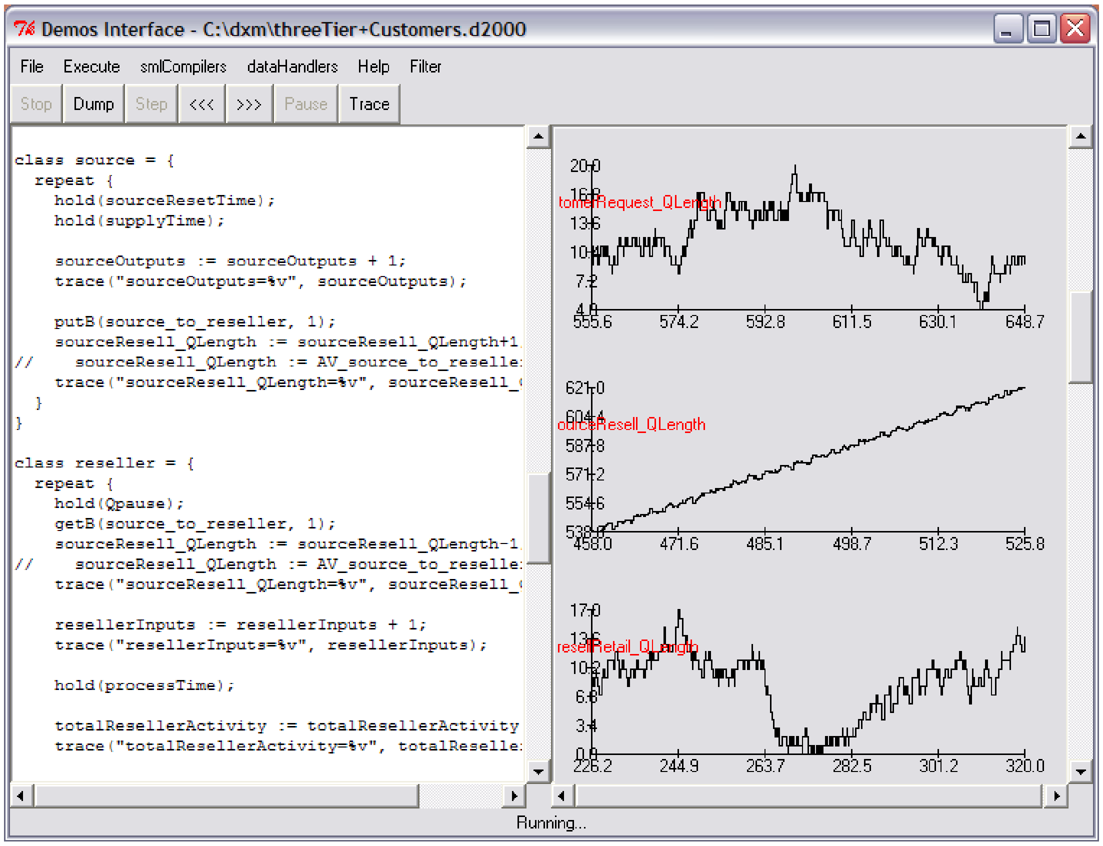
Much of Demos2k is concerned with the allocation of resources and the use of queues (or bins). The short example in Figure 1 illustrates this.

```
cons activity = normal(4, 1.2);

class doWork = { getR(staff, 3); hold(activity); putR(staff, 3); }

do 4 { entity(doWork, doWork, 2); }
```

Figure 1: A small fragment of Demos2k code

The first line defines `activity` as a *normally distributed random variate*, with mean 4 and variance 1.2 (Note: **not** standard deviation). This is used to specify time durations. The process class `doWork` describes a process that simply claims a number of staff (3 in this case) from a resource pool called `staff`, performs some work whose duration is specified by `activity` and then returns the claimed staff for further duty. Note that 'doing work' here is simply modelled by the passing of (simulation) time. If, for any reason, there were not enough staff to get, the **getB** construct will *block* (i.e. wait) until there is enough available. The final line simply launches four instances of `doWork` after a delay of 2 time units.

Some further discussion can be found in Appendix E. In addition, the full source of the extended example we use in the next section can be found in Appendix F. The screenshot in Figure 2 shows the Demos2k tool and GUI in action.


Demos Interface - C:\dxm\threeTier+Customers.d2000
File Execute smlCompilers dataHandlers Help Filter
Stop Dump Step <<< >>> Pause Trace
tomerRequest_QLength
ourceResell_QLength
resellRetail_QLength
Running...

```
class source = {
  repeat {
    hold(sourceResetTime);
    hold(supplyTime);

    sourceOutputs := sourceOutputs + 1;
    trace("sourceOutputs=%v", sourceOutputs);

    putB(source_to_reseller, 1);
    sourceResell_QLength := sourceResell_QLength+1
//    sourceResell_QLength := AV_source_to_reselle
    trace("sourceResell_QLength=%v", sourceResell_
  }
}

class reseller = {
  repeat {
    hold(Qpause);
    getB(source_to_reseller, 1);
    sourceResell_QLength := sourceResell_QLength-1
//    sourceResell_QLength := AV_source_to_reselle
    trace("sourceResell_QLength=%v", sourceResell_

    resellerInputs := resellerInputs + 1;
    trace("resellerInputs=%v", resellerInputs);

    hold(processTime);

    totalResellerActivity := totalResellerActivity
    trace("totalResellerActivity=%v", totalReselle
```

Figure 2: The Demos2k tool and GUI – see Appendix E

2 Three-tier service chain - with customer demand

We illustrate DXM and associated tools such as DXV by considering a hypothetical, but reasonably substantial, systems dynamics example: the three tier service chain, with customer demand.

2.1 Our example

Imagine that we have the following macro-scale (but simplified) set-up for a particular product line. At the head of the service chain are the original source suppliers of the product, followed by intermediary value-add resellers and lastly, the retailers. We further add customer demand into the mix as a further constraint/stimulus on the dynamics of the system. A simplified picture of our system is in Figure 3.

$$\begin{tikzcd}\text{source} \arrow[r] & \text{reseller} \arrow[r] & \text{retail} & \text{customer} \arrow[l]\end{tikzcd}$$

Figure 3: A picture of a three-tier service chain, with customer demand

What makes this example interesting is that there are several stages in delivery from end-to-end, whose behavioural dynamics may be affected by the numbers of elements available for each stage. A more detailed picture of the situation is given by Figure 4.

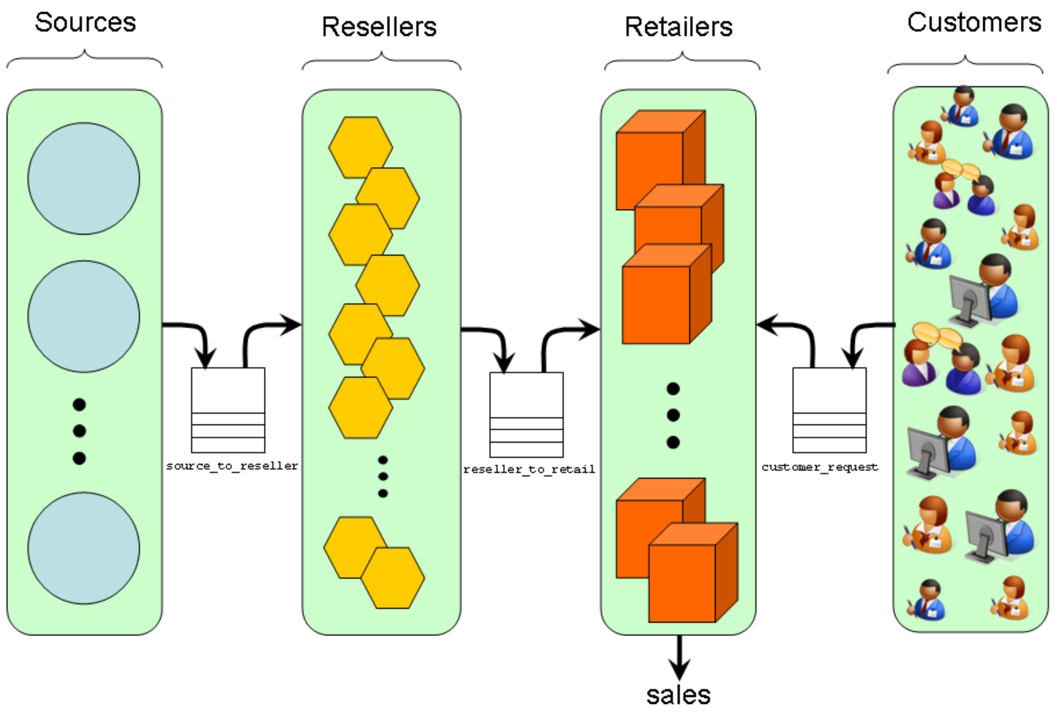


Figure 4: A more detailed picture of a three-tier service chain, with customer demand

However, we recognise that this account of service chains could have been made significantly more elaborate; for our purposes, the current example is sufficiently interesting to illustrate how DXM itself is used.

3 Modelling our example using Demos2k

We now briefly walk through the representation or *model* of our example in Demos2k. We shall quickly go through the four classes used to represent the individual instances that populate each stage and also talk about the model parameters – these take on greater importance vis-a-vis DXM.

3.1 The customer process class

Each customer is modelled by a process that simply (metaphorically) fires off customer requests for the product (admittedly, a somewhat idealistic model of a customer!) by placing them into the `customer_request` bin.

```
class customer = {
  repeat {
    hold(requestInterval);

    putB(customer_request, 1);
    customerRequest_QLength := customerRequest_QLength + 1;
    trace("customerRequest_QLength=%v", customerRequest_QLength);

    totalCustomerRequests := totalCustomerRequests + 1;
    trace("totalCustomerRequests=%v", totalCustomerRequests);
  }
}
```

Figure 5: The customer class

3.2 The source process class

Sources are modelled here by a process that repeatedly places completed product into the `source_to_reseller` bin and keeps track of how many items have been shipped out. Making or supplying each item is modelled by holding for a certain amount of (simulation) time.

```
class source = {
  repeat {
    hold(sourceResetTime);
    hold(supplyTime);

    sourceOutputs := sourceOutputs + 1;
    trace("sourceOutputs=%v", sourceOutputs);

    putB(source_to_reseller, 1);
    sourceResell_QLength := sourceResell_QLength+1;
    trace("sourceResell_QLength=%v", sourceResell_QLength);
  }
}
```

Figure 6: The source class

3.3 The reseller process class

The Resellers are modelled by processes that repeatedly wait for items in the `source_to_reseller` bin, processes them for a certain amount of time (i.e. `processTime`) and then pushes the product onto the `reseller_to_retail` bin.

```
class reseller = {
  repeat {
    hold(Qpause);
    getB(source_to_reseller, 1);
    sourceResell_QLength := sourceResell_QLength-1;
    trace("sourceResell_QLength=%v", sourceResell_QLength);

    resellerInputs := resellerInputs + 1;
    trace("resellerInputs=%v", resellerInputs);

    hold(processTime);

    totalResellerActivity := totalResellerActivity + 1;
    trace("totalResellerActivity=%v", totalResellerActivity);

    putB(reseller_to_retail, 1);
    resellRetail_QLength := resellRetail_QLength+1;
    trace("resellRetail_QLength=%v", resellRetail_QLength);

    resellerOutputs := resellerOutputs+1;
    trace("resellerOutputs=%v", resellerOutputs);
  }
}
```

Figure 7: The reseller class

3.4 The retail process class

The retail processes are different from the other kinds of process seen so far, as they repeatedly wait until *both* a customer request and a product from a reseller are present *together*. When they are, that event then counts as a *sale*.

```
class retail = {
  repeat {
    try [getB(customer_request, 1), getB(reseller_to_retail, 1)] then {
      customerRequest_QLength  :=  customerRequest_QLength-1;
      resellRetail_QLength     :=  resellRetail_QLength-1;

      trace("customerRequest_QLength=%v", customerRequest_QLength);
      trace("resellRetail_QLength=%v", resellRetail_QLength);

      totalSales := totalSales + 1;
      trace("totalSales=%v", totalSales);
    }
  }
}
```

Figure 8: The retail class

3.5 Model parameters

Throughout the code shown above are a number of entities and constants which determine things like how frequently a process runs or the number of instances of each kind of process. These quantities essentially determine the *character* of the dynamic behaviour of the model. Thus a model parameter can technically be any declared **cons** element in a Demos2k model. The DXM tool then provides a way to specify how these parameters are varied to produce a set of experiment variations that may then be executed. Figure 9 presents the (default) model parameters used for our example.

```
cons runTime = 1000;

cons numOfCustomers     = 1;
cons numOfSourceProc    = 1;
cons numOfResellerProc  = 1;
cons numOfRetailProc    = 1;

cons customerAvgRequestInterval  =  4;
cons sourceResetAvg              =  4;
cons sourceAvgSupplyTime         =  4;
cons resellerAvgProcessTime      =  4;

cons requestInterval   =  negexp (customerAvgRequestInterval);
cons sourceResetTime   =  normal (sourceResetAvg,          sourceResetAvg/5);
cons supplyTime        =  normal (sourceAvgSupplyTime,     sourceAvgSupplyTime/10);
cons processTime       =  normal (resellerAvgProcessTime,  resellerAvgProcessTime/12);
```

Figure 9: Model parameters (defaults)

Notice that some of these parameters are essentially random variates generated by distribution. We regard even these to be a part of the parameter set as this also allows us to change the distributions themselves and experiment with different alternatives.

3.6 Model outputs

What is the output result of running a Demos2k model? Each run produces a trace reporting the real-valued simulation time at which certain events occurred. The **trace** statement is used in models to report the current value of particular numerical variables within both the Demos2k graphical interface and the output trace.

Demos2k models can declare and use (global) variables to record the behaviour observed within the model during simulation runs. Figure 10 lists the particular variables whose values we choose to gather during runs of our example.

3.6.1 Using `demos_sample_tick`

Unfortunately, there is something of a catch with just using trace statements in the way suggested above. The trace statements produce trace entries that are unweildy and bulky and thus require significant processing to extract the appropriate information. This becomes especially expensive in terms of both the time taken and storage space consumed during longer-running experiments. To counter this, we have adapted Demos2k so that entries for all global variables can be systematically written out to a specified CSV (Comma Separated Value) file on an appropriate signal.

But then we needed to decide how to trigger this output and to do so conveniently from within a Demos2k model. The solution adopted was to use the change/modification of a particular, fixed global variable called '`demos_sample_tick`'. In this way, we obtain a simple-to-use and pragmatic means of yielding outputs that is also under the control of the modeller and not imposed by Demos2k itself.

```
// Activity of reseller
var totalCustomerRequests = 0;
var totalResellerActivity = 0;

// Queue lengths
var customerRequest_QLength  =  0;
var sourceResell_QLength     =  0;
var resellRetail_QLength     =  0;

// Numbers of outputs
var sourceOutputs    = 0;
var resellerInputs   = 0;
var resellerOutputs  = 0;

// Sales
var totalSales = 0;

// DXM bureaucracy
var demos_sample_tick = 0;
```

Figure 10: Model Outputs)

DXM relies on this technique being used within models to obtain the CSV output files containing the results. To avoid potentially significant wasted effort, DXM pragmatically performs a syntactical check to ensure that `demos_sample_tick` is defined within the models it manages. Additionally, the CSV output files are named according to a convention that DXM relies upon for gathering results; accordingly, it is not necessary for the modeller to specify how these files are named explicitly.

3.6.2 Trace output vs. `demos_sample_tick`

The outputs obtained by the `demos_sample_tick` approach produce a single vector of values - these are usually generated on a regular basis by some measurement process. On the other hand, trace statements output a single value, at any point in the process. In principle, using trace statements can exhibit greater variation than the `demos_sample_tick` method - because the latter generally produces output on a regular schedule, irrespective of what processes are doing.

4 So, what's the question?

Finally, having set up our example, we are now ready to go ... but wait, what was the question that our model should help provide answers to? Naturally, we should have had this in mind when formulating the model in the first place. Anyway, it is rather unwise to proceed further with using DXM until this issue is settled and understood.

The reason for this caution is clear – it is because the next stage is to set-up and define an experiments plan that DXM can execute and perform repeated runs of. The experimental question to be answered will need to be formulated in terms of making explicit variations of the parameters identified in the Demos2k model.

Once identified, we can then use DXM to explore the space. The successful use of DXM intrinsically depends upon the particular Demos2k model, it's parameters and the simulation data to be observed. Whether all this effort is sufficient to help gain useful insights can, in the final analysis, only be answered by the modeller and systems analyst. Caveat emptor.

So, for the sake of argument, it seems natural to suppose that we are interested in identifying situations (i.e. sets of parameter values) which systematically yield good sales figures. The next major section shows how DXM and friends can help us to explore that question using our executable Demos2k models.

5 Doing experiments with DXM

As hinted at earlier, we now need to create a*DXM experiments plan* that specifies how the parameters of interest are to be varied.

Broadly speaking, each combination of the parameters will produce a specific *experiment variation*; one that gets its own uniquely named directory into which experiment results are placed. This in turn involves automatically generating a copy of the Demos2k model instantiated to the specific values of the parameters, corresponding to the particular experiment variation. The resulting model is then repeatedly executed to produce a series of output CSV files, one for each run.

Once the runs are complete, the `stats.csv` file is compiled that lists the final values of the output variables gathered for each run. This typically makes sense for cumulative count variables that form the majority of the variables output. Averages and other statistical information (e.g. max/min, std dev. and std error) are then computed for each variable across all the runs and tabulated at the end of the `stats.csv` file.

The overall averages and other variation data summarised in the `stats.csv` are then added to the `summaryStats.csv` file. Thus, the`summaryStats.csv` file contains the averages from each experiment variation. Briefly, we have that:

- `stats.csv`: Contains final values of output variables for each run and their computed averages and other statistical information for a *particular* experiment variation.
- `summaryStats.csv`: Contains aggregated statistics across each of the experiments variations.

5.1 How does Demos2k perform randomisation for repeated runs?

The very short answer to that question is that it doesn't - instead, it does something more better and more useful.

The Demos2k system provides a standard, well-behaved, deterministic pseudo-random number generator that always starts from the same place for the first run. On the second and subsequent runs, the system uses the final state reached in the preceding run to initialise the RNG for the next, forthcoming run.

This subtlety ensures that a *single* sequence of high-quality pseudo-random numbers is available for the simulation. Each run of a model appears to be different, essentially because it starts from a different place in this single sequence.

The clear advantage of this approach is that the Demos2k simulations are deterministically 'replayable', whilst also appearing to provide an apparently unpredictable source of random input for use within the simulation.

Most importantly, the way that Demos2k produces runs makes it possible to *share* Demos2k models amongst your colleagues – it makes Demos2k models portable. This is because a sequence of runs will always produce the same outputs, guaranteeing that everyone sees the same set of results no matter who runs them or how often they are rerun. A further essential advantage is that Demos2k models can then be tested out in a deterministic way, thus making debugging feasible.

5.2 Identifying the parameter space

As we said earlier, identifying the parameter space to be varied is largely a matter of the question one is hoping to answer or at least obtain insight into. We offer here some general heuristics to help make appropriate choices:

- Initially start broadly - conduct a broad and wide search to get some idea of the overall behavioural variation. The downside is that this is often very expensive in terms of time taken and the storage space consumed – however, it is frequently necessary to help argue that important, relevant conditions and variation have been adequately captured by the search.
- Try to ensure that the parameters to be varied encode relevant properties. This means that different choices of value for each parameter typically make some kind of difference to the outcomes under investigation - otherwise they should have already been eliminated. If the model's outcome is insensitive to the value of a parameter then the particular value it has doesn't have significant impact - and so can be set conveniently and 'traded-off' when setting other more sensitive parameters.

  It is also the case that it is hard to know for sure which parameters are insensitive – for example, it is fairly easy to get *conditionally insensitive parameters* – variables whose degree of insensitivity depends upon the values of other parameters.

  The upshot is to try and ensure that only effective and meaningful parameters are to be varied systematically. Unfortunately it is possible that the only way to know this is to have already explored the parametric variation you were seeking to avoid! Sometimes, there is no other way except to just try it out and see what happens.
- When all else fails, try out small-scale parametric variations around specific combinations, holding most parameters constant. This will help get some ideas about the overall shape and if any particular ranges seem to be interesting.

5.3 Experiment plans

The objective of a DXM experiment plan is to describes how certain model parameters are varied. Figure 11 contains a plan for our example. The syntax of DXM experiment plans is defined in Appendix A.4.

```
# ttc1.dxm - DXM file for the threeTier+Customers example
#

      file = threeTier+Customers.d2000
      rootdir = ttcExample

      runs = 100

      param numOfCustomers      : 1; 10; 50;
      param numOfSourceProc     : 2; 4;  10;
      param numOfResellerProc   : 5; 10; 20;
      param numOfRetailProc     : 2; 10; 20;

      param customerAvgRequestInterval :  1; 10; 20; 100;
      param sourceResetAvg             :  0.1
      param sourceAvgSupplyTime        :  1;
      param resellerAvgProcessTime     :  1;

      param runTime                    : 1000

      naming = ttc+num=%n-%n-%n-%n+time=%a-%a-%a-%a+%Z
```

Figure 11: An experiments plan for our example

The first line of the plan specifies the Demos2k model to use in our investigation (i.e. `threeTier+Customers.d2000` in our case). This is followed by a specification of the **root directory** for all the experiments to be performed (i.e. `ttcExample`). The directories for each experiment variation are to be created within this root directory. The `runs` statement specifies how many times the model is run, for each experiment variation (i.e. in our case, 100 times).

The `param` statements specify a particular parameter to be varied, together with the set of discrete values it will be assigned to. Each parameter and its values thus contribute to forming the set of experiment variations. Note that one of the parameters is the `runTime` itself - this allows the modeller to easily change, via DXM, the amount of time spent simulating *within* each run performed.

Finally, the `naming` statement specifies a *pattern* to allow *structured naming* of the directories made for each experiment variation. In some sense, the pattern exploits the ordinal position of the parameters themselves (i.e. relative ordering of each `params` statement) to create a unique naming for each directory. This issue is covered in greater detail in Appendix A.5.

5.4 The amount of work to be done

We can immediately see from the plan how many experiments will be run and calculate the total number of runs across all the experiments.

There are 4 parameters having three alternative values and one parameter with four alternatives. All other parameters have only one value. This makes $324 = (3 \times 3 \times 3 \times 3) \times 4 = (81 \times 4)$ experiments to be done in total. Each experiment involves 100 runs, making $32,400$ runs overall. Finally, each run has a runtime of 1000 time units.

5.5 Executing the plan

Having now got our plan, we need to run or execute it using DXM. There are two major stages involved in doing this:

1. Creating the directory structures corresponding to each experiment variation and populating them with appropriate content.
2. Performing the runs for each of the experiment variations and constructing appropriate summary information.

5.5.1 Creating the experiments

To create the experiments using DXM, we simply type the following at the command line:

```
$ dxm -create ttc1.dxm
```

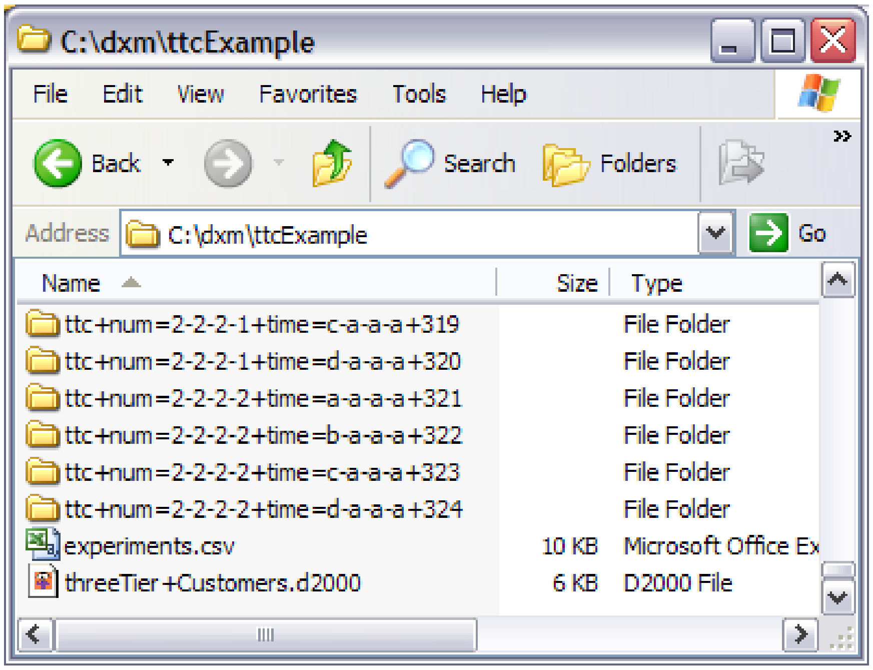
A screenshot of the directory structure this produces is given in Figure 12.

The name of the top-level root directory (i.e. *ttcExample* here) for the experiments created by DXM is specified by the plan. Importantly, the plan file will reside in the *parent* of the top-level root directory i.e. wherever DXM was run from.

The content of the root directory includes a (master) copy of the original Demos2k model. This is used to create the experiment variants of the model to be run later.

The root directory also contains the `experiments.csv` file. This file usefully presents the mapping from directory names to the specific values of parameters that each experiment variation tests against. This is very useful information when analysing the results later on.

The initial content of each experiment variation is illustrated in Figure 13. The file `params.csv` contains the specific parameter values for the variation, and the file `script.d2000` contains the specific version of the Demos2k model in the root directory.



Name	Size	Type
ttc+num=2-2-2-1+time=c-a-a-a+319		File Folder
ttc+num=2-2-2-1+time=d-a-a-a+320		File Folder
ttc+num=2-2-2-2+time=a-a-a-a+321		File Folder
ttc+num=2-2-2-2+time=b-a-a-a+322		File Folder
ttc+num=2-2-2-2+time=c-a-a-a+323		File Folder
ttc+num=2-2-2-2+time=d-a-a-a+324		File Folder
experiments.csv	10 KB	Microsoft Office Ex
threeTier+Customers.d2000	6 KB	D2000 File

Figure 12: Screenshot showing the directory structure for our DXM plan

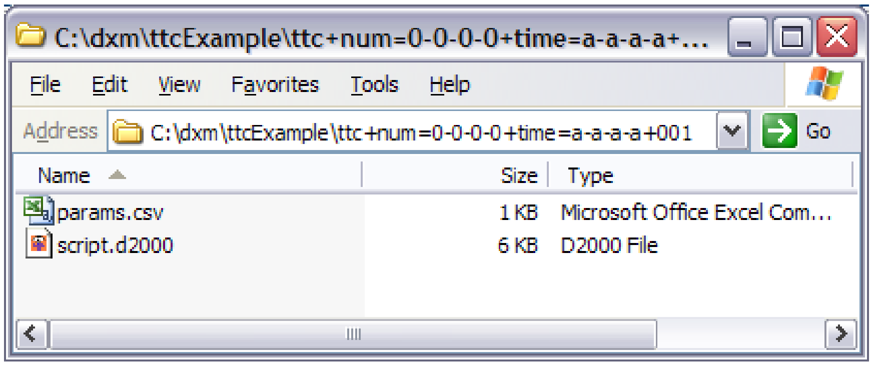
5.5.2 Running the experiments

Having now created the experiments variations, we execute the experiments themselves. We do this by simply typing the following at the command line:

```
$ dxm -run ttc1.dxm
```

This launches a (detached) process that executes the plans, allowing the user to do other work in the same shell. It is possible to monitor how far the simulation has got by typing the following:

```
$ dxm -list ttc1.dxm
```


C:\dxm\ttcExample\ttc+num=0-0-0-0+time=a-a-a-a+...
File Edit View Favorites Tools Help
Address
C:\dxm\ttcExample\ttc+num=0-0-0-0+time=a-a-a-a+001
Go

Name	Size	Type
params.csv	1 KB	Microsoft Office Excel Com...
script.d2000	6 KB	D2000 File

Figure 13: Screenshot showing initial content for an experiment

5.6 Looking at the results

Once the runs have been completed, it is then time to inspect what we have got. The (slightly edited) screenshot in Figure 14 shows the resulting content in the top-level directory.

In addition to the files `threeTier+Customers.d2000` and `experiments.csv` that we had initially, we now have:

`summaryStats.csv`:
: Summary stats file describing outcomes from all the experiments - this file was also briefly described in § 5.

`expt.log`:
: A log file describing overall progress such as the starting/stopping times for each experiment.

`breakdownStats.csv`:
: This CSV file contains more detailed information than `summaryStats.csv` and is organised by variable rather than by experiment and thus provides capability for drill-down. It contains all the stats summary information contained in the individual `stats.csv` files.

We also have an additional directory, called `stats`. This contains the copies of the `stats.csv` files from each experiment whose runs all succeeded. If an experiment had any failed runs (i.e. where the simulation itself failed), the stats file is copied instead into a `failedStats` directory. This `failedStats` directory is only created once some experiment has a failed run.

Significantly, this approach ensures that the information leading to a failed experiment is captured for later failure analysis and diagnosis of what went wrong. A failed experiment is often very informative as these can often result from genuine extreme dynamic behaviour arising naturally from within the model (e.g. singularities), rather than routine programming-type errors.

5.6.1 Results from each experiment

Each experiment directory now contains a typically large number of CSV results files - these are the outputs from each simulation run[1]. The screenshot in Figure 15 illustrates the results content of a typical completed experiment.

Apart from the `params.csv` and `script.d2000` files we had initially, we now have:

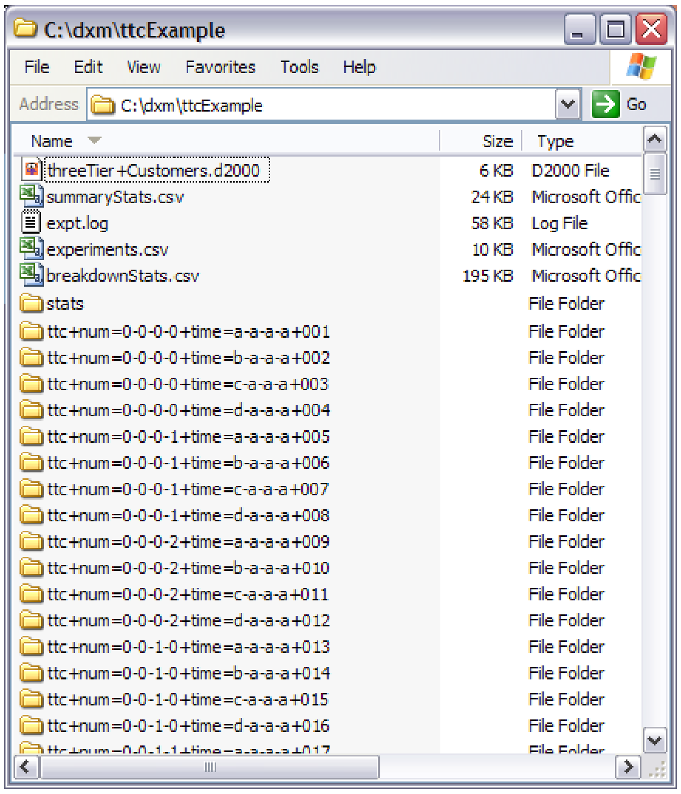
`exp_1.csv` **...** `exp_100.csv`:
: The CSV files output as a result of the ithsimulation run.

`sim.log`:
: The log of the simulations run - this records some information about starting and stopping and, in the case of a failed simulation, a brief failure message.

`seedData.csv`:
: A CSV file that contains the starting RNG seeds and stopping RNG seeds for each simulation run. This information can be used with the `dxms` to rerun particular runs of a specific model - see Appendix D.

`stats.csv`:
: This CSV file firstly contains the final line of run information from each of the CSV run files `exp_*.csv`. Secondly it contains calculated statistical information for each variable (i.e. column) such as its average value, the standard deviation, the standard error, and max/min values. The content of this file has been discussed earlier in § 5.

[1]Standard Demos2k trace files are typically not generated - we only use the more compact CSV files for data analysis.



C:\dxm\ttcExample
File Edit View Favorites Tools Help
Address C:\dxm\ttcExample
Go

Name	Size	Type
threeTier+Customers.d2000	6 KB	D2000 File
summaryStats.csv	24 KB	Microsoft Offic
expt.log	58 KB	Log File
experiments.csv	10 KB	Microsoft Offic
breakdownStats.csv	195 KB	Microsoft Offic
stats		File Folder
ttc+num=0-0-0-0+time=a-a-a-a+001		File Folder
ttc+num=0-0-0-0+time=b-a-a-a+002		File Folder
ttc+num=0-0-0-0+time=c-a-a-a+003		File Folder
ttc+num=0-0-0-0+time=d-a-a-a+004		File Folder
ttc+num=0-0-0-1+time=a-a-a-a+005		File Folder
ttc+num=0-0-0-1+time=b-a-a-a+006		File Folder
ttc+num=0-0-0-1+time=c-a-a-a+007		File Folder
ttc+num=0-0-0-1+time=d-a-a-a+008		File Folder
ttc+num=0-0-0-2+time=a-a-a-a+009		File Folder
ttc+num=0-0-0-2+time=b-a-a-a+010		File Folder
ttc+num=0-0-0-2+time=c-a-a-a+011		File Folder
ttc+num=0-0-0-2+time=d-a-a-a+012		File Folder
ttc+num=0-0-1-0+time=a-a-a-a+013		File Folder
ttc+num=0-0-1-0+time=b-a-a-a+014		File Folder
ttc+num=0-0-1-0+time=c-a-a-a+015		File Folder
ttc+num=0-0-1-0+time=d-a-a-a+016		File Folder
ttc+num=0-0-1-1+time=a-a-a-a+017		File Folder

Figure 14: Screenshot showing top level experiment variation

5.7 Parallelism, Multiple cores and using the 'multiple' option to DXM

Each experiment constructed and managed via DXM is entirely *independent* of any other experiment in the set – each experiment is standalone. This means that they all could be calculated freely in any order, since each experiment cannot depend upon the presence, absence or failure of any other experiments.

The pleasing consequence of this is that, in principle, the performance of experiments could immediately exploit whatever independent computing resources there are available - e.g. multiple processor cores in data centers.

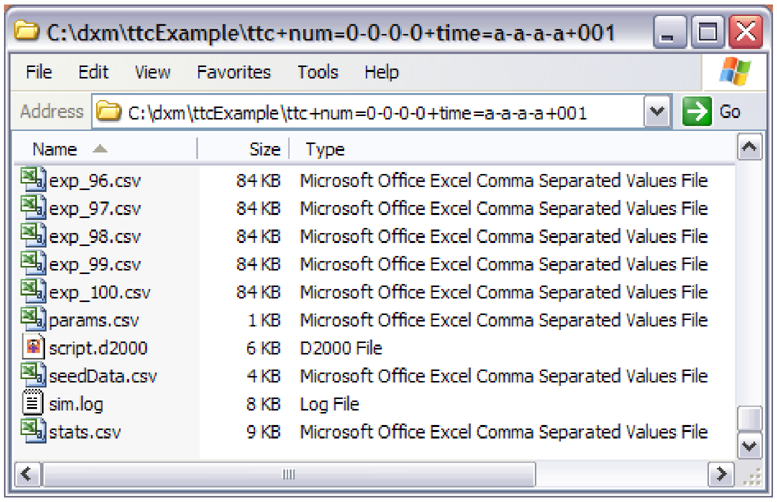
But there is a structural management and configuration issue here - to do this in an effective way, we need to split up the experiments work into independent (i.e. disjoint) pieces. This means that instead of a single top-level root directory, there will now be several - one for each independent processor. Moreover, we need to do this in a clean way so that it is possible to *assemble* the separate experiments sets again so as to form a single, unified experiment set.

In other words, the work first needs to be split up, performed separately and then put back together again as though the entire set of experiments had been done by a single processor.

DXM helps to manage this by providing the 'multiple' option to split the experiments up into a specified number of disjoint experiment sets at creation time. Once created, each of the experiments sets can then be performed by DXM in the standard way. Once all the experiments sets have been completed, the total set of experiments can be assembled from these pieces.

So for example, suppose we have 4 processor cores available. We can generate the experiment set as 4 disjoint subsets by typing the following at the command line:

```
    $ dxm -create ttc1.dxm -multiple 4
```


C:\dxm\ttcExample\ttc+num=0-0-0-0+time=a-a-a-a+001
File
Edit
View
Favorites
Tools
Help
Address
C:\dxm\ttcExample\ttc+num=0-0-0-0+time=a-a-a-a+001
Go

Name	Size	Type
exp_96.csv	84 KB	Microsoft Office Excel Comma Separated Values File
exp_97.csv	84 KB	Microsoft Office Excel Comma Separated Values File
exp_98.csv	84 KB	Microsoft Office Excel Comma Separated Values File
exp_99.csv	84 KB	Microsoft Office Excel Comma Separated Values File
exp_100.csv	84 KB	Microsoft Office Excel Comma Separated Values File
params.csv	1 KB	Microsoft Office Excel Comma Separated Values File
script.d2000	6 KB	D2000 File
seedData.csv	4 KB	Microsoft Office Excel Comma Separated Values File
sim.log	8 KB	Log File
stats.csv	9 KB	Microsoft Office Excel Comma Separated Values File

Figure 15: Screenshot showing the results content for a completed experiment

This creates a 'master' root directory that in turn contains all four subsets. Each of these subsets have their own root directories from which a run of DXM can be launched independently of the other directories. In principle, each subset can be copied elsewhere for the purposes of getting back results and then copied back for assembly into the single set of results.

Once all of the results have been completed, we can assemble them together into a single experiments set by typing at the command line:

```
$ dxm -assemble ttc1.dxm
```

For this to work successfully, it is very important that the above command is invoked from the same place originally used to invoke the DXM multiple create command. Additionally, the `-assemble` option is only intended to work with experiment sets that were initially split apart when they were created (i.e. using the `-multiple` option); it is clearly not a general assembly operation for arbitrary experiment sets.

6 Viewing experiment results graphically using DXV

This brief section cannot do full justice to the Demos2k eXperiments Viewer (DXV) tool here. We shall therefore be content to simply illustrate it's use by showing some of the charts that the tool may generate on behalf of the analyst/modeller. Brief outline documentation for DXV in included in Appendix B.

6.1 Using DXV to explore results

The original question was to find conditions that maximised the value of `totalSales`. This variable is clearly cumulative and so only its final value at the end of each run is relevant. Our strategy, then, is to locate combinations of parameter values for which the average value of the `totalSales` parameter is (close to) maximised. As each experiment is associated with a specific such parameter combination, all we need to do is locate those experiments at which `totalSales` is maximised.

6.1.1 Charting `totalSales` from the summary stats - first pass

We first of all chart the value of `totalSales` across all of the experiments[2] - it turns out that this is fairly easy to do by using the DXV view plan given in Figure 16. The chart produced by DXV is given in Figure 17.

By inspection of this chart, we can see that the maxima in `totalSales` is attained within a cluster of experiments. Unfortunately, due to the scale of the chart, we cannot see more precisely what the values are and where they lie. For that we need a more refined chart.

```
#   ttc1.dxv -- a DXV file for the threeTier+Customers example
#
    plan = ttc1.dxm

    size (900, 600)

    font.title = (plain, bold-italic, 12)

    line basic = color : dark red, point = plus, pointsize=0.8

    chart summary

      title = "totalSales"

      title.x-axis : 'Experiments'
      title.y-axis: 'Units'

      plot style : points

      data = summary

      y = totalSales     with line = basic
```

Figure 16: View plan charting totalSales across all experiments

[2]Technically, we can linearly order the experiments lexicographically by name.

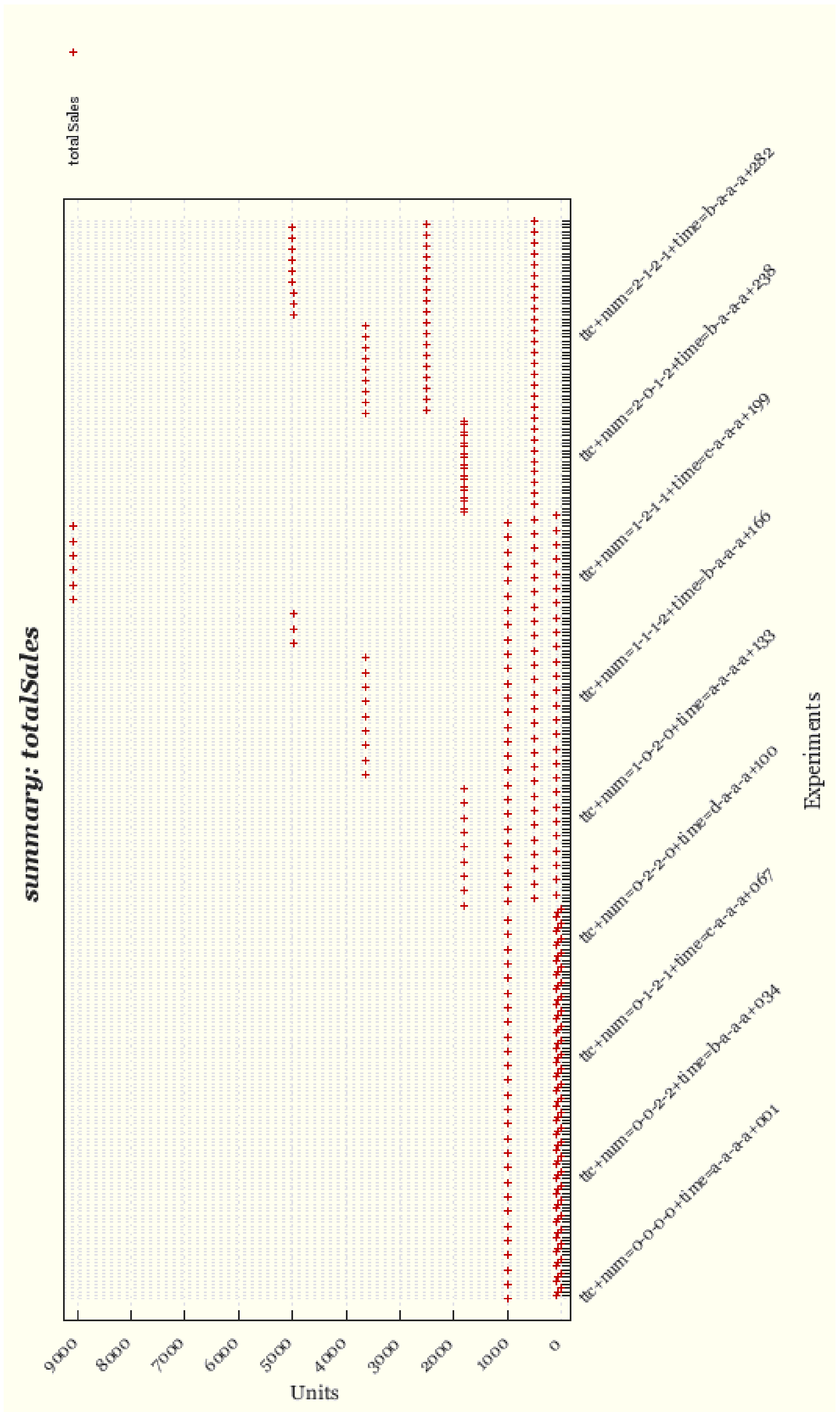


Figure 17: Chart of `totalSales` across all experiments

6.1.2 Charting `totalSales` from the summary stats - refined

Fortunately, it is possible to refine our chart to get a sharper view of which experiments maximise the value `totalSales`. Again, this turns out to be fairly straight forward to do with DXV. We can focus in on a sub-range by adding a single 'sample' statement which restricts which experiments are to be plotted. The refined view plan is given in Figure 18 and the corresponding refined chart is presented in Figure 19

From the refined charts, we can see that the maxima in `totalSales` seems to lie just above 9000 units in six different, but clustered, experiments To get an exact picture, we eventually need to look directly at the spreadsheet data – but even so, we have used the charts to effectively glean some heuristic information to know what to look for and also where to look for it.

```
#   ttc1a.dxv -- a DXV file for the threeTier+Customers example
#
#   Refined range using the sample statement
#
    plan = ttc1.dxm

    size (900, 600)

    font.title = (plain, bold-italic, 12)

    line basic = color : dark red, point = plus, pointsize=0.8

    sample = ( expt ttc+num=1-1-1-2+time=b-a-a-a+166 <= expt ttc+num=2-0-1-2+time=b-a-a-a+238 )

    chart summary

      title = "totalSales"

      title.x-axis : 'Experiments'
      title.y-axis: 'Units'

      plot style : points

      data = summary

      y = totalSales     with line = basic
```

Figure 18: View plan that charts `totalSales` across a range of experiments

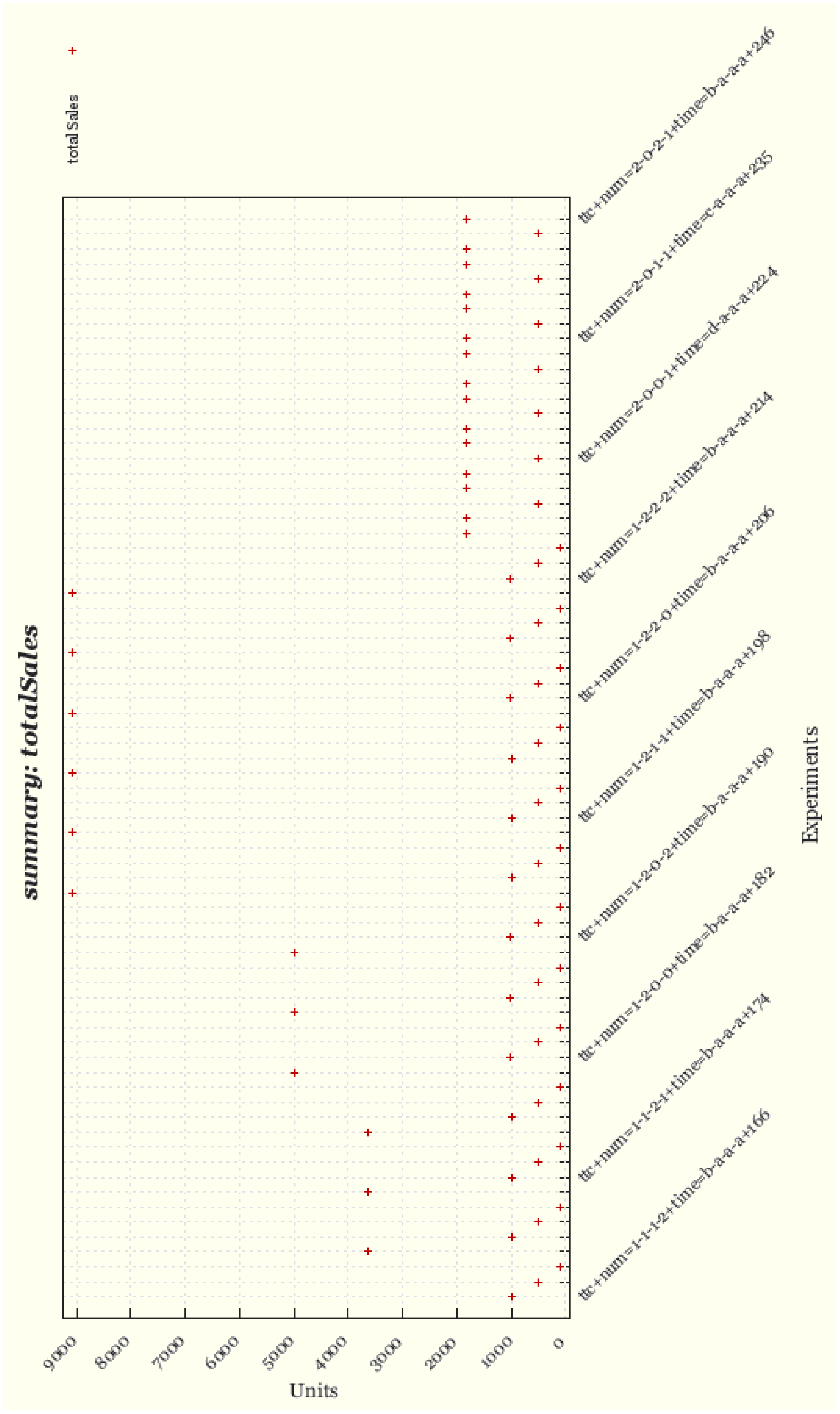


Figure 19: Refined chart of `totalSales` for a range of experiments

7 Answering our question ...

Having now gone to all the effort of running the experiments and preparing charts, we should briefly discuss what the actual result or outcome was!

From the preceding section, we determined that the `totalSales` have maxima exceeding 9000, in six separate experiments. Turning now to the `summaryData.csv`, we extract the data on the cluster of experiments containing the maxima, and the data obtained is summarised in Table 1. The column names appearing in the spreadsheets are essentially the names of the associated Demos2k model parameters. We include in the table only those model parameters that changed within the cluster of experiments[3].

The remaining model parameters were all constant over the above range of experiments: `numOfCustomers` = 10, `sourceAvgSupplyTime` = 1, `numOfSourceProc` = 10, `resellerAvgProcessTime` = 1, `sourceResetAvg` = 0.1, `runTime` = 1000.

From the table, we can see a number of interesting features of the data:

- The `totalSales` output data is very regular - this may be due to both the nature of the model itself and the fact that the numbers given here are averages over a 100 runs, with each run having duration 1000 in simulation time.
- We can see an immediate inverse correlation between `customerAvgRequestInterval` and `totalSales`. This is easily accounted for by the fact that `customerAvgRequestInterval` is the (average) time interval *between* customer requests being made. As this value goes down, the average number of customer requests will go up, leading to increased demand and increased total sales (assuming the service chain can deliver).
- There are two closely matched maxima in `totalSales` - one of 9069.55 and the other of 9067.12. By inspection, these differences are inversely correlated with the number of resellers: viz. `numOfResellerProc`. However, it isn't immediately clear why increasing the number of reseller processes from 10 to 20 should lead to `totalSales` decreasing slightly overall. This is a good example of some unanticipated data result deriving from the dynamics in a model that may lead to further questions for the investigation to consider and insights to be obtained.

[3]All the other model parameters were constant: (`numOfCustomers` = 10; `sourceAvgSupplyTime` = 1; `numOfSourceProc` = 10; `resellerAvgProcessTime` = 1; `sourceResetAvg` = 0.1; `runTime` = 1000.

Expt. folder	Model parameters:			Result:
	numOfResellerProc	numOfRetailProc	customerAvgRequestInterval	totalSales
ttc+num=1-2-1-0+time=a-a-a-a+193	10	2	1	9069.55
ttc+num=1-2-1-0+time=b-a-a-a+194	10	2	10	999.5
ttc+num=1-2-1-0+time=c-a-a-a+195	10	2	20	501.94
ttc+num=1-2-1-0+time=d-a-a-a+196	10	2	100	101.26
ttc+num=1-2-1-1+time=a-a-a-a+197	10	10	1	9069.55
ttc+num=1-2-1-1+time=b-a-a-a+198	10	10	10	999.5
ttc+num=1-2-1-1+time=c-a-a-a+199	10	10	20	501.94
ttc+num=1-2-1-1+time=d-a-a-a+200	10	10	100	101.26
ttc+num=1-2-1-2+time=a-a-a-a+201	10	20	1	9069.55
ttc+num=1-2-1-2+time=b-a-a-a+202	10	20	10	999.5
ttc+num=1-2-1-2+time=c-a-a-a+203	10	20	20	501.94
ttc+num=1-2-1-2+time=d-a-a-a+204	10	20	100	101.26
ttc+num=1-2-2-0+time=a-a-a-a+205	20	2	1	9067.12
ttc+num=1-2-2-0+time=b-a-a-a+206	20	2	10	1000.72
ttc+num=1-2-2-0+time=c-a-a-a+207	20	2	20	502.21
ttc+num=1-2-2-0+time=d-a-a-a+208	20	2	100	99.91
ttc+num=1-2-2-1+time=a-a-a-a+209	20	10	1	9067.12
ttc+num=1-2-2-1+time=b-a-a-a+210	20	10	10	1000.72
ttc+num=1-2-2-1+time=c-a-a-a+211	20	10	20	502.21
ttc+num=1-2-2-1+time=d-a-a-a+212	20	10	100	99.91
ttc+num=1-2-2-2+time=a-a-a-a+213	20	20	1	9067.12
ttc+num=1-2-2-2+time=b-a-a-a+214	20	20	10	1000.72
ttc+num=1-2-2-2+time=c-a-a-a+215	20	20	20	502.21
ttc+num=1-2-2-2+time=d-a-a-a+216	20	20	100	99.91

Table 1: Results table

7.1 The Dark Side of Simulation : Failures and partial behaviour

Up till now, you may have been prepared to believe that the example we chose to simulate was entirely well-behaved dynamically and that the experiment runs always terminated. If so, then you are sadly mistaken. Some of the experiments failed during their runs.

Such failures during runs are typically not about programming errors - these are typically uncovered *prior* to running DXM anyway. Some of the failures may arise because the simulated dynamics in the model forces the Demos2k tool to exceed its internal limits - typically the *process spawning limit* and the *livelock limit* - and we discuss these limits in greater detail in Appendix A.6. It turns out that we can increase these limits within DXM simulations by use of structured pragma comments embedded within the Demos2k model - see Appendix A.6.

However, the remaining failures could represent unavoidable extreme dynamics (i.e. singularities) - these are in fact of extreme interest and capturing any data about them is highly desirable. Another, perhaps less exotic explanation is that the model entered an inappropriate region of operation without check - which implies that the model had been misapplied or was somehow unrealistic and misleading. In such a case, the failures would represent the breakdown of applicability of the model itself i.e. "crossing the line" or "going over the edge".

Clearly all of these cases may arise in practice - to know which of these is relevant to any given situation requires considerable skill, a certain amount of modelling/analytical experience - and a lot of evidence/data!

In our case, there were 27 failed experiments - and all of them were caused by the `customer_request` bin growing too large. It is certainly possible to rerun all these particular experiments again with an increased set of limits - but it may also be prudent to look more closely at what the simulation is doing and perhaps trace more of the behaviour to help the analyst formulate hypotheses about what is causing these failures. In any event, the DXM tools can help analyst/modellers to investigate even these partial-behaviour phenomena.

7.2 Running the experiments for our example

As noted earlier in § 5.4, there are 324 experiments in total, each with 100 runs, making a total of 32400 runs. We exploited the `-multiple` option to fully utilise the dual-core workstation we had available (HP xw8400 Workstation Intel Xeon CPU 5160 (dual core), clocked at 1.97 GHz, with 3.5GB RAM). The experiments were split into two equal halves and then worked on independently.

The first half was a lighter computational load than the second half. The first half took 13hrs 11mins to complete its set of 162 experiments, whereas the second half took much longer with 23hrs 24mins – around twice the effort. The reason for the difference seems to be that the second half involved experiments setup to use significantly more concurrent processes than the first half.

The total storage used by the experiments was 2.18GB (2,350,980,034 bytes).

8 Further developments and extensions

The DXM software has arisen out of a practical need to design, manage and implement large scale experiments using the Demos2k process modelling tool. It has so far been highly successful in enabling analyst/modellers within HP to conduct investigations that would otherwise have been out of the question on grounds of infeasibility..

Software is never completely finished - especially an application of this complexity. However, there are some natural further developments that extend the range of what can be achieved with DXM:

- GUI front-end to make the business of taking a Demos2k model, building an experiments plan and then executing it smoother and less involved. Such a GUI should also help with the clerical management of results sets and their drilldown.

- Better integration with DXV to give improved visualisation of results and linkage back to the Demos2k model.
- Connection to external ‘best of breed’ data analysis tools, including commercial products such as: MathWorks and Mathematica.
- Provide a fully-fledged interface between DXM and SQL databases – this already exists in a simple, rudimentary form but would need to be extended for it to be of much benefit to DXM users.

8.1 Status of the DXM application code

The DXM and DXV applications are currently robust (but experimental) prototypes that are currently able to be installed on Windows platforms, internally within HP and our research partners. The tools are entirely implemented using a number of non-proprietary open-source software technologies that users would have to independently acquire/download and then install for themselves (e.g. Python 2.5.1, Perl, Demos2k and Gnuplot).

An external offering of this software would involve getting the tools into a more usable and lower maintenance state. Further developments such as providing GUI support for DXM and DXV, and ensuring that the tool can widely interoperate with external data analysis toolkits (e.g. MathWorks, Mathematica, etc.) would need to be stongly considered before that state is reached.

However, we are considering applying to release the software in some format, and so it could potentially be made available as open-source in the future.

9 Acknowledgements

Mike Yearworth pointed me towards the need for an application like DXM when conducting any kind of extensive ’Monte-Carlo style’ investigation using Demos2k. Jonathan Griffin was especially helpful and encouraged further development of the application to make it more useful for larger scale applied modelling. Marco Casassa Mont and Siva Raj Rajagopalan both made helpful remarks and took the trouble to use the system and to report bugs. The overall systems approach illustrated here owes much to the HP Open Analytics project espoused by Richard Taylor, Chris Tofts and Mike Yearworth. Thanks are also due to Chris Tofts & Graham Birtwistle who developed the Demos2k language and its toolkit in the first place.

A Demos2k eXperiments Manager – the DXM tool

A.1 Objectives for DXM

- Takes input from an experiments plan document and from that computes a collection of directories containing instantiated Demos2k scripts etc., one for each experiment variation.
- Performs a sequence of Demos2k runs for each of these experiment variations and calculates elementary statistics for that set.
- Support interoperability by providing SQL import of experiment results.
- Provide support for multiple experiment runs operating independently and concurrently (e.g. data center operations).

An outline diagram illustrating what DXM needs as input and produces as outputs is presented in Figure 20.

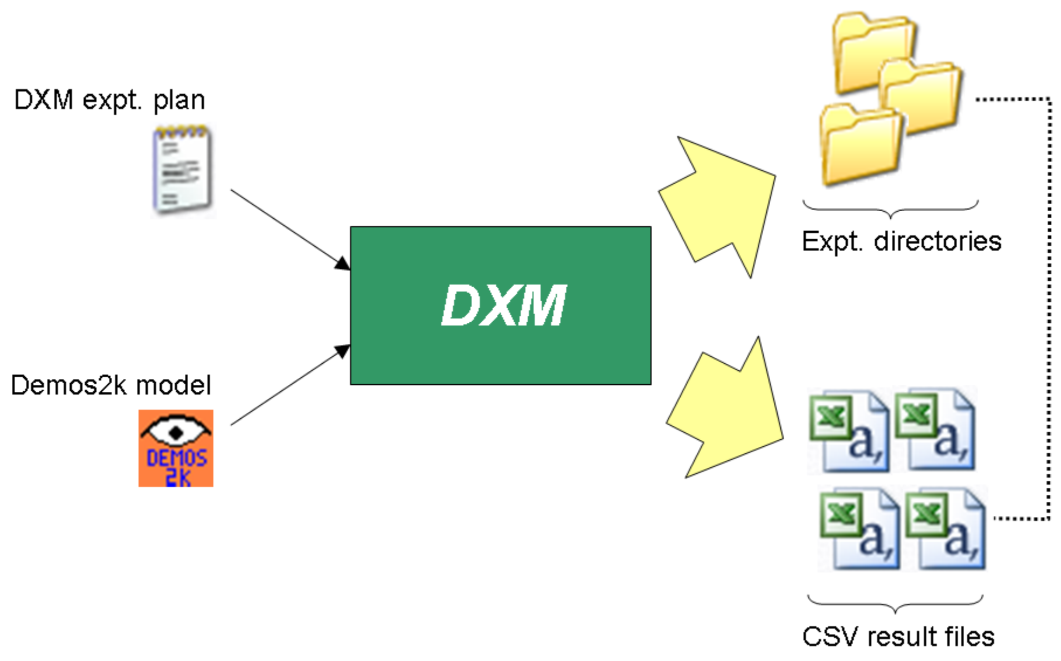


Figure 20: Block diagram describing input and outputs for DXM

A.2 Installation (Windows-only)

The following need to have been installed:

- Systems:

`Perl`	Use `Perl 5` to run the installer.
`Python`	See `http://www.python.org/download/` for latest installation.
`Standard ML`	As per standard 'SeymourSys' installation.

- Other Demos2k-related tools:

`demosCmd`	Stand-alone Demos2k engine (v1.06 or greater)

Install instructions:

- To install DXM:
  1. . Unpack the `ZIP` file
  2. . `CD` into the unzipped directory.
  3. . Type: `perl install.pl`
     – This installs DXM by creating a 'dxm' directory in your Seymour installation directory and copying content to it.
  4. Finally, you will need to copy the dxm.bat into *your* shell directory on your PATH.
     (As a convenience, `dxm.bat` will be copied into your Seymour bin directory, which might be on your PATH.)
- To uninstall DXM:
  1. `CD` into the unzipped directory.
  2. Type: `perl uninstall.pl`

A.3 Experimental Plan

The purpose of the experiments plan is to define a finite set of experiment variations based upon ranges of specific Demos2k parameters. The plan also needs to specify other details such as whereabouts the data is stored, the naming convention for the subdirectories and the basic Demos2k script that is used as the basis for each experiment variation.

The experiments plan is thus a text document (typical extension .dxm) that specifies:

1. The root Demos2k script providing the basis for each experiment variation.
2. The root directory/folder containing all of the experiment variations.
3. Specific list of Demos2k parameters and the ranges of values they take within each experiment variation. These variations are formed by taking a specific choice of each of the specified Demos2k parameters.
4. The naming convention for each experiment variation. Each experiment variation will contain an instantiated Demos2k script used to generate trace data when executed. It also contains the generated traces and the derived spreadsheets. Thus, each of these directories must have a unique name - and the naming convention determines precisely what this naming is. The idea is that the naming should be structured so as help to find specific variations etc., rather than simply be a pure number.

Here is an example of an experiments plan:

```
file = patching-v16-310807.d2000
rootdir = expt-2ndSept2007

runs = 45

param patchAssessmentStaff : from 2 to 5 step 1
param vulnRate : 1/100; 5/100; 15/100; 35/100
param volitility : uniform(0.00, 0.01); uniform(0.01, 0.02); uniform(0.02, 0.03)

naming = x-%A-%N-%A
```

This plan specifies that:

- The Demos2k model file is named 'patching-v16-310807.d2000'
- The root directory is named 'expt-2ndSept2007'
- There will be 45 runs performed/attempted - for each experiment variation.
- There are three Demos2k parameters whose values are varied - note that we can include (literal) Demos2k expressions here:
  - The parameter 'patchAssessmentStaff' takes the 4 values: 2, 3, 4, 5
  - The parameter 'vulnRate' takes the 4 values: 1/100, 5/100, 15/100, 35/100
  - The parameter 'volitility' takes the 3 expressions:
    'uniform(0.00, 0.01)', 'uniform(0.01, 0.02)', 'uniform(0.02, 0.03)'
- The directories are named according to the scheme: x-%A-%N-%A
  - An example directory name would then be 'x-B-1-A' which would correspond to having:
    param patchAssessmentStaff = 3;
    param vulnRate = 1/100;
    param volitility = uniform(0.00, 0.01);

    Note that we cannot really use the actual values of these parameters to name the directory since they could be specified by lexically complex expressions like '1/500' or 'negexp(25)'.

    This directory naming notation is explained further below.

The above example specifies a set of experiment variations, one for each choice of value for the parameters `patchAssessmentStaff`, `vulnRate`, and `volitility`.

Therefore, there are $48 = (4 \times 4 \times 3)$ particular variations.

A.3.1 Constrained parameters

It is sometimes useful and indeed necessary to constrain the parameters of an experiment so that they avoid unnecessary, redundent or otherwise unwanted experiment variations.

We may do this by specifying some constraints on the parameters and then using these when generating the experiments framework to give the acceptable combinations.

Each constraint is essentially an arithmetical inequality involving the parameters e.g.

$$(a + b) > (c + d) * e$$

There may be several such constraints given - and all must be satisfied for an acceptable set of parameter values to result.

If no constraints were specified then this is interpreted as no constraint at all.

Example: Suppose the DXM plan contained:

```
...
param p1 : from 0 to 1.0 step 0.2
param p2 : from 0 to 1.0 step 0.2
param p3 : from 0 to 1.0 step 0.2

constraint: p1 + p2 + p3 = 1
```

The above constraint example results in solutions for `(p1, p2, p3)` such as:

```
...
(1, 0, 0)
(0.4, 0.2, 0.4)
(0.2, 0.2, 0.6)
(0.8, 0, 0.2)
...
```

and so on.

Note: To include solutions like:

(0.33, 0.33, 0.33)

or

(0.5, 0, 0.5)

we could increase the granularity of the scan over the parameter space:

```
param p1 : from 0 to 1 step 0.01
param p2 : from 0 to 1 step 0.01
param p3 : from 0 to 1 step 0.01
```

and perhaps relax the constraints slightly to allow for approximation:

```
constraint: p1 + p2 + p3 =< 1
constraint: p1 + p2 + p3 >= (1 - 0.01)
```

As can be seen, a constraint condition is essentially an arithmetic inequality. Multiple constraint conditions are permitted and are implicitly conjoined together (i.e. intersection).

A.4 Syntax for experiments plans

More formally, here is a BNF grammar for the syntax of an experiment plan document.

As usual, * means 0 or more repetition, + means 1 or more repetition,   means optional and standard brackets are used for grouping. Literals are enclosed in single quotes.

```
plan            ::=  defn+

defn            ::=  fileSpec | rootdirSpec | runsSpec | paramSpec |
                     seedSpec | namingSpec  | constraintSpec

fileSpec        ::=  'file' '=' str

rootdirSpec     ::=  'rootdir' '=' str

runsSpec        ::=  'runs' '=' num

paramSpec       ::=  'param' id ':' enumSpec

enumSpec        ::=  valSpec ( ';' valSpec )*  ';'

valSpec         ::=  demosExpr | rangeSpec

rangeSpec       ::=  'from' num 'to' num  stepSpec

stepSpec        ::=  'step' num | 'by' num

seedSpec        ::=  seedFileSpec | seedGenSpec

seedFileSpec    ::=  'seedfile'  '=' str

seedGenSpec     ::=  'seedgen'  seedGenOpts

seedGenOpts     ::=   ':' seedOptSpec ( ';' seedOptSpec )*  ';'

seedOptSpec     ::=  'init' num  |  'step'  num  |  'size' num

namingSpec      ::=  'naming' '=' (char+ formatSpec)+ char*

formatSpec      ::=  '%%' | '%a' | '%A' | '%n' | '%N' | '%z' | '%Z'

constraintSpec  ::=  'constraint' ':' constraintExpr

Lexicals:
  char            ==   visible character that can be used for a directory names
  str             ==   strings
  id              ==   Demos2k identifiers
  num             ==   numeric values
  demosExpr       ==   simple Demos2k expressions (typically, with no variables)
  constraintExpr  ==   simple (one-line) arithmetical inequality involving parameter id's.
```

A.5 Naming and Format specifiers

This section describes how unique directory names are constructed for each experiment, based upon `namingSpec`'s.

The idea is that the `namingSpec`'s are used to construct a unique name from the values of the experiments parameters in the experiment. The section following contains an illustration.

Each parameter spec. defines a finite range of values, placed in an ordered sequence. These values can be 'indexed' to give a particular ranking. We illustrate this for three parameters `P`, `Q`, `R` as follows. Suppose the parameters `P`, `Q`, and `R` have the following ranges:

P	:	$x_1; x_2; ...; x_D$
Q	:	$y_1; y_2; ...; y_E$
R	:	$z_1; z_2; ...; z_F$

Thus, there are $(D \times E \times F)$ experiments = N. Accordingly, each experiment variation could be uniquely numbered from 1 .. N.

The first experiment has values for (P, Q, R) = (x_1, y_1, z_1), whereas the final experiment has values for (P, Q, R) = (x_D, y_E, z_F).

The experiments are enumerated in standard lexicographic order - i.e. last range fastest.

Suppose the naming specifier was the string 'x-%A-%N-%A'. Then we would have the following association of tuples with directory names:

Tuple (P, Q, R)	**Directory name**
(x_1, y_1, z_1)	x-A-1-A
(x_1, y_1, z_2)	x-A-1-B
...	...
(x_1, y_2, z_3)	x-A-2-C
...	...
(x_2, y_3, z_4)	x-B-3-D
...	...

In more detail, the format specifiers (i.e. `formatSpec` above) mean the following:

%%	A literal % character
%a	Use lower-case alphabetic enumeration: i.e. a, b, ..., z, aa, ab, ...
%A	Use upper-case alphabetic enumeration: i.e. A, B, ..., Z, AA, AB, ...
%n	Use numerical enumeration, starting from 0: i.e. 0, 1, 2, 3, ...
%N	Use numerical enumeration, starting from 1: i.e. 1, 2, 3, 4, ...
%z	Use absolute numbering (i.e. number each experiment variation uniquely, starting from 0). This is formatted fixed width, padded by zeroes.
%Z	Use absolute numbering (i.e. number each experiment variation uniquely, starting from 1). This is formatted fixed width, padded by zeroes.

Notes:

1. The requirement is that each namingSpec must provide a means to uniquely name each directory.
2. The namingSpec's clearly depend upon the ordering that the parameters are specified in the plan. In general, each formatSpec contributes a nominal signifier into the overall directory name, based upon the corresponding parameters's value.
3. The absolute numbering specifiers %z and %Z can be used on their own, since they uniquely specify each experiment variation.

A.5.1 Illustration

We now illustrate these format specifiers based upon our earlier example. Assume that the experiments parameters are bound as follows:

```
param patchAssessmentStaff = 3;
param vulnRate = 1/100;
param volitility = uniform(0.00, 0.01);
```

Here are various examples of naming specs and the corresponding directory name:

Naming spec	Directory name
x-%A-%N-%A	x-B-1-A
x-%a-%n-%A	x-b-0-A
x-%z-%a-%n-%a	x-04-b-0-a
x-%a-%n-%a-%Z	x-b-0-a-05
x-%Z	x-05
x-%a-%Z-%a	x-a-05-a

A.6 DXM Pragmas for Demos2k

DXM supports *Pragma Comments* (i.e. *structured comments*) for Demos2k as described below. This provides a handy way to explicitly record important experiment settings information within Demos2k source files.

Each pragma comment takes the form:

```
//* <name> = <positive-number>
```

where `<name>` is one of `INIT-RUN-SEED`, `LIVELOCK-STEPS` or `SPAWN-LIMIT`. The equals sign can also be ":" and spaces/tabs are arbitrary. The names specify these settings:

`INIT-RUN-SEED`	RNG seed for the *first* simulation run only.
`LIVELOCK-STEPS`	Number of steps for livelock detection.
`SPAWN-LIMIT`	Max. number of processes/bins.

IMPORTANT NOTE: pragma comments are *only* available in DXM – the GUI already has the means to change most of these limits directly (see menu File → Settings). In particular, the GUI will *ignore* all pragma comments.

A.7 Supporting multiple subordinate experiment runs using DXM

Large experiments, typically involving several ranges, will need the experiments space to be shared out for processing amongst a set of machines. These are 'subordinate' experiment sets which can be performed independently.

DXM provides support for subordinate experiments in two ways:

1. Splitting a given experiment setup into a set of subordinate folders, each containing a specific collection of experiments, disjoint from the others.

   DXM can then be operated in each of these subordinate folders independently.

2. Assembling results from the subordinate folders into a consistent set of outcomes for the entire set of experiments performed.

A.7.1 Organisation of the multiple experiment sets

How are the multiple experiment sets organised? The directory structure involved turns out to be a little intricate and it is probably worth explaining here what the various issues are.

Lets consider the standard situation of a single experiment set - this is diagrammed in Figure 21.

The *user directory* is the place at which DXM is invoked to do the initial creation of the experiments root directory and its subsidiary experiment directories.

Now, invoking the DXM `-multiple` option creates a more complex structure that starts from a'master' root directory in which the appropriate experiment subsets are in turn created. This more complex situation is diagrammed in Figure 22.

As we can see, the master root directory contains a number of *subsidiary user directories*, one for each subset. These in turn contain experiment directories as in the single experiment case. The reason that these subsidiary user directories are needed is that, once split up, DXM still needs somewhere to launch experiment runs independently for each particular subset.

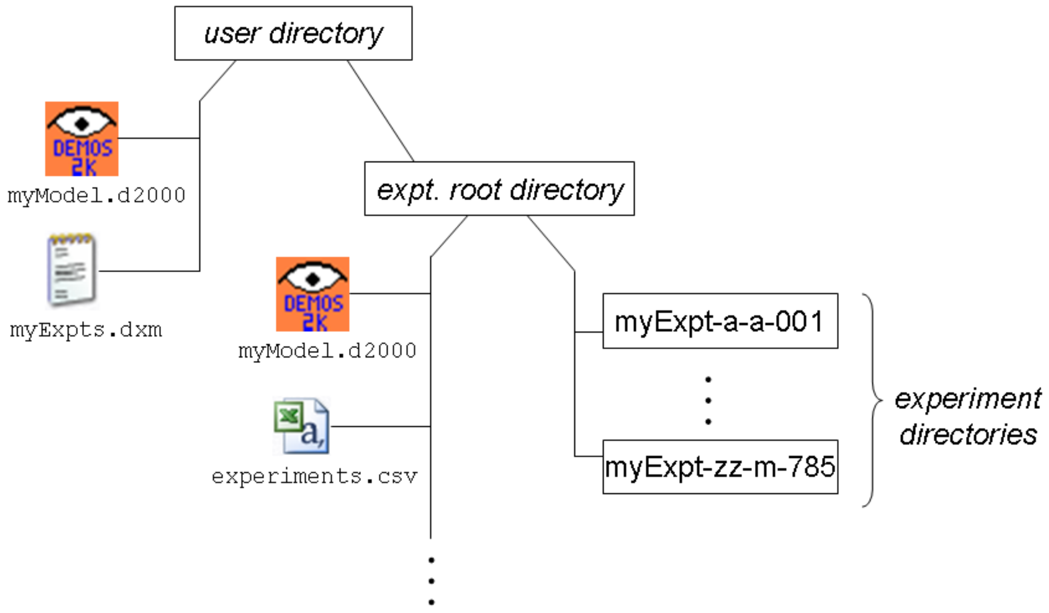


Figure 21: Diagram of the directory structure for a single experiment set

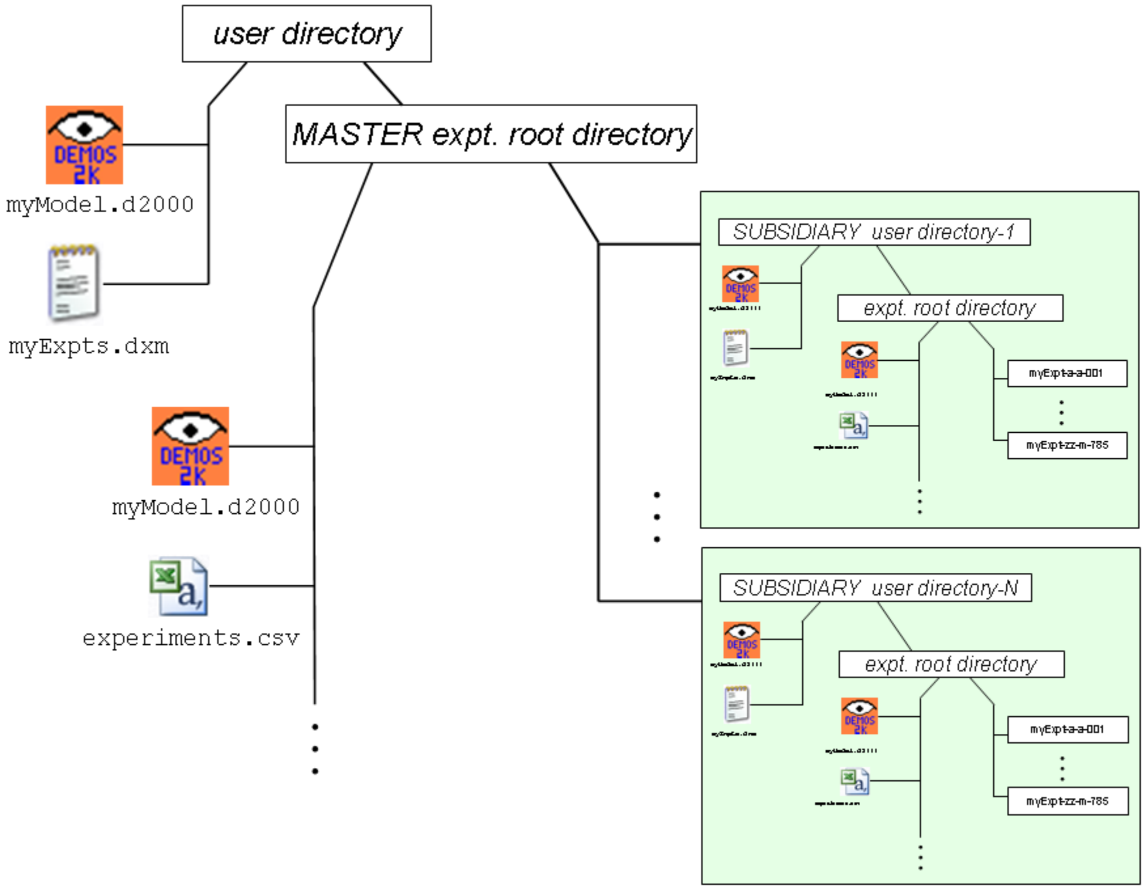


Figure 22: Diagram of the directory structure for multiple experiment subsets

A.8 Usage information for the DXM application

```
dxm - Demos eXperiments Manager     Version: v1.2b, Date: 13th October 2008

Usage: dxm <actions> <planfile>   -- Read planfile and perform actions
       dxm <planfile>             -- Read planfile and report spec. to user
       dxm -h                     -- This usage message

where:

   <planfile> : Experiment plan file (typical file extension: .dxm)

                Once the experiment directory has been created, the current plan
                file is saved and can be omitted subsequently.

   <actions>  : Actions to be performed (can be abbreviated):

   -assemble        -- Assemble subordinate results into a single set of results for
                       the entire set of experiments.  It is not necessary for all
                       subordinate experiments to have completed - a partial snapshot
                       of results will be calculated.

   -create          -- Create experiment structure (error if it already exists).  This
                       must be performed successfully prior to running any experiments.

   -keep-traces     -- Keep all trace files - do not delete them after the stats have
                       been computed.

   -list            -- List the current experiment log (i.e. expt.log) to view how far
                       the experiment runs have got so far.

   -multiple <num>  -- Split all the experiments amongst <num> subordinate folders.

   -run             -- Spawn a process to run any uncompleted experiments remaining -
                       partially completed runs will be recalculated from the start.

                       Note: the number of simulations in each experiment (i.e. the
                       number of runs) is specified in the plan using the 'runs' spec.

   -process         -- Recalculate statistics.

   -stop            -- Stop/halt current experiment runs.  These can be restarted by
                       running again (any incomplete runs are deleted and begun afresh).

   -wipe            -- Erase ALL experiments performed so far - essentially a global
                       reset of the current configuration.  CAUTION - THIS IS IRREVERSIBLE.

   -help            -- This message.
```

B Demos2k eXperiments Viewer – the DXV tool

The Demos2k eXperiments Viewer tool provides an elegant and effective way of generating large numbers of particular charts, once we have obtained the experiment results. The charts produced by DXV are specified by the analyst/modeller using a 'view plan' that is similar in form and concept to the 'experiment plan' we used with DXM. Altough the syntax of DXV view plans is defined below in Appendix B.1, explaining what this means in detail is left for another occasion. An outline diagram describing what information DXV uses and produces is given in Figure 23.

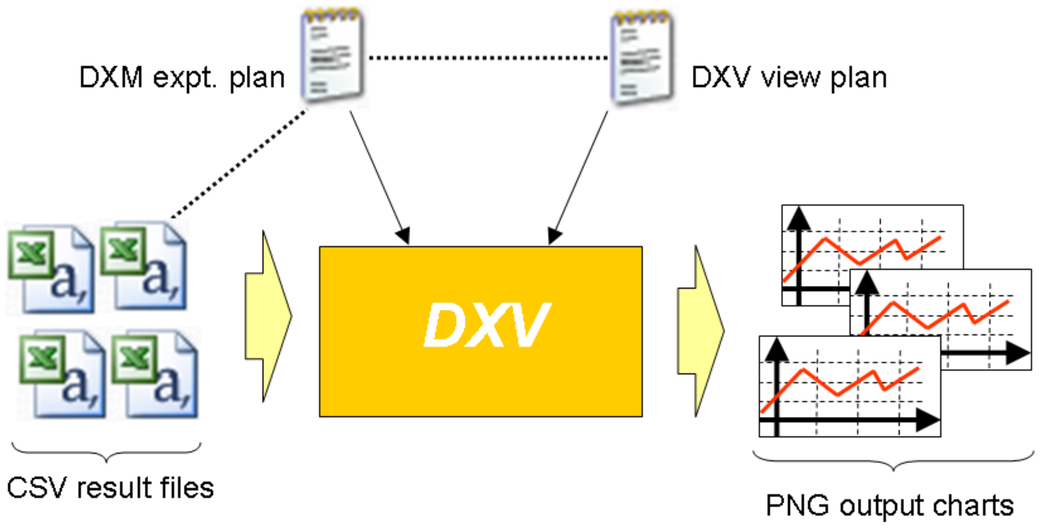


Figure 23: Block diagram describing input dependencies and outputs for DXV

B.1 Syntax for DXV view plan files

Here is a BNF grammar for the syntax of view plan documents.

As usual, * means 0 or more repetition, + means 1 or more repetition, means optional and standard brackets are used for grouping. Literals are enclosed in single quotes.

```
    viewPlan        ::=  paramsDefn+ chartSpec+


Parameters:
-----------
    paramsDefn      ::=  planSpec | fontSpec   | funSpec   | sizeSpec
                      |  plotSpec | sampleSpec | lineSpec

    planSpec        ::=  'plan' eq pathString

    funSpec         ::=  'fun' funName '(' paramList ')' '=' arithExpr

    paramList       ::=   id ',' id*

    sizeSpec        ::=  'size' eq '(' width ',' height ')'

    pathString      ::=   string

    funName         ::=   id

    width           ::=   num

    height          ::=   num


Charts:
-------
    chartSpec       ::=  'chart' chartName chartDefn+

    chartDefn       ::=  titleSpec  | xAxisSpec | yAxisSpec
                      |  plotSpec   | fontSpec  | funSpec
                      |  groupSpec  | dataSpec  | sizeSpec
                      |  sampleSpec | lineSpec  | histogramSpec

    titleSpec       ::=  'title' titleType eq str

    titleType       ::=  '.x-axis' | '.y-axis' | '.chart'

    xAxisSpec       ::=  'x' '=' colSpec

    colSpec         ::=  demosTime  |  colName

    demosTime       ::=  'demos_time' | 'demostime' | 'demos-time'

    groupSpec       ::=  'grouped' 'by' groupItem

    groupItem       ::=  'sum' | 'average' | 'maximum' | 'minimum' | 'envelope'

    yAxisSpec       ::=  yExprSpec | yFunctionSpec

    yExprSpec       ::=  'y' '=' curveExpr  'with' curveSpec

    yFunctionSpec   ::=  'y' '(' 'x' ')' '=' curveExpr  'with' curveSpec

    curveSpec       ::=   curveSpecItem  ',' curveSpecItem

    curveSpecItem   ::=   curveName  |  curveLine

    curveName       ::=  'name' eq curveLabel
    curveLine       ::=  'line' eq lineName

    curveLabel      ::=  string

    chartName       ::=  id

    colName         ::=  id


Data sources:
-------------
    dataSpec        ::=  'data' 'source' eq dataSource

    dataSource      ::=  'summary' | statsChart | runChart

    statsChart      ::=  exptSource '/' 'stats'

    runChart        ::=  exptSource '/' 'run' runSpec

    exptSource      ::=  'all' | exptDir

    exptDir         ::=  'expt' id

    runSpec         ::=  'all' | runGroup | runRange | runInt

    runGroup        ::=  'group' groupDesc

    groupDesc       ::=  'all' | runRange

    runRange        ::=  runInt '-' runInt

    runInt          ::=  int


Plot styles:
------------
    plotSpec        ::=  'plot' 'style' eq styleType

    styleType       ::=  'steps' | 'lines' | 'points' | 'lines+points' | 'points+lines'


Lines:
------
    lineSpec        ::=  lineDefn | lineUpdate

    lineDefn        ::=  'line' lineName eq lineItemSeq

    lineUpdate      ::=  'line' lineName 'is' lineName 'with' lineItemSeq

    lineItemSeq     ::=  lineItem ',' lineItem*

    lineItem        ::=  widthSpec | colourSpec | pointSpec | pointSizeSpec | plotSpec

    widthSpec       ::=  'width' eq num

    colourSpec      ::=  colourToken eq colourDesc

    colourToken     ::=  'colour' | 'color'

    colourDesc      ::=  rgb | colourNaming

    rgb             ::=  'rgb' hex2 hex2 hex2
                      |  'rgb' posNum posNum posNum

    colourNaming    ::=  shading  rootColour

    shading         ::=  'light' | 'dark' | 'basic'

    rootColour      ::=  'red'  | 'blue'   | 'green'  |  'grey'
```

```
                    |  'gray' | 'yellow' | 'orange' | 'purple'

    pointSizeSpec   ::=  'pointsize' eq num

    pointSpec       ::=  'point' eq pointName

    pointName       ::=  filling rootPoint

    filling         ::=  'solid' | 'empty' | 'filled' | 'unfilled'

    rootPoint       ::=  'plus'     | 'cross'   | 'star'   | 'box'
                    |  'triangle' | 'diamond' | 'circle' | 'square'

    lineName        ::=  id


Samples:
--------
    sampleSpec      ::=  'sample' 'size' eq sampleType

    sampleType      ::=  sampleSize | sampleFraction |  sampleRange

    sampleSize      ::=  'final' int points

    points          ::=  'pts'  | 'points' | 'samples'

    sampleFraction  ::=  'final' num percent

    percent         ::=  '%' | 'percent'

    sampleRange     ::=  '(' data '<=' data ')'

    data            ::=   '_' | num | exptDir


Fonts:
------
    fontSpec        ::=  'font' fontQualifier eq fontDefn

    fontQualifier   ::=  '.axes' | '.title' | '.key' | '.all'

    fontDefn        ::=  '(' fontFamily  fontModifier  ')'

    fontFamily      ::=  'plain' | 'fixed' | 'sans-serif'

    fontModifier    ::=  ',' fontStyle  ',' fontSize

    fontStyle       ::=  'bold' | 'italic' | 'bold-italic' | 'italic-bold' | 'normal'

    fontSize        ::=  int


Histogram spec:
---------------
    histogramSpec   ::=  'histogram' histoParam ',' histoParam

    histoParam      ::=  histoBin | histoRange

    histoBin        ::=  'bins' eq int

    histoRange      ::=  sampleRange | autoRange

    autoRange       ::=  'auto' 'range'


Miscellaneous:
--------------
    eq              ::=  '=' | ':'
```

```
Lexical Classes:
----------------
  arithExpr ==   numerical expressions involving formal parameter names and other functions.
  curveExpr ==   numerical expressions involving column names, "x" and other functions.

  string    ==   quoted strings (using balanced pair of either ¨ or ´)
  id        ==   identifiers

  num       ==   general numerical literals
  int       ==   integer literals
  posNum    ==   simple positive/unsigned numeric literals
  hex2      ==   a pair of hex digits [0-9a-f]

Tokens are generally not sensitive to case - although of course user-specified
strings remain case sensitive.
```

B.2 Output directories and names of charts

- The output graphs etc. are placed inside a separate subdirectory of the DXM root Directory called `graphs`. Each experiment directory is replicated inside the `graphs` directory and contains the resulting graphs etc.
- This helps provide an explicit and clear scheme for naming graphical output and allowing for intermediate results needed for plotting.

B.3 Naming scheme for charts and related files

Note that each chart will be computed from a data file specified via the data sources. Individual charts are named according to the data source they are derived from ...

Only PNG format graphics is supported - hence the `.png` file extension for all charts produced.

Let `cName` be the (unique) name of the chart. For each kind of data source, we produce chart output named as follows:

CSV filename pattern:	Graph output filename
`summaryStats.csv`	`graphs/summaryStats__cName.png`
`exptDirName/stats.csv`	`graphs/exptDirName/stats__cName.png`
`exptDirName/exp_num.csv`	`graphs/exptDirName/exp_num__cName.png`

Grouped charts	`graphs/exptDirName/group__cName.png`
	
	These have data source statements like:
	`expDirName/run group 13-57`
	
	(Note: this relies on the fact that a chart `cName` is either grouped or not, and that there is at most one grouped chart for that particular chart name.)

There are two other types of file needed to produce the graph/chart output:

Gnuplot script files
There is one of these for each chart and this is saved with the same name as the corresponding chart – with file extension `.gp` (e.g. `graphs/summaryStats__cName.gp`).

Intermediate data files
These contain derived data tables which are needed to, for example, compute various outputs for display.

For simple data hygene reasons, we choose not to rely upon Gnuplots internal functions or its data analysis features, but instead we make use of Python code for tabulation. These files are saved with the name given as follows: Suppose the chart is named as:

`graphs/exptDirName/exp_num__cName.png`

then any needed intermediate data file(s) are named:

`graphs/exptDirName/exp_num__cName__data`N`.csv`

where N is a number. These files will be referred to by the Gnuplot script to build the graphics file.

B.4 Usage information for the DXV tool

```
dxv - Demos eXperiments Viewer         Version: v1.2b, Date: 13th October 2008

Usage: dxv <viewplan>   -- Read the viewplan plan and generate diagrams/charts
                           for experiments.
       dxv -h           -- This message.

where:

   <viewplan> : This specifies the experiments plan and the charts/graphs to be drawn,
                using data contained in an DXM experiments tree.   The viewPlan itself
                specifies a corresponding DXM experiments plan.
```

C The DXMF tool

```
dxmf - DXM statistics filter          Version: v1.2b, Date: 13th October 2008

Usage: dxmf <options> <plan-file>       -- filter DXM summary stats.
       dxmf <options>
       dxmf

where <options> have the form (may be abbreviated):

      -help                  -- This help message.

      -output <output-file>  -- Specifies the CSV file where the filtered output is saved.
                                (default: filterStats.csv)

      -vars <vars-file>      -- Specifies the file containing which variables to
                                include in the output.  The file lists the variable names
                                separated by white-space.
                                (default: variables.txt)

and <plan-file> is the DXM experiments plan file - this specifies the root
directory for the experiments.  If this is omitted, then the current DXM plan
is specified by the __CUR_PLAN__ file, if that exists.

Note: like all DXM scripts, this script must be applied in the *parent* of the
experiments directory i.e. where the DXM file is typically located.
```

D Seed files and experiments – the DXMS tool

In general, the simulations are dependent upon settings of their internal parameters etc. and the state of a (deterministic) pseudo random number generator. This generator needs an initial seed value for it to produce values. By default, all Demos2k simulation runs start from a fixed seed value (120). Successive runs then simply start from where the previous run leaves off (this is also described in section § 5.1).

This is fine - except we may want to use a seed sequence that is generated independently of the current simulation. For example, we may want to comparing variations of the same basic simulation against each other, on a run-by-run basis (i.e. horizontally), rather than simply looking at the stats looking at the end of a series of runs.

To do this, we can specify in DXM the use of an externally generated seed file to be used to start each run. DXM provides ways of specifying the seed file to be used - or to generate a new seed file based upon a couple of simple parameters. Additionally, the DXMS seed file generation utility may be used to generate stand-alone seed files.

```
dxms - DXM seed file generator       Version: v1.2b, Date: 13th October 2008

Usage: DXMS    <options>
       DXMS

where <options> have the form (can be abbreviated):

  -help         : This help message

  -file <file>  : Filename to save seed entries
                  (default : seed.txt)

  -init <num>   : Initial seed value for generator and will be the first value output
                  (default : 1066)

  -step <num>   : Number of iterates made of randGen() between output entries
                  (default : 5)

  -size <num>   : Number of entries in file
                  (default : 200)

The seed files generated contains positive integers in the range (1,
2147483646), one per line. These are suitable for use as seed files in DXM
(see documentation for the DXM file).

RandGen:
--------
The psuedo-random number generator used is called randGen and is defined in
the file RandGen.py in the DXM installation.  The intention is that users can
replace this generator function by another if they wish.
```

E Demos2k

In this section we discuss some basic aspects of Demos2k – the expected ‘shape’ of models, as well as details of how to get hold of the tools and documentation.

E.1 The Shape of Demos2k models

The general shape of a typical Demos2k model goes as follows:

1. Constant definitions:
   - Demos2k constants are special in that they may be defined in terms of probability distributions — each time such ’constants’ are evaluated during simulation, a fresh sample is taken from the specified distribution. The probability distributions supported include standard distributions such as Uniform, Binomial, Geometric, Negative Exponential, Normal, Poisson, and Weibull, as well as arbitrary point/discrete distributions;
2. Global variable definitions;
3. Resource definitions:
   - In Demos2k, resources represent pure synchronisations (in the process-calculus sense) and can be claimed and released by means of `getR` and `putR` expressions;
4. Bin definitions:
   - In Demos2k, bins represent synchronisable entities (note that the term ‘resource’ is used in the rest of the paper to encompass both the DEMOS notion of ‘resource’ and the DEMOS notion of ‘bin’, as described here) into which some quantity of material may be placed and retrieved. These may be used to provide the effect of one entity making a synchronous, concurrent process call on another;
5. Class definitions:
   - In Demos2k, each entity is a concurrently executing instance of some class. Classes thus represent the behaviour of entities in conventional procedural terms, by manipulating resources in some fashion and by ‘holding’ (letting time pass) for defined periods of time;
6. Initial model population, and entity creation;
7. Run length control, typically a `hold` of some fixed duration;
8. The all-important `close` statement ends the simulation run.

In this form, we may regard Demos2k descriptions as defining system behaviour in terms of a Dijkstra-like guarded command language. All active commands test the current system state. If the condition they represent can be met then they are executed — otherwise they are blocked until such time as the condition holds, if at all. Note that Demos2k simulations will typically run for a specified length of time. If either deadlock or livelock arise during simulation runs then these situations are checked for pragmatically. The major difference between process modelling languages (like Demos2k) and pure guarded command languages is that the conditions have side effects, due to the assignment of resource to the active entity. Hence change of state is mediated not only by assignment to variables, but also by the competing claims of resource.

The standard Demos2k package does contain a form of experiments management tool - our contribution with DXM is to provide a more extensive capability in a scriptable, externalised form that can be conducted offline (i.e. batch) and to produce summaries in spreadsheet format.

E.2 Getting Demos2k

Demos2k is currently accessible for download as Open Source in two ways:

- From the website: `http://www.demos.org`. Although this currently contains much of the reference material (i.e. operational semantics, guide, etc.), the download is a little out-of-date.
- A more up-to-date download is available from *SourceForge* under the *Seymour*[4] project: See `http://sourceforge.net/projects/seymoursys/`

  The download package contains all the relevant documentation for Demos2k.

[4]*Beware: there have been several projects called Seymour in SourceForge - so be sure to get the right one!*

F Complete example

This appendix contains the complete Demos2k example used earlier in § 2.

```
(* threeTier+Customers.d2000

   Brian Monahan, Systems Security Laboratory

   (C) Hewlett-Packard 2008


   A classic, three-tier service chain - with customer demand

   There are four kinds of producer/consumer process - source, reseller, retail and
   customer, connected as follows:

      source -> reseller -> retail <- customer

   The idea is that:
      - the source process represents supply of basic widgets
      - the reseller process adds value to the basic widget, making ready for
        sale by retail.
      - the retail process sells finished widgets to customers.
      - the customer process represents customer demand.

   The retail process can only make sales when they have both a
   customer and a widget from the reseller process.


   The parameters of this system are:

     - Numbers of process instances of each kind.
     - The average time that each kind of process takes to perform its
       activities, whatever they may be.

   Outputs are:

     - Total numbers of outputs from source and reseller processes.
     - Total number of inputs to reseller and retail processes.
     - Total number of customer retail requests

     - Total amount of time spent between source outputs.
     - Total amount of time spent by resellers between input from sources
       and output to retail.
     - Total amount of time spent by retail between inputs.

     - Total amount of sales made by retail

     - Total number of customers.
*)

//************************************
//*  Demos2k systems settings (DXM):
//*
//*     LIVELOCK-STEPS =  100000
//*     SPAWN-LIMIT    =   10000
//*
//************************************

cons runTime = 1000;

cons numOfCustomers     = 1;
cons numOfSourceProc    = 1;
cons numOfResellerProc  = 1;
cons numOfRetailProc    = 1;

cons customerAvgRequestInterval  =  4;
cons sourceResetAvg              =  0;
cons sourceAvgSupplyTime         =  4;
cons resellerAvgProcessTime      =  4;

cons requestInterval   =  negexp (customerAvgRequestInterval);
cons sourceResetTime   =  normal (sourceResetAvg,          sourceResetAvg/5);
cons supplyTime        =  normal (sourceAvgSupplyTime,     sourceAvgSupplyTime/10);
cons processTime       =  normal (resellerAvgProcessTime,  resellerAvgProcessTime/12);

// Queues and bins
bin(customer_request, 0);
bin(source_to_reseller, 0);
bin(reseller_to_retail, 0);

// Activity of reseller
var totalCustomerRequests = 0;
var totalResellerActivity = 0;

// Queue lengths
var customerRequest_QLength  =  0;
var sourceResell_QLength     =  0;
var resellRetail_QLength     =  0;

// Numbers of outputs
var sourceOutputs    = 0;
var resellerInputs   = 0;
var resellerOutputs  = 0;

// Sales
var totalSales = 0;

// DXM bureaucracy
var demos_sample_tick = 0;

// Classes
class source = {
  repeat {
    hold(sourceResetTime);
    hold(supplyTime);

    sourceOutputs := sourceOutputs + 1;
    trace("sourceOutputs=%v", sourceOutputs);

    putB(source_to_reseller, 1);
    sourceResell_QLength := sourceResell_QLength+1;
    trace("sourceResell_QLength=%v", sourceResell_QLength);
  }
}

class reseller = {
  repeat {
    hold(Qpause);
    getB(source_to_reseller, 1);
    sourceResell_QLength := sourceResell_QLength-1;
    trace("sourceResell_QLength=%v", sourceResell_QLength);

    resellerInputs := resellerInputs + 1;
    trace("resellerInputs=%v", resellerInputs);

    hold(processTime);

    totalResellerActivity := totalResellerActivity + 1;
    trace("totalResellerActivity=%v", totalResellerActivity);

    putB(reseller_to_retail, 1);
    resellRetail_QLength := resellRetail_QLength+1;
    trace("resellRetail_QLength=%v", resellRetail_QLength);

    resellerOutputs := resellerOutputs+1;
    trace("resellerOutputs=%v", resellerOutputs);
  }
}

class retail = {
  repeat {
    try [getB(customer_request, 1), getB(reseller_to_retail, 1)] then {
      customerRequest_QLength  :=  customerRequest_QLength-1;
      resellRetail_QLength     :=  resellRetail_QLength-1;

      trace("customerRequest_QLength=%v", customerRequest_QLength);
      trace("resellRetail_QLength=%v", resellRetail_QLength);

      totalSales := totalSales + 1;
      trace("totalSales=%v", totalSales);
    }
  }
}

class customer = {
  repeat {
    hold(requestInterval);

    putB(customer_request, 1);
    customerRequest_QLength := customerRequest_QLength + 1;
    trace("customerRequest_QLength=%v", customerRequest_QLength);

    totalCustomerRequests := totalCustomerRequests + 1;
    trace("totalCustomerRequests=%v", totalCustomerRequests);
  }
}

// Measurement
class measure = {
  entity(M,measure,1);

  demos_sample_tick := demos_sample_tick + 1;
}

// Launch process entities
do  numOfSourceProc       { entity(source, source, 0); }
do  numOfResellerProc     { entity(reseller, reseller, 0); }
do  numOfRetailProc       { entity(retail, retail, 0); }
do  numOfCustomers        { entity(customer, customer, 0); }

// Launch measure class
entity(measure, measure, 0);

hold(runTime);

close;
```