

# Batch Scripting Facilities for Design/CPN

Bo Lindstrøm & Lisa Wells

Department of Computer Science

University of Aarhus

Ny Munkegade, Bldg. 540

DK-8000 Aarhus C

Denmark

E-mail: {blind,wells}@daimi.au.dk

## Abstract

This paper contains a design proposal for facilities for doing batch simulations in Design/CPN. These facilities can be used for running a group of simulations without user interaction. We discuss what kind of functionality is needed to run batch simulations. We also give proposals on how to implement this functionality in Design/CPN. To illustrate the use of the batch scripting facilities we present examples of how batch simulations can be defined.

**Keywords.** Coloured Petri Nets, Design/CPN, batch simulations.

## 1 Introduction

It is often necessary to run many simulations in order to obtain satisfactory or necessary results. There are many situations in which a series of simulations of Coloured Petri Nets (CP-nets or CPNs)[2–4] need to be run in Design/CPN[1], and it is not always necessary for a user to be present while the simulations are executed. Several such situations are:

- Calibration - tuning parameters in the CP-net so that simulation results resemble results from the modelled system.
- Performance analysis and/or data collection - analysing results generated by different start parameters.
- Sensitivity analysis - discovering which parameters have the most significant impact on simulation results.

Design/CPN lacks support for running a group or *batch* of simulations without user interaction between simulations. This paper describes ideas for the design of batch facilities which could solve this problem. The batch facilities will largely consist of a number of *primitives* that can be combined in *scripts* for defining a batch of simulations. Batch scripts will simply be ML code which executes a series of ML functions, also referred to as primitives. The batch facilities are currently being implemented. With the batch facilities it will be easy to specify and run a batch of simulations. It will be possible to change model parameters between simulations and specify where output should be saved. Furthermore, using the batch facilities will save time when running several simulations because the simulations will be automatically run one after the other without having to wait for a user to explicitly start each simulation.

The structure of this paper is as follows. Section 2 describes the basic design proposal for the batch facilities in terms of how primitives and scripts can be used. Section 3 describes some of the changes that need to be made in existing primitives so that they can be used to define a batch run. Section 4 discusses user scripts, which are scripts that can be totally defined, and therefore customised, by a user. Finally, Sect. 6 presents a design proposal for a standard script which allows a user to add some extra functionality to a partially-predetermined batch script.

## 2 Basic Design Proposal

One of the simplest forms of batch simulations is the execution of a constant number of simulations. Figure 1 contains a script that can be used to execute a given number of simple simulations. The primitive `simulate` will run a simulation, and the primitive `init_state` initialises the state of the CP-net. Evaluating the expression `runBatch(10)` will execute 10 simulations, and the state of the CP-net will be initialised before each simulation is started.

---

```
fun runBatch (noOfSimulations) =  
  if noOfSimulations > 0 then  
    (init_state();  
     simulate();  
     runBatch (noOfSimulations-1))  
  else ();  
  
runBatch 10;
```

---

Figure 1: A simple batch script.

### 2.1 Primitives

Running a batch of simulations usually requires more than just executing a group of simulations. Users will use the batch facilities for different purposes which means it should be possible to customise a batch run. It has to be possible for a user to specify actions, details, or changes to be made for each individual simulation. The central idea of this design proposal is to ensure

that a set of essential primitives are available for defining a batch run. The following list is a set of actions that are likely to be useful when specifying a batch run. For each action, it is noted whether or not a primitive for executing the action currently exists in Design/CPN.

1. **Update reference variables.** A reference variable can be used for many different purposes in a CP-net: as a parameter for a function, as an initial marking, to identify an input file, to model a characteristic of a part of the modelled system, e.g. the speed of a CPU, etc. The value of reference variables can be changed anywhere which means that reference variables will be particularly useful when running batch simulations. *Primitive:* the ML assignment operator `:=` can be used to change the value of a reference variable.
2. **Manipulate input files.** Files are often used to provide input for a simulation. *Primitive:* there are already primitives for opening and closing files by means of standard ML functions.
3. **Initialise the state of the CP-net.** *Primitive:* the function `init_state` changes the state of a CP-net to the initial marking. For timed CP-nets, the function `init_time` resets the model clock to the initial time.
4. **Initialise data collectors.** This is relevant only when running batch simulations in the Design/CPN Performance Tool [6, 7]. *Primitive:* the function `initAllDataCollectors` initialises all data collectors.
5. **Set and read simulation options.** For example, simulation stop criteria. *Primitive:* there are currently no primitives for setting or reading options, but it will be easy to add such primitives. Simulation options are currently set using dialog boxes.
6. **Run the simulation.** *Primitive:* the function `simulate`<sup>1</sup> will execute a simulation.
7. **Gather results.** A user may want to collect the following types of information after a simulation finishes: markings, values of reference variables, performance reports, simulation options. *Primitive:* files can be opened, updated, and closed by users. Reference variables can be dereferenced, and the values can be converted to strings which can be saved in files. The function `savePerformanceReport` will save a performance report in a file when simulating in the performance tool.
8. **Stop batch run.** It is necessary to be able to stop a batch run. Stopping a batch could depend on different types of criteria, e.g. number of simulations run, the value of a reference variable, the number of steps taken, the output of the previous simulation, etc. *Primitive:* users can define functions which examine different criteria, and which determine whether the batch should stop based on the status of the criteria. The functions `step` and `time` can be used to examine how many steps have been taken and the current model time.

---

<sup>1</sup>The primitive `simulate` is just an alias for the function `sac_step` which is used to execute a simulation in Design/CPN.

If the batch facilities contain primitives for executing these 8 actions, then a wide variety of batch simulations can be defined. Most of the actions mentioned here are already supported in the version of Design/CPN that includes the Design/CPN Performance Tool. Some of the primitives that exist will be modified slightly in the future, and other primitives need to be created. Section 3 discusses the modifications that will be made.

## 2.2 Scripts

The aforementioned actions can be combined in different ways to create different batch runs. For example, using varying combinations of these actions in `runBatch` in Fig. 1 would result in different forms of batch runs. We propose using ML functions or expressions to specify exactly which actions should be executed for a batch of simulations. Such a function or expression will be called a *script*. Again, Fig. 1 is an example of a simple script. The reason for selecting ML as the scripting language is that ML is used throughout Design/CPN.

Exactly how these scripts should be defined and executed is an important issue. In order to make the batch facilities as general as possible, it must be possible for a user to specify precisely which actions should be executed and when they should be executed during a batch run. For this reason, the facilities will support *user scripts* with which a user can totally define a batch run using the available primitives. A user script will consist of arbitrary ML code. Section 4 presents examples of user scripts.

The building blocks of scripts are primitives. As the name indicates these are low-level functions. Certain actions, e.g. updating variables, will often be repeated in different batch runs. In order to make the batch facilities easier to use, high-level functions for performing often used functionality will be introduced. These high-level functions are discussed in Sect. 5.

While user scripts may be useful for defining special or non-standard batch runs, they may be difficult to define for people with limited ML experience. It is also reasonable to expect that many users will define similar batches. For example, many batch runs will probably consist of changing simple variable values, running simulations and gathering relevant results. Therefore, the batch facilities will eventually include support for running standard batch runs or *standard scripts*. A standard script will often be parameterised, offering some predetermined functionality, but it will still be possible for a user to specify certain actions to be executed during the batch run. Examples of a standard script will be discussed in Sect. 6.

The idea for supporting both user scripts and standard scripts was inspired by the Design/CPN Occurrence Graph Tool (OG-Tool, [5]). In the OG-Tool queries can be made concerning the details of occurrence graphs. There are a number of standard queries which are either totally predefined or parameterised. For the parameterised queries a user needs only to specify relatively simple parameter values. These two types of queries are very easy to use. Finally, it is possible for a user to totally define his own queries. User-defined queries can be used to investigate characteristics that are relevant only for a given net. For example, a user query could be used to discover if a token with a certain colour was ever found on a given place. By supporting these three different types of queries, the query facility in the OG-tool is both user friendly and general. The batch facilities will support user scripts and standard scripts in an attempt to achieve the same balance between user friendliness and generality.

## 3 Basic Modifications

Most of the primitives that were mentioned in Sect. 2.1 can be used as they are. However, there are some primitives that will be modified in order to make them more useful within the batch facilities. One of the actions that will be used frequently is initialising the state of the CP-net between simulations. Section 3.1 discusses the implications of and problems concerning initialising the state of a CP-net. Section 3.2 contains suggestions for improving the functionality of the `simulate` primitive which executes a simulation. Section 3.3 discusses what kind of primitives need to be created for setting and reading options.

Before discussing the modifications that will be made to primitives, it should be noted that slight modifications will also have to be made to the user interface of the simulator for running batch simulations. The user interface of Design/CPN provides feedback about the status of a simulation to a user. This is done by means of dialog boxes, updating the status bar, beeps, etc. In particular, a dialog box is opened when a simulation stops, and this dialog box must be closed by user interaction before a new simulation can start. Since the purpose of batch simulations is to execute several simulations without user interaction, the batch facilities must include a primitive for disabling the relevant parts of the user interface when running batch simulations. The feedback that is provided by the user interface could instead be redirected to a log file.

### 3.1 Initialise State Function

A user will have to decide whether or not the state of the CP-net should be initialised between simulations in a batch of simulations. Section 3.1.1 discusses the difference between initialising the state and not initialising the state of the CP-net between simulations. Item 1 in the list of essential actions (Sect. 2.1) mentions that variables may be changed between simulations. It should be possible to influence the initial marking of a CP-net by updating reference variables. Section 3.1.2 discusses why the function that calculates the initial marking will need to be modified in order to make this feasible.

#### 3.1.1 To Initialise, or Not

Whether or not the state of a CP-net is initialised between simulations will have a significant impact on the meaning of the batch simulations. In particular, it will influence the independence of the individual simulations from each other.

If the state of the net is initialised between simulations, then each simulation in a batch run is a new simulation. In this case all simulations will be independent from the others. Every simulation will start from an initial marking where  $step = 0$ . This type of batch run can be used for a variety of different purposes. By initialising the state – and possibly setting the random seed of the simulator (see Sect. 3.3) – it is possible to use different start parameters or different input sources and compare the performance of the system for different start conditions. It is also possible to run several independent simulations for one given set of parameters, and then study variations in the performance of the system for the different simulations. This last option is useful when the behaviour of the system is non-deterministic.

The other alternative is to choose not to initialise the state between simulations. If no state initialisation is undertaken, the batch run itself is a single simulation that is divided into sub-simulations. The outcome of each simulation is dependent on the simulations that preceded it. This type of batch run can be used to collect data and differentiate between data from different intervals of one simulation.

### 3.1.2 Calculating the Initial State

We have previously mentioned that it should be possible for users to change the values of reference variables between simulations. These variables can be used, for example, as parameters for functions, in arc inscriptions, and in initial markings for places. The idea is that after changing the value of the variable before a simulation, the new value of the variable should always be used throughout the simulation. Unfortunately, the current implementation of `init_state` does not use the changed variable values. This section will describe this area in the current design of the state initialisation primitive in Design/CPN, and it will propose an alternative design for the primitive.

The following situation will be used to illustrate the difference between the current design of the state initialisation facility and our proposal. The left-hand side of Fig. 2 illustrates how we would like to be able to indicate the initial markings for a place. Assume that place **Place 1** should have an initial marking that is dependent on the reference variable *X*. The initial marking of **Place 1** could be calculated using the function `initFun` which dereferences the reference variable *X*.

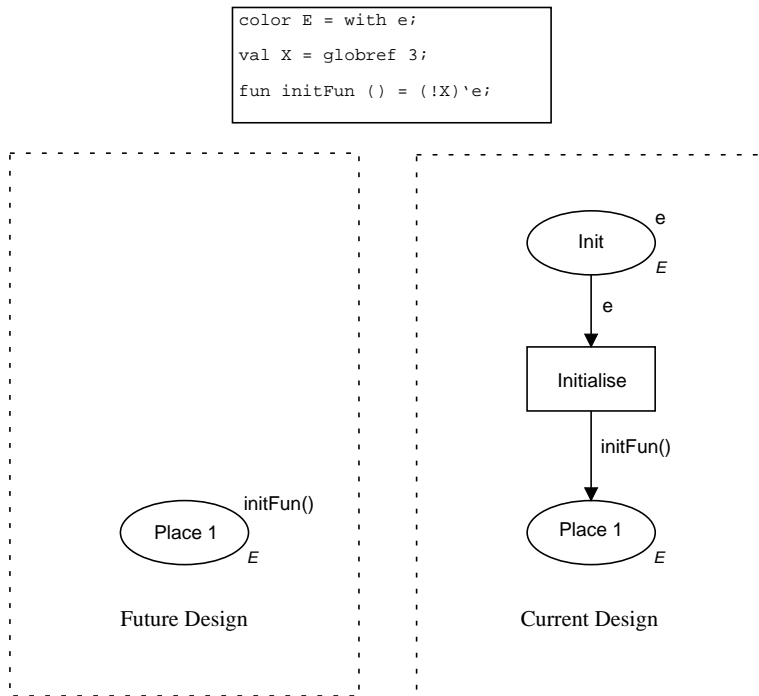


Figure 2: Calculating the initial markings.

In the current implementation of Design/CPN, the initial marking of the CP-net is calculated once and for all during the switch from the editor to the simulator. The marking for each place is calculated during the switch, and the marking is saved for possible later use. If a user chooses to return to the initial state while in the simulator, then the markings that were previously calculated are simply reinstalled for each place instead of being recalculated. Assume that the initial marking of **Place 1** was given directly by the inscription `initFun()` (as shown on the left-hand side of Fig. 2). Assigning a new value to  $X$  before returning to the initial state will *not* change the initial marking of the place. Installing pre-calculated markings can save execution time, but it means that it is difficult to use different initial markings.

If one is interested in using different initial markings, then there are ways to work around the problem in the current implementation. The right-hand side of Fig. 2 illustrates one workaround: using transitions to compute pseudo-initial markings that can differ from one simulation to another. Instead of declaring an initial marking for the place **Place 1**, it is possible to add extra nodes to the net for initialisation purposes. In this case **Place 1** will have an empty initial marking, and when the transition `Initialise` occurs, the desired tokens will be put on the places (see the arc from `Initialise` to **Place 1**). Thus, the pseudo-initial marking is created. Now, if the value of  $X$  is changed before initialising the state, then the pseudo-initial marking of **Place 1** will reflect the new value of  $X$ . With this workaround it is possible to define different “initial” markings without user interaction.

At the moment it is unclear how much extra time would be needed to recalculate the initial marking of a net when using the new design proposal. It may be the case that recalculating the initial marking during state initialisation requires significantly more time than simple re-installation of pre-calculated values. It may also be possible to combine the two strategies and recalculate only certain parts of the initial marking when returning to the initial state. With this alternative the “constant” initial markings of places could be calculated once and for all during the switch, while only the markings that could change from one simulation to another are recalculated when initialising the state of the net.

An alternative to ensuring that the desired initial marking is recalculated when the state of a CP-net is initialised, is to make adjustments to the marking after the state has been initialised. In the simulator, there is a dialog box that can be used to change the marking of a place. The functionality of this dialog box could be captured in a primitive, which could then be used to modify markings. The primitive `changeMarking` would take a place name, and a marking as parameters, and it would replace the old marking of the given place with the new marking. An advantage of having such a primitive would be that this primitive could be used at any point during a simulation and not just when initialising the state of the net.

## 3.2 Simulate

We now turn to the primitive for executing an individual simulation in a batch run. An individual simulation of a batch run will be a slightly modified version of a simulation that is currently started by invoking *Automatic Run* in the simulator. Section 3.2.1 contains design ideas for new stop criteria. Section 3.2.2 discusses ideas for better exception reporting during simulation.

### 3.2.1 Stop Criteria

Many simulations can run for many steps before there are no more enabled transitions. Some CP-nets give rise to infinite firing sequences which means that other simulations could continue running forever. Therefore, the batch facilities need some mechanism for specifying when each individual simulation is to stop. It is possible to allow a simulation to run until there are no more enabled transitions. In Design/CPN it is also possible to set some so-called stop criteria. The stop criteria that are currently supported are based either on the number of steps that have occurred or on the model time. These stop criteria are quite useful, but they are somewhat limited. In this section we propose designs for new stop criteria. These stop criteria would be incorporated in the standard simulator and would, therefore, also be available when using the batch facilities. In other words, these new stop criteria would not just be available when running batch simulations.

**User Defined** The current stop criteria are net-independent, i.e. the stop criteria can be used for every CP-net. It is currently impossible to define stop criteria that are dependent on how a net is defined (net-dependent), e.g. stop criteria based on markings or binding elements. In order to make the stop-criteria facilities net-dependent, we propose adding a function for setting a stop flag. When this function is invoked the simulator will stop after the current step. The function for setting the stop flag can be used anywhere in a model. This means that a simulation can be stopped at any time. Some examples are:

1. When a certain transition occurs. In this case the stop function could be written in the corresponding code segment.
2. When a specific marking has been reached. This will currently only be possible when in the performance tool. A data collector can be used to inspect the marking of the net, and could call the stop function if the appropriate marking is present.

Regarding item 2 there are two possible drawbacks to using the facilities of the performance tool only for stopping a simulation using the stop flag. The first drawback is that more time will be needed to make a switch to the simulator because extra structures need to be created during the switch for accessing markings and binding elements. The other drawback is that more time will be needed to run a simulation if the entire net marking is extracted after each step of the simulation. Preliminary tests show that simulation time increases by approximately 25% when the net marking is calculated after each simulation step (for more details see Sect. 8 in [7]).

**Real Time and CPU Time** The stop criteria that are currently supported in the simulator are dependent only on the state of a simulation of a CP-net, i.e. either the model time or the number of steps that have occurred. In some cases it may be useful to be able to stop a simulation after it has run for a certain amount of real time. For example, a user may want to stop a simulation 30 minutes after it had started. Similarly it might be useful to be able to stop a simulation after it has had a certain amount of CPU time. Using CPU time instead of real time would ensure that the simulation had actually run for the desired amount of time. It is likely that a simulation would

be only one of many processes running on a CPU, therefore real time would only indicate how long ago the simulation started, and not actually how long it had run. Supporting stop criteria depending on real and CPU time could be advantageous.

### **3.2.2 Exception Reporting**

Another aspect of the simulator that needs to be improved is exception reporting. Currently, a user is only notified that an exception has been raised causing a simulation to stop. No indication is given about what type of exception has been raised. There is also no feedback about where an exception has been raised, e.g. when evaluating a code segment or a guard. The performance tool provides some support for exception reporting (see Sect. 9 in [6]). However, this exception reporting is used only when an exception has been raised while executing performance tool specific functions, e.g. evaluating observation functions, opening observation log files, updating statistical variables within data collectors, etc.

Exception reporting facilities would be useful when running single simulations, and they would be especially useful when running batch simulations. If a batch simulation is started, and some or all of the simulations stop because of exceptions, then a user will want to know why the exceptions were raised. If detailed feedback is not provided when exceptions are raised during batch simulations, then it may be difficult to find the source of the problem for each individual simulation.

## **3.3 Set and Read Options**

There are several different types of simulation options. Stop criteria are simulation options. It is also possible to indicate whether step information or bindings should be saved in case a simulation report will be saved at the end of a simulation. Selecting a fast simulation or a fair simulation is also done by setting a simulation option. New primitives will need to be created for setting and reading simulation options.

There may be situations in which a user will want to set simulation options differently for each simulation in a batch run. For example, given the results of one simulation, it may be clear that the next simulation should run for more steps, in which case it would be useful to be able to set new stop criteria during a batch simulation. Another example of a simulation option that could be changed is the seed for the random number generator of the simulator. When several simulations are performed using different input, e.g. from a file, it could be useful to be able to minimise random behaviour by ensuring that each simulation uses the same seed when running a batch.

Currently, all simulation options are set using dialog boxes. There are no primitives for either reading or setting simulation options. Simulation options are saved using reference variables. Reading and setting a simulation option is simply a matter of dereferencing or reassigning the appropriate reference variable. A user should not have direct access to the reference variables in question, therefore data encapsulation methods should be used. To this end simple primitives can be made for reading and writing the relevant simulation options. It is possible to define

meaningful batch simulations without primitives for reading and setting options, but the batch facilities would be improved with such primitives.

## 4 User Scripts

In this section we illustrate how the batch scripting facilities can be used to create user scripts. User scripts are defined using the primitives discussed in Sect. 2.1. In addition to these primitives, it is also possible to define a script using any other user-defined ML function, ML primitive, or any ML function that is available in the simulator of Design/CPN.

### 4.1 Updating Several Variables

In this section we will give a simple example of a batch that runs simulations with different values of two reference variables. Batch simulations may be used to explore the impact of different combinations of parameter values on simulation results. Therefore, it will often be the case that the values of one or more reference variables in the CP-net will need to be changed before starting a new simulation. An example of such a situation is described in Example 1:

**Example 1** *Assume that  $x$  and  $y$  are parameters in a CP-net, and that they are declared in the global declaration box as global reference variables. A simulation of a model should be performed for every integer value of the variable  $x$  within the interval  $[3; 4]$  and for every even value of the variable  $y$  within the interval  $[2; 6]$ . This means that a simulation should be run for each of the following combinations of parameters  $(x, y)$ :  $(3, 2)$ ,  $(3, 4)$ ,  $(3, 6)$ ,  $(4, 2)$ ,  $(4, 4)$ , and  $(4, 6)$ .*

Figure 3 illustrates how Example 1 *could* be implemented using the primitives that are available in the batch facilities. In this user script the two reference variables  $x$  and  $y$  are updated between the simulations. A simulation is run for each combination of the parameter values that have been given. The model will possibly behave differently for each combination of the variables.

### 4.2 Data Collection

It is likely that the batch facilities will often be used to perform batches of simulations in the Design/CPN Performance Tool. The performance tool is used to collect and output data during simulations. Observation log files are generated automatically during a simulation, but a user needs to explicitly ask for a performance report to be generated. Figure 4 illustrates a user script for running simulations in the performance tool.

The parameter  $i$  is used to count how many simulations have been run. Before each simulation is started, the state of the CP-net is initialised. When initialising the state of a CP-net in the performance tool, any existing data collectors are deleted, and the performance functions are reevaluated in order to install new data collectors. When a simulation is run, data is collected

---

```

fun varScript(i) =
  if i <= 4 then
    let
      fun runLocalScript(j) =
        if j <= 6 then
          (x:= i;      (* x is a parameter in the model *)
           y:= j;      (* y is a parameter in the model *)
           init_state();
           simulate(); (* Run a simulation using the newly assigned values of x and y *)
           runLocalScript(j+2))
        else ()
    in
      runLocalScript 2;
      varScript (i + 1)
    end
  else ();

varScript 3;

```

---

Figure 3: A batch script changing two reference variables.

---

```

fun dataCollectionScript(i:int, n:int) =
  if i <= n then
    (init_state();
     simulate();
     savePerformanceReport("PerfReportBatch" ^ makestring(i) ^ ".txt");
     (dataCollectionScript(i+1, n))
  else ();

dataCollectionScript(1, 10);

```

---

Figure 4: A batch script for collecting data in the performance tool.

as usual, and statistical variables and observation log files are updated. After a simulation has stopped the primitive `savePerformanceReport` is used to save a performance report. Note that the value of  $i$  is incorporated into the name of the file containing the performance report. This is done to create unique file names that indicate from which simulation the performance report came. Before starting the next simulation the counter is incremented, and if  $n$  simulations have been run the batch stops. Evaluating `dataCollectionScript(1,10)` will execute ten simulations in the performance tool, and ten performance reports with unique names will be saved during the batch.

## 5 High-level Functions

When users write user scripts for running batches some functionality is implemented over and over again, e.g. changing values of variables between simulations. It would be useful if the batch facilities provided functions for standard actions. This section discusses some functions that will be provided in the batch scripting facilities to provide auxiliary functionality, and it will give a few examples of how the functions can be used in user scripts. Section 5.1 describes high-level facilities for updating variables. Section 5.2 discusses batch status files for giving an overview of the batch simulation by summing up some of the main results of the batch. Section 5.3 describes some of the results that can be gathered during a batch run, and it describes how to manage the many files that may be generated. Finally, Sect. 5.4 discusses support for standardised batch stop criteria.

### 5.1 High-level Support for Updating Variables

Figure 3 in Sect. 4.1 illustrates how to write a batch script for updating two variables with some specific values. When the number of different variables increases it may become a laborious task to write the code for considering all combinations of variables. We want to provide high-level support for indicating which variables should be updated before a simulation is started.

Supporting this kind of value<sup>2</sup> specification would be useful, and it would increase the user-friendliness of the batch facilities. When considering how general the specification of values of variables should be, we discussed several different possibilities. The first idea was that for each variable the user should be able to specify a minimum value, a maximum value, and a constant step value as described in Example 1 in Sect. 4.

We realised that one may not be interested in increasing the value by a constant for each simulation. Instead, one might want to specify a function that would return the step increase. By using such a step-increase function one could, for example, specify an exponential increase in the value of a variable between simulations. This step-increase function would make the value-specification facility more flexible.

The value-specification facility will include the following functionality. The user can specify which variables to change and how to change them, as discussed in the last proposal above, by means of the initialisation function `initVarUpdate` shown in Fig. 5. The parameter for this function is a list of records. Each record specifies how to update one reference variable: it contains the name of the reference variable (*var\_name*), the minimum value (*min*), the maximum value (*max*), and the step-increase function (*inc*).

After defining which variables should be updated, it is time to start simulating. After executing a single simulation the variables need to be updated. This will be done using the function `varUpdate`. By invoking the function `varUpdate` all of the variables are automatically updated properly. After each simulation a user ought to be able to check whether all combinations of values in the variables have been used. This is done by means of the function `stopVarUpdate`

---

<sup>2</sup>In this section the term *values* refers to values of type integer or real, though, some of the discussion may be relevant for other types, too.

---

```

fun varScript() =
  let
    fun runLocalScript() =
      if not (stopVarUpdate()) then
        (init_state();
         simulate();
         varUpdate();
         runLocalScript())
      else ()
    in
      initVarUpdate([
        {var_name=x, min=3, max=4, inc=fn () => 1},
        {var_name=y, min=2, max=6, inc=fn () => 2}]);
      runLocalScript()
    end;
  varScript ();

```

---

Figure 5: A batch script changing two reference variables using high-level primitives.

which returns `true` when no more variable updates need to be performed.

As an alternative to specifying the maximum value of a variable, we also considered supporting the use of a predicate function that could be used to indicate when a value no longer should be increased. However, to keep the design of this value specification facility simple, this idea was discarded. This facility is designed to be *easy to use* and to support *simple* variable updates. Requiring a user to define a predicate function for determining when to stop increasing a value would contradict these two goals. Furthermore, this facility of specifying intervals of values is particularly useful for users with limited experience in programming CPN ML. Therefore, it is important to keep the interface simple.

## 5.2 Batch Status File

After having performed a batch of simulations it may be difficult for a user to maintain a general overview of all the different simulations: which simulations have stopped with no more enabled transitions, which exceptions have been raised, and information like this. One way to preserve the overview is to save the status of each simulation in a so-called *batch status file*. The purpose of the batch status file is to present the most important information from each individual simulation.

Some of the questions that arise when designing support for a batch status file are the following. How detailed should the information in this batch status file be? Should the contents of the file be totally predetermined by the batch facilities, or should it be possible for the user to influence the contents of the file?

If the contents of the batch status file are totally predetermined by the batch facilities, then one of the advantages is that it will be easy to compare different batch status files. However, the information that will be contained in such a file would be restricted to information like:

- Simulation number.
- Number of simulated steps.
- Model time.
- Why the simulation stopped:
  - No more enabled transitions.
  - Stop criteria fulfilled.
  - Exception raised during the simulation.
- How long the simulation took:
  - In real time.
  - In CPU time.
- Use of memory.

In some cases this kind of model-independent information may not be sufficient for the user. The user may want some user-defined information to be written in the batch status file, e.g. the value of a particular variable. The reason might be that he would like a single file containing the main results of the batch of simulations. We think that it will be necessary to include the possibility to make the batch status file user definable. If we disallowed this possibility then the user would have to make the files himself – and therefore introduce yet another status file. If we allow the user to specify some text to be included in the batch status file, then we could omit the need for having more than one status file.

---

```

fun statusFileScript n =
  let
    fun runLocalScript i =
      if i > 0 then
        (simulate();
         updateBatchStatusFile("x has the value: "^(makestring (!x)));
         runLocalScript(i-1))
      else ()
    in
      initBatchStatusFile("filename.txt");
      runLocalScript(n)
  end;

statusFileScript 10;

```

---

Figure 6: A batch script updating the batch status file.

Figure 6 illustrates how a batch status file could be used. First of all the batch status file needs to be created. This can be done using the function `initBatchStatusFile` as shown in Fig. 6.

When the batch status file should be updated, the function `updateBatchStatusFile` must be invoked. The parameter to the function is the user-defined text that the user wants to be added to the standard contents of the file which are described above.

A user may not want to have all of the standard information in the batch status file. Other primitives could be designed so that a user could control exactly which information should be included. Separate primitives could be made for each of the items of information that were mentioned earlier. Then a user could choose exactly which information was relevant and use the appropriate primitives for including it in the batch status file. For example, a function like `updateBatchStatusFileModelTime` would add only the model time to the batch status file. The advantages of using such primitives is that a user can easily save relevant information, but he does not have to manage the details of opening, closing, and updating the status file himself.

### 5.3 Gathering Results

It is very likely that users will want to collect data and results from batch simulations. This means that the data and results need to be stored – usually in files. The batch status file can be used to collect some standard information concerning the status of simulations, and it can also include some user defined information. However, a user may want to save simulation results or data separately from simulation status information. For example, if a batch consists of twenty different simulations and five variables are changed in each simulation, then it would be desirable to remember the exact values for each variable within each simulation. In such a situation, it may be valuable for a user to save relevant information about each simulation after the simulation ends. Batch simulations will also often be used to compare and evaluate results from several different simulations. Again, files are needed for storing the results. When simulating in the performance tool, it is possible that many observation files will be created in addition to performance reports.

**Other Results** Here we will briefly mention other types of results or information that may be useful to save after a simulation is finished. The first item is a *simulation report*. Options can be set such that step information and bindings are saved. These can then be saved in yet another file. A primitive will need to be created for doing this, since the only way to save a report in the current implementation is via a dialog box. Another primitive will be needed for clearing the simulation report.

The results and information that we have discussed previously all have one thing in common: it is data that is extracted from either the CP-net or from the the simulator. In other words, the data represents a small part of the information that is available in the simulator. When running single simulations, the entire state of the simulator is often saved as an ML image. This state image can then be used later to continue a simulation or to avoid a lengthy switch from the editor to the simulator. It is conceivable that there will be occasional need for saving the state of the simulator during batch runs. A new primitive will be created for this purpose.

**File Management** It should now be obvious that it is quite possible that many files will be generated during a batch run. Remembering which files were generated by which simulation

may be a real problem. In order to alleviate this problem the batch facilities will include some high-level support for file management.

One way to differentiate between files is to ensure that the files have names that indicate from which simulation they were generated. This could be done by using a string value, or label, which contains information that is unique for each separate simulation. For example, in Example 1 in Sect. 4.1 the values of  $x$  and  $y$  could be used to make unique labels. It should be possible to provide a function for creating labels given variable names. For example if  $x = 3$  and  $y = 2$ , then the function could return the label “x3\_y2”. This type of label could certainly be used to define unique file names for different simulations. If one file is used to gather results from all of the simulations, then the label could also be used to separate results in that file.

Another alternative for managing files, is to save results in a new directory for each new simulation. Again, a label could be used to give each directory a unique, mnemonic name. The batch facilities could also include a function which automatically creates directories with simple names, together with a function that returns the name of the current batch directory. Figure 7 illustrates the use of primitives for creating new directories.

---

```
fun dirScript i =
  if i > 0 then
    (createNewDirectory();
     init_state();
     simulate();
     savePerformanceReport(newDirectoryName() ^ "PerfReport.txt");
     dirScript (i-1))
  else ();

dirScript 10;
```

---

Figure 7: A batch script creating new directories.

The function `createNewDirectory` would create a new directory name with a name that indicates when the directory was made. For example the directory name “Batch1\_Sim1” would indicate that the directory was created for the first simulation in the first batch of simulations. After the simulation is finished, the performance report is saved in this directory by creating a file name using the function `newDirectoryName` to access the name of the most recently created directory. Additionally, if the names of observation log files are also prefixed with the name of the new directory, then all of the performance related output from one simulation can be found in the same directory.

## 5.4 Batch Stop Criteria

In this section we consider high-level functions for specifying when to stop a batch of simulations. Of course the user could define stop criteria himself, but it may be useful to provide functions for the criteria that are frequently used. In the following we will describe some of the most useful criteria for determining when to stop a batch of simulations.

All of the criteria that are currently used to stop simulations could also be used as batch stop criteria. In other words, batch stop criteria could also depend on the number of steps taken or the model time. The simplest form of batch stop criteria would be to stop the batch after a constant number of simulations have been run. We have introduced simulation stop criteria that depend on real or CPU time. An often recurring question when running simulations that run for a long time is: will the batch ever stop, or is it cycling forever? It would also be useful to have batch stop criteria that ensure that the entire batch will stop within a certain time limit. New functions will need to be created for setting and checking batch stop criteria.

In contrast, by giving the user the ability to write his own criteria for stopping a batch, we allow the number of simulations to depend on the results of previously performed simulations (within the same batch run). This gives the possibility of setting parameters in a CPN model, then simulating the model, and deciding whether another simulation should be run based on the results obtained.

## 6 Standard Scripts

As described in Sect. 5 some batch runs have a certain common structure. An example could be that many batches have a well-defined loop structure, possibly with a well-defined initialisation before starting a simulation and a well-defined way of reporting results after a simulation. When such standard skeletons for implementing batches are identified, it is possible to integrate these in the batch facilities by providing standard scripts implementing these skeletons. Depending on how standardised the batch run is, it may be necessary for the user to be able to slightly extend standard scripts, i.e. to add some functionality to the standard script. Standard scripts are not replacements for user scripts, instead they are alternatives to user scripts. Some standard scripts may be for very specific, but often used, situations, e.g. just updating variables before running a simulation. This idea of using standard scripts is similar to using standard queries in the OG-tool [5] of Design/CPN. In this section we will propose a standard script that will make it simple to run standard batches. In the future when applying the batch facilities in practice, we will surely identify other useful standard scripts.

The idea behind a general standard script is that many batches will contain the scenario illustrated in Fig. 8. First of all, the batch job is initialised. To prepare the simulation the user specifies some initialisation, followed by possibly initialising the state of the model before starting a simulation. Then simulations are executed until a stop criterion is satisfied. When the simulation stops the user may want to gather results, and finally possibly start a new simulation.

This standard batch script can be used in many different situations. It is so general that many different batches can be expressed by means of this standard batch. It should not be hard to see that the batch script in Fig. 5 using high-level primitives could easily be implemented by means of this standard script. To get the intended behaviour the user supplies functions defining the user-specified behaviour as parameters to the standard script. It should be easier for the user to use such a standard script than having to implement the entire script himself.

---

```

fun simpleStandardScript(initBatch, endBatch, stopCriteria, beforeSimulation, doInitState, afterSimulation) =
  let
    fun runLocalScript() =
      if not (stopCriteria()) then
        (beforeSimulation());
        if doInitState() then
          init_state()
        else ();
        simulate();
        afterSimulation();
        runLocalScript()
      else ()
    in
      initBatch();
      runLocalScript();
      endBatch();
    end;

```

---

Figure 8: Implementation of a general standard script.

## 7 Conclusion

In this document we have described the design of batch scripting facilities for running batch simulations in Design/CPN. We have described what changes need to be made in the Design/CPN tool, and what functions would be useful to have when writing batch scripts. Many of the primitives that we have discussed are already implemented. As soon as users know what primitives are available, they will be able to begin writing user scripts. The facilities will be improved when new primitives are created. We also discussed different levels of writing batch scripts, i.e. user-defined scripts and standard scripts.

An issue that we have not discussed is how the scripts should be defined and invoked. There are at least two different possibilities. First of all, the user could write the batch script in an auxiliary node and then use ML-evaluate to start the batch. The batch could also be run by starting Design/CPN from the command prompt with a batch option and a name of a file containing a batch script. Then Design/CPN would automatically run the batch script without the graphical interface. This would make it possible to use Design/CPN as an engine, and it would be easier to use for a user who is not familiar with the graphical interface of Design/CPN.

In Sect. 6 we discussed using standard scripts for easing the work when implementing batch scripts. Another option would be to use templates for batch scripts. A user could select a certain template, and then some code, possibly something like Fig. 8, could be added to an auxiliary box. Then the user could modify this code to get the intended behaviour of the batch. Template scripts would be both easy to use and fully customisable.

The design of the batch facilities was developed immediately after designing, implementing and using the Design/CPN Performance Tool. This may have influenced the design of the batch facilities in both positive and negative ways. On a positive note, we were very aware of how

the batch facilities could be used to easily collect data from a variety of different simulations. Therefore, we were determined to include primitives that would ease the collection and output of data during batch simulations. On the other hand, we may have overlooked some important issues pertaining to simulating without using our data collection facilities in the performance tool. We are indeed aware that the present design is only based on needs that have been expressed by Design/CPN users. In the future we will look into other simulation tools that use batch facilities and consider relevant literature in the area of batch simulation. Additionally, we will also take into account comments and feedback from future users of the batch facilities.

To test the design of the batch facilities we made a few scripts using the primitives discussed in this paper. Preliminary tests have shown that the batch scripting facilities are well-suited for defining batch runs. From early experiments we think that this design provides very general and easy-to-use facilities for writing batch scripts. Additional experiments will indicate whether or not the design presented here will need to be modified.

## References

- [1] Design/CPN Online  
Online: <http://www.daimi.au.dk/designCPN/>.
- [2] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.
- [3] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.
- [4] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 3, Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997.
- [5] Kurt Jensen, Søren Christensen, and Lars M. Kristensen. *Design/CPN Occurrence Graph Manual*. Department of Computer Science, University of Aarhus, Denmark, 1996.  
Online: <http://www.daimi.au.dk/designCPN/man/>.
- [6] Bo Lindstrøm and Lisa Wells. *Design/CPN Performance Tool Manual*. Department of Computer Science, University of Aarhus, Denmark, 1999.
- [7] Bo Lindstrøm and Lisa Wells. *Tool Support for Simulation Based Performance Analysis using Coloured Petri Nets*, 1999. Department of Computer Science, University of Aarhus, Denmark.