

The Batch Simulation Framework

Francois Roewer-Despres

Last update: June 29, 2018

Contents

1	Introduction	4
2	Manager-worker architecture	4
3	Requirements and configuration	4
3.1	General	4
3.2	Syntax highlighting	5
4	Running BatchSim	5
4.1	General requirements	5
4.2	Minimal working example	5
4.3	Running BatchSim from the command line	6
4.3.1	Running the <code>BatchManager</code>	6
4.3.2	Running <code>BatchWorkers</code>	7
4.4	Running BatchSim from Eclipse	8
4.4.1	Running the <code>BatchManager</code>	8
4.4.2	Running <code>BatchWorkers</code>	8
4.5	Advanced BatchSim options	9
4.5.1	Options to the <code>BatchManager</code>	9
4.5.2	Options to <code>BatchWorkers</code>	11
5	The Property Specification Language	12
5.1	The target model and <code>Properties</code>	12
5.2	Property paths	12
5.3	Simulation tasks and property-value pairs	12
5.4	Combinatorial property specifications	13
5.5	Probabilistic property specifications	13
5.6	Running hybrid combinatorial/probabilistic simulations	13
5.7	Property specifications	14
5.7.1	Property path syntax	14
5.7.2	Combinatorial specification syntax	14
5.7.3	Probabilistic specification syntax	15
5.7.4	General syntax	15
5.7.5	Component identifier sets	16
5.7.6	Decorators	18
5.8	Control structures	19
5.8.1	<code>skip</code> statements	19
5.8.2	<code>redef</code> statements	19
5.8.3	Jython code blocks	20

6	Customizing BatchWorkers	22
6.1	What BatchWorkers do	22
6.2	Subclassing BatchWorkerBase	23
6.3	Conditions and the StopConditionMonitor	25
6.3.1	The need for Conditions and the StopConditionMonitor	25
6.3.2	How Conditions and the StopConditionMonitor work	26
6.3.3	Adding Conditions to a StopConditionMonitor	27
6.4	Customizing a Jython driver script	28
6.5	Using the SimpleTimedBatchWorker	29
7	Miscellaneous	29
7.1	Killing a BatchSim session	29
7.2	Post-processing	29
7.2.1	Merging all BatchWorker output files using the <code>taskno</code> instance variable	29
7.2.2	Cleanup	30
7.3	Useful diagrams	30
7.4	Ideas for future work	30

1 Introduction

The Batch Simulation Framework (BatchSim) is a framework for running large batches of ArtiSynth simulations automatically and in parallel. The user specifies desired input configurations of a target model, then, for each configuration, BatchSim automatically runs a simulation and records desired outputs, with simulations optionally being performed in parallel.

The input configurations are specified using ArtiSynth's `Property` mechanism, enabling BatchSim to be model-agnostic. Throughout this document, familiarity with `Properties` is assumed. For an introduction to `Properties`, see the [ArtiSynth Modeling Guide](#). For details, see the [Maspack Reference Manual](#).

Throughout this document, assume that `<ArtisynthRoot>` and `<ArtisynthModels>` denote the paths to the root folder of the ArtiSynth installation and the ArtiSynth Models extension, respectively. Given this, the root folder of BatchSim is `<ArtisynthModels>/src/artisynth/tools/batchsim` (referred to as `<BatchsimRoot>` for short).

BatchSim uses ArtiSynth's Jython interface. Throughout this document, familiarity with this interface is assumed. For details, see [Interfacing ArtiSynth to MATLAB and Jython](#).

2 Manager-worker architecture

BatchSim treats all simulations it is meant to run as independent. That is, the output of one simulation cannot influence the input to another. As such, performing multiple simulations becomes an [embarrassingly-parallel problem](#). BatchSim exploits this potential for parallelism by adhering to the manager-worker architecture (a special case of the [client-server architecture](#)). By following this design pattern, BatchSim achieves terrific load balancing and serves as a general-purpose framework for automatically running simulations in either distributed or parallel fashion.

Within the context of BatchSim, the manager-worker architecture works as follows. One `BatchManager` process creates a number of “tasks” (based on user specifications), where each task represents a simulation to perform. The `BatchManager` then places these simulation tasks in a “bag of tasks.” Next, a certain number of `BatchWorker` processes are created. Each `BatchWorker` requests a task from the `BatchManager`, completes the task (by performing the simulation the task describes), records user-specified simulation results as output, and repeats until the bag of tasks is empty. The `BatchWorkers` complete their tasks independently and in parallel (on a multicore system) or in a distributed fashion (on a multicomputer system), enabling a large **batch** of **simulations** to be completed in less time.

BatchSim includes a number of convenience methods for recording certain kinds of simple simulation outputs, but custom outputs can be specified by supplying custom `BatchWorkers` to BatchSim (see [Section 6.2](#)).

3 Requirements and configuration

3.1 General

BatchSim can run from within Eclipse or from the command line. However, the latter is *significantly* easier since BatchSim requires at least two processes to be running simultaneously (one `BatchManager` and at least one `BatchWorker`). This is possible but quite cumbersome to do in Eclipse, whereas using two separate command line consoles (or using background processes) makes running BatchSim from the command line trivial.

Whether running in Eclipse or from the command line, the following steps are necessary.

1. Ensure ArtiSynth is properly installed, and that all environment variables needed to run ArtiSynth are set. Note that Eclipse environment variables do not automatically create equivalent command line environment variables, and vice versa. Refer to the [ArtiSynth Installation Guide](#) for details.
 2. Ensure `<ArtisynthRoot>/classes` and `<ArtisynthModels>/classes` are on the Java class path. If additional custom classes are used, these must appear on the class path as well.
 3. (Optional) In addition to hardcoded values, BatchSim can use random simulation inputs from a variety of [probability distributions](#). In this case, an additional Java library, called JDistlib, is required to run BatchSim. The JDistlib jar can be downloaded [here](#). Once downloaded, ensure the jar file is on the Java class path.
-

Note: ArtiSynth does not ship with JDistlib because its license is slightly stricter than ArtiSynth's, leading to open-source incompatibilities. By using JDistlib, you agree to its license. **This only applies to using BatchSim with random simulation inputs (see next Note).**

Note: JDistlib is **only** required for running probabilistic simulations **directly** in BatchSim. If you do not agree to JDistlib's license, then probabilistic simulations can still be approximated by first sampling from the desired probability distribution outside of BatchSim (using software such as R or Python) and then hardcoding the resulting values in BatchSim.

3.2 Syntax highlighting

The input configurations file to BatchSim will contain code written in a very simple programming language called the Property Specification Language (PSL) (see Section 5). Syntax highlighting for PSL has been developed for Eclipse and Vim. See `<BatchsimRoot>/syntaxColoring` for instructions on how to get this feature up and running.

Note: at present, using the syntax highlighting in Eclipse is quite cumbersome. Using Vim is recommended.

4 Running BatchSim

Running BatchSim from the command line is recommended (see Section 3.1), but running from Eclipse is also possible.

4.1 General requirements

To run BatchSim (from Eclipse or the command line), a number of files will be needed. Below is a summary of these file requirements. Details are given in the appropriate sections.

1. A target model (see Section 5.1) with appropriate `Properties` (see Section 5.2).
2. A `BatchWorker` to customize simulation stop conditions and output recording (see Section 6).
3. A very short Jython script for driving the custom `BatchWorker` (see Section 6.4).
4. A file containing the desired input configurations (see Section 5).
5. (Optional) Since BatchSim potentially produces many output files, an output folder may be convenient for organizational purposes, especially in post-processing (see Section 7.2).

Note: simple defaults for the `BatchWorker` and the Jython script exist (see Section 6.5).

Note: for repeatable results while running probabilistic simulations, set the random number generator seed of the `BatchManager` (see `-s` option of the `BatchManager` in Section 4.5.1) and use the `-disableHybridSolves` option to ArtiSynth when running `BatchWorkers` (see Section 4.3.2).

4.2 Minimal working example

A minimal working example (MWE) of the entire BatchSim pipeline can be found in `<BatchsimRoot>/example`. It can be used as a way to try things out, or to make sure BatchSim is working properly. The only part of BatchSim not tested by this MWE by default is the proper functioning of JDistlib, which requires additional steps (see Section 3.1). To test this, use the file `<BatchsimRoot>/example/props-jdistlib.psl` as the input configuration file (see `-f` option of the `BatchManager` in Section 4.5.1).

This MWE was constructed as follows.

1. Decide on (or code up) a target model. In this case, [SpringMeshDemo](#) was used for simplicity.
2. Decide on a `Property` of the target model to configure during the simulations. In this case, the mass of the Particle named `pnt7` was used.
3. Write an input configurations file. In this case, two files were written. One is the “default” file, and is called `props.psl`. The other file is used to (optionally) run probabilistic simulations, and is called `props-jdistlib.psl`. To understand the contents of these files, see [Section 5](#).
4. Write a custom `BatchWorker` to add a simulation stop condition as well as output recording. In this case, the custom `BatchWorker` is `SpringMeshDemoBatchWorker`. To understand what it is doing, see [Section 6](#).
5. Write a very short Jython script to drive `SpringMeshDemoBatchWorker`. In this case, the Jython script is `exampleBatchDriver.py`. To understand what it is doing, see [Section 6.4](#).

Feel free to use this MWE to test out the commands/concepts in the remaining sections.

4.3 Running BatchSim from the command line

With a given target model, `BatchWorker`, Jython driver script, and input file, running `BatchSim` from the command line is done as follows.

4.3.1 Running the `BatchManager`

To run the `BatchManager`, in one console window, execute the command

```
> java artisynth.tools.batchsim.manager.BatchManager -f <path-to-input-file>
```

This starts the `BatchManager` as a standalone process, and is the quickest and most lightweight option.

Note: by default, the `BatchManager` looks for an input file called `props.psl` in the current working directory. If such a file exists (and is the input file you want the `BatchManager` to use) then the `-f` option to the `BatchManager` can be omitted, as follows

```
> java artisynth.tools.batchsim.manager.BatchManager
```

Unless otherwise noted, this is assumed to be the case in all subsequent examples involving the `BatchManager`.

Alternatively, execute the command

```
> artisynth -script <BatchsimRoot>/managerInit.py
```

This starts the `BatchManager` within an instance of `ArtiSynth`. This option is more heavyweight, but allows the `BatchManager` to verify that the contents of the input file represent valid `Properties` for the target model. Doing this once is plentiful; it is thereafter much easier to run the `BatchManager` as a standalone application. To perform this validation, the `-c` option must be passed to the `BatchManager`, and the target model must be loaded into `ArtiSynth`

```
> artisynth -model <target-model-classname> \
> -script <BatchsimRoot>/managerInit.py [ -c ]
```

Note: a `BatchManager`’s validation of `Properties` is not perfect. For each input configuration, the `BatchManager` will determine whether the target model has the appropriate `Properties` and whether every *hardcoded* `Property` value is valid, but no such guarantee can be provided for *probabilistic* `Property` values, since these are not known ahead of time. Instead, the `BatchManager` checks that an *arbitrary value* sampled from the probability distribution of each `Property` is valid. Due to the inherent randomness in sampling probability distributions, this validation works on a best-effort basis only.

A `BatchWorker` will also validate the `Properties` for the target model, but unlike the `BatchManager`, this validation is only done as quickly as the `BatchWorker` receives simulation tasks from the `BatchManager` (*just-in-time* validation). Just-in-time validation does have the benefit of verifying *probabilistic* `Property` values, though, since at this point a random draw from the appropriate probability distribution has already been performed for the particular simulation the `BatchWorker` is about to run, so the `BatchWorker` can validate that particular value before running that particular simulation.

For these reason, most users prefer running the `BatchManager` as a standalone process (relying on just-in-time validation), rather than within an instance of `ArtiSynth`.

4.3.2 Running `BatchWorkers`

Unlike the `BatchManager`, which is typically run as a standalone application, each `BatchWorker` *must* run within its own instance of `ArtiSynth` (in order to actually run simulations). The first step in running `BatchWorkers` is to decide how many to run. Call this number n . The appropriate number of `BatchWorkers` is entirely situation-dependent. For example, each `BatchWorker` writes its output to a separate file (see Section 7.2.1). For small simulation batches, using a single `BatchWorker` may work best, since it avoids the hassle of having to merge these separate output files. For larger batches, the parallelism offered by multiple `BatchWorkers` is critical to speeding up completion times (hours vs. days).

Note: in general, each `BatchWorker` uses of `OMP_NUM_THREADS` number of threads (see the [ArtiSynth Installation Guide](#)). As such, it is generally most efficient to use approximately as many threads as the number of cores on your system, as this maximizes parallelism while minimizing context switching and/or thrashing.

Once n is set, execute the following command n times to start n `BatchWorkers`

```
> artisynth -model <target-model-classname> \
> -script <path-to-Jython-driver-script>
```

There are essentially two ways of achieving this n -fold repetition.

1. Open n consoles, and execute the command once in each console. This gives access to each individual `BatchWorker`, but is quite tedious when n is large.
2. (Recommended) In a single console, execute the command n times in a loop. In this case, it is vital to run each `BatchWorker` as a background process. In addition, the `BatchWorker` console outputs can each be redirected to a separate file so they aren't interleaved in the console window. On a Unix-like system using Bash, this can be achieved with the following command

```
> for ((i = 0; i < n; i++)) ; do
>   CONSOLE_OUT="$i"_console.log
>   artisynth -model <target-model-classname> \
>   -script <path-to-Jython-driver-script> >> $CONSOLE_OUT 2>&1 &
> done
```

If no errors have been made, `BatchSim` should now be running simulations as per the specifications in the input file. In addition, the simulations will run in parallel if $n > 1$.

Note: **performance is greatly improved** by running `BatchSim` without the `ArtiSynth` GUI. Typically, `BatchSim` is started once with the `ArtiSynth` GUI and with $n = 1$ to visually verify that the simulations are running as expected (using the correct target model, with desired inputs, etc.). After just one simulation, `BatchSim` is killed (see Section 7.1) and then restarted with `ArtiSynth` running in “no GUI mode” and with $n > 1$ to increase performance.

To run `ArtiSynth` in “no GUI mode,” add the `-noGui` option to `ArtiSynth`. For example, to run a `BatchWorker` in “no GUI mode,” execute the command

```
> artisynth -noGui -model <target-model-classname> \
> -script <path-to-Jython-driver-script>
```

Using the `-noGui` option is usually coupled to low interactivity of the `BatchManager` (see option `-i` of the `BatchManager` in Section 4.5.1).

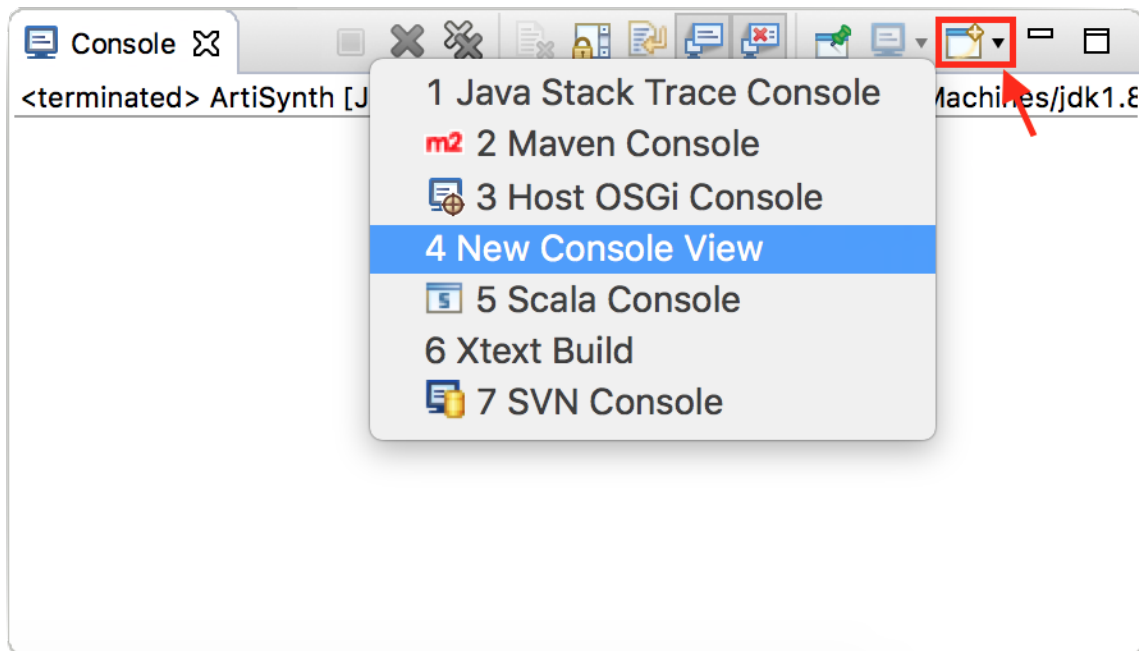


Figure 1: Creating a new Console View in Eclipse.

4.4 Running BatchSim from Eclipse

Running BatchSim from within Eclipse is similar to running it from the command line. One notable difference is that the working directory in Eclipse is a project root directory (<ArtisynthRoot> for BatchWorkers and <ArtisynthModels> for the BatchManager), so the path to the various files will have to be specified relative to those directories.

4.4.1 Running the BatchManager

To run the BatchManager **for the first time**: in the Eclipse Package Explorer, right-click on the [BatchManager.java](#) file in <BatchsimRoot>/manager/ and select Run As > Java Application from the context menu. This will start the BatchManager, but will immediately crash since the input file could not be found.

After running the BatchManager for the first time, right-click on the [BatchManager.java](#) file and select Run As > Run Configurations.... In the Run Configurations window, click on the Arguments tab, and add appropriate command line arguments (see Section 4.3.1) in the text box labeled Program Arguments. Then, click Apply and Run. Subsequently, the BatchManager can be started like any other Java program in Eclipse. Repeat this step whenever new command line arguments are desired.

4.4.2 Running BatchWorkers

To run a BatchWorker **for the first time**: in the Eclipse Console View, right-click on arrow next to the window icon with a plus-sign in the top-right corner (see Figure 1). From the menu that opens, select New Console View.

Next, return to the original console (where the BatchManager is running), and select the Pin Console icon (a window with a green pin) in the top-right corner (see Figure 2). This will pin the BatchManager process to that console.

Finally, run the desired BatchWorker by running ArtiSynth with the appropriate command line arguments for the BatchWorker (Run As > Run Configurations... > Arguments > Program Arguments > Apply > Run). Subsequently, the BatchWorker can be started like any other Java program in Eclipse. Repeat this step whenever new command line arguments are desired.

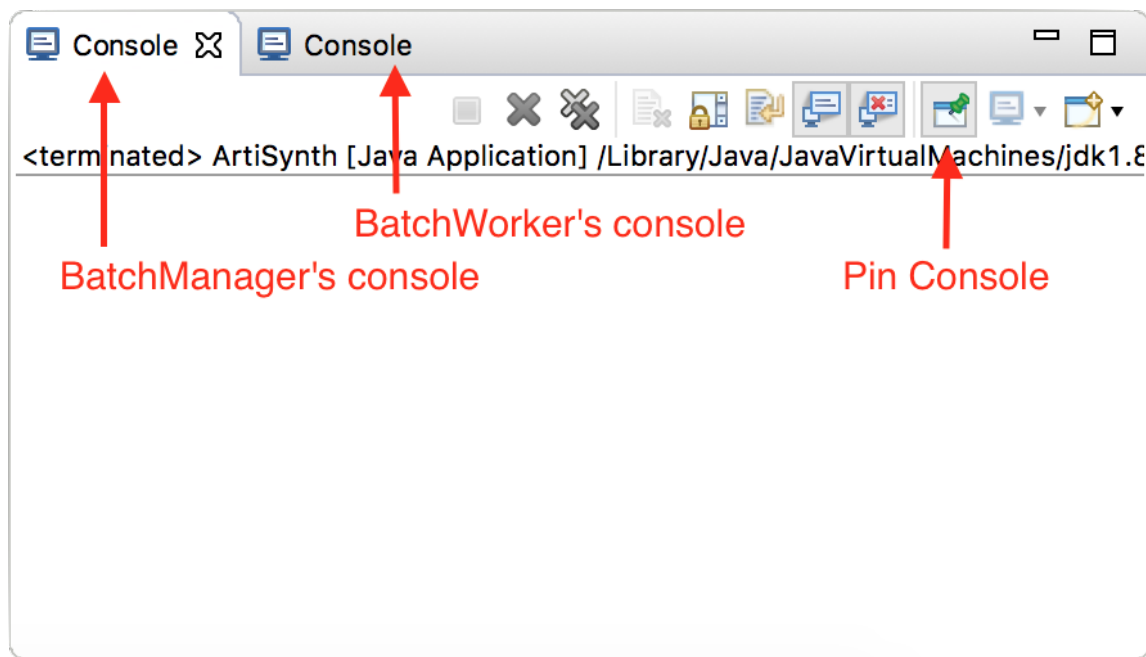


Figure 2: Pinning a Console View in Eclipse.

Note: running more than one `BatchWorker` from within Eclipse is not recommended, because Eclipse does not differentiate between instances of the same program when it comes to pinning program outputs in the Console Views. As such, the output from all but one `BatchWorker` will not be visible in the Console Views (the program **will** run, but not in an accessible fashion).

Note: `BatchSim` can also run using Eclipse and the command line at the same time. For example, the `BatchManager` can run in Eclipse, while the n `BatchWorkers` can run from the command line.

4.5 Advanced BatchSim options

The preceding examples (Sections 4.3 and 4.4) for running `BatchSim` present only the simplest and most common use cases. They allow `BatchSim` to run in its default mode. Additional modes and behaviours can be specified by appropriate command line arguments/options.

Note: running their respective program with the `-help` option prints a summary of available options for both the `BatchManager` and `BatchWorkers`.

Note: for additional details, consult the Javadocs of [BatchWorkerBase](#) and [BatchManager](#).

4.5.1 Options to the `BatchManager`

At the time of writing, the options for the `BatchManager` are

`-h, -help`

Prints a help message and exits.

`-f, -file <FILENAME>`

Reads the input configurations from file `FILENAME`, which defaults to `props.psl` in the current working directory. If `FILENAME` is `-`, reads from standard input (which forces an interaction level of 0; see `-i` option).

`-m, -monteCarlo <N>`

If the input file (see `-f` option) contains **only** probabilistic specifications, performs `N` (which defaults to 1) [Monte Carlo](#) (i.e. random/probabilistic) simulations (see Sections [5.5](#) and [5.7.3](#)). **Otherwise, this option is ignored.**

`-p, -port <PORT>`

Binds the port to which the `BatchManager` socket is listening for requests from a `BatchWorker` to `PORT`, which defaults to 34365.

`-i, -interactionLevel <LEVEL>`

Sets the level of interaction with the user to `LEVEL`, which defaults to 2 (unless `-f -` is given; see `-f` option). A `BatchManager` can run in “interactive mode” by initiating an interactive `ArtiSynthJythonConsole` session with the user. There are 3 levels of interaction. Level 0 means the `BatchManager` outputs nothing except error messages. Level 1 means the `BatchManager` outputs error and warning messages. Level 2 means fully interactive. In the latter mode, several builtin commands allow the user to interact with the `BatchManager`. These are

```
commands() -- output this list of commands
progress() -- output progress information
quit()      -- exit with status 0
quit(st)    -- exit with status 'st'
```

Note: Low interactivity of the `BatchManager` is usually coupled to using the `-noGui` option to `ArtiSynth` when running `BatchWorkers` (see Section [4.3.2](#)).

`-v, -debug`

Prints debugging (i.e. verbose) messages. This is independent of the interaction level (see `-i` option).

`-r, -dryRun`

Reads the input file (see `-f` option) and exits. This ensures the input file can be parsed without errors.

`-b, -bufferCap <CAP>`

Sets the maximum capacity of the buffer used to store created but unsent simulation tasks to `CAP`. This is essentially the maximum size of the `BatchManager`’s “bag of tasks.” `CAP` defaults to 2^{20} (just over 1 million), but can be set to a larger value if more than that many simulations tasks will be created and the user is interested in knowing (before running all of them) the exact number of simulations to be run. On the other hand, it can be set to a smaller value if so many simulation tasks are being created that the JVM runs out of memory and crashes.

`-c, -checkProps`

If the `BatchManager` is running within an instance of `ArtiSynth`, verifies that the **currently-loaded** target model has all the `Properties` listed in the input file (see `-f` option), and that, for each such `Property`, all the listed values are assignable to that `Property` (see Section [4.3.1](#)).

`-d, -delimiter <CHAR>`

Sets the value delimiter character (see Section [5.7.2](#)) of the input file (see `-f` option) to `CHAR`, which defaults to `%`. Note that some characters are not recommended (high risk) and others are completely disallowed as the delimiter character (see `-o` option; see Section [5.7.2](#)).

`-o, -riskyDelimOk`

Allows the use of delimiter characters (see `-d` option) that are otherwise deemed too risky due to having syntactic significance (see Section [5.7.2](#)). Using risky delimiters may require escape characters (see Section [5.7.2](#)). At the time of writing, the illegal characters are: `{`, `}`, `[`, `]`, `#`, `(`, `)`, and all digit and whitespace characters. At the time of writing, the risky characters are: `=`, `~`, `"`, `'`, `@`, and `$`.

`-e, -epsilon <EPS>`

Sets the epsilon for comparing two doubles for equality to `EPS`, which defaults to 0.0001.

`-s, -seed <SEED>`

Sets the seed for the random number generator to `SEED`, or use no seed at all if `SEED` is `-1`. `SEED` defaults to `-1`.

Note: for truly repeatable results, setting this random seed is not enough. In addition, the `-disableHybridSolves` option to ArtiSynth must be used when running `BatchWorkers` (see Section 4.3.2).

4.5.2 Options to `BatchWorkers`

At the time of writing, the options for each `BatchWorker` are

`-h, -help`

Prints a help message.

`-n, -workerName <NAME>`

Sets the `BatchWorker`'s name to `NAME`, which defaults to `worker<PORT>`, where `PORT` is supplied by the `-p` option (or its default runtime value; see `-p` option). Setting a unique name for each `BatchWorker` is important, as it creates a one-to-one correspondence between each `BatchWorker` and its output files, which prevents data corruption and facilitates post-processing (see `-d`, `-l`, and `-f` options; see Section 7.2).

`-d, -outputDirName <DIRNAME>`

Sets the `BatchWorker`'s output directory to `DIRNAME`, which defaults to the current working directory. Having unique `BatchWorker` names (see `-n` option) allows each `BatchWorker` to have its own (uniquely named) output file (see `-l` and `-f` options), and yet for all `BatchWorkers` to share a single output directory. Having separate output files is critical to avoid data corruption, and sharing an output directory makes post-processing easier (see Section 7.2).

`-e, -rerunSims <LIST>`

If a simulation fails, reruns that simulation using the time steps listed in `LIST` until success is achieved. `LIST` should be of the form `<double>[[, ...], <double>]` (a comma-separated list of doubles given as a single string). `LIST` defaults to the empty list, meaning simulations are not rerun. Certain models, especially those containing finite-element models, are quite unstable and tend to cause simulation crashes. In this case, rerunning the simulation with one-tenth of the original time step can often increase stability enough to resolve the crash. However, running every simulation at that time step is not ideal, since smaller time steps require significantly more computation time. Hence the existence of this option. Since rerunning at one-tenth of the original time step is not always desired, providing a list of arbitrary doubles renders this options far more general.

`-l, -logFileName <FILENAME>`

Sets the `BatchWorker`'s log file name to `FILENAME`, which defaults to `<worker_name>_log.txt`, where `worker_name` is supplied by the `-n` option. The log file is intended to store per-simulation logging information. Making use of this file is at the user's discretion (see `addLogEntry()` in Section 6.2).

`-f, -propValFileName <FILENAME>`

Sets the `BatchWorker`'s property-value file name to `FILENAME`, which defaults to `<worker_name>_propVal.txt`, where `worker_name` is supplied by the `-n` option. The property-value file is intended to store the input configuration (i.e. the value of each `Property` in the simulation task; see Section 5.3) used for each simulation run by that `BatchWorker`. Making use of this file is at the user's discretion (see Section 6.2).

`-p, workerReplyPort <PORT>`

Sets the `BatchWorker`'s **base** reply port to `PORT`, which defaults to 23159. Note that the actual port used may be different if the port `PORT` is already in use. In this case, the `BatchWorker` will try binding each successive port (incrementing by one) until a port is successfully bound. This incrementing feature is extremely useful in practice, since starting n `BatchWorkers` will automatically cause n successive ports to be bound, without having to explicitly pass a different port number to each `BatchWorker`. In addition, since `BatchWorker` names and log files (see `-n`, `-l`, and `-f` options) are linked to the port number (by default), this feature ensures unique file names for each `BatchWorker` by default (see `-d` option).

`-m, managerHostName <HOSTNAME>`

Informs the `BatchWorker` that the `BatchManager`'s host machine's name is `HOSTNAME`, which defaults to the `BatchWorker`'s localhost name or loopback address name (by default, `BatchSim` assumes the network connection between the `BatchManager` and the `BatchWorkers` is local).

```
-r,managerRequestPort <PORT>
```

Informs the `BatchWorker` that the `BatchManager` is listening for requests on port number `PORT`, which defaults to 34365.

Note: custom `BatchWorkers` can be given additional options (see `otherParserOptions()` in Section 6.2).

5 The Property Specification Language

Whereas `BatchWorkers` are specifically designed to be customizable, the `BatchManager` is a complete program that is not intended to be modified by the user. Instead, the `BatchManager`'s behaviour is influenced through the contents of the input file it receives from the user (see the `-f` option of the `BatchManager` in Section 4.5.1). This file contains the input configuration of every simulation that `BatchSim` should run for the current batch. These configurations are provided in a compact form in which the user specifies the values or probability distributions of various `Properties` of a target model, which are known as property specifications. Property specifications are written in a very simple programming language called the Property Specification Language (PSL). The input file simply contains a list of property specifications.

5.1 The target model and `Properties`

`BatchSim` is model-agnostic in the sense that it can run simulations for any `RootModel` subclass, which `BatchSim` refers to as the **target model**. Like much of the `ArtiSynth` and `Maspack` codebases, this agnosticism is achieved by accessing model attributes through `ArtiSynth`'s `Property` mechanism, which is very general and widely used. Frequently, the input parameters that are to vary with each simulation are already `Properties`, either of the target model itself, or else of one of its subcomponents (e.g. a finite-element model). Whenever an input parameter is not already a `Property` of some component in the target model's component hierarchy, a `Property` can readily be defined to “wrap” said input parameter so as to become a `Property`. Refer to the [Maspack Reference Manual](#) for details on adding custom `Properties` to any class, including a `RootModel` subclass.

5.2 Property paths

Throughout this document, the terms “property”, “`Property`”, and “property path” are used interchangeably and refer to both `Properties` and property paths. Outside this document, a property path is simply a path from the target model to one of its subcomponents, followed by `:` (a colon), followed by the name of a `Property` of that component. For `CompositeProperties`, sub-`Properties` can be referenced by separating the name of the `CompositeProperty` and its sub-`Properties` with `.` (a period). Refer to the [Maspack Reference Manual](#) for details on property paths.

5.3 Simulation tasks and property-value pairs

The `BatchManager` parses the input file and collects all property specifications listed therein. It then processes these property specifications in one of three ways (see Sections 5.4, 5.5, 5.6) to create simulation tasks. Each simulation task contains a list of `Properties` and, for each, a particular value to which it should be set for that particular simulation. These are known as property-value pairs. Each value is a textual (i.e. string) representation of a value that can be assigned to the associated `Property`. **The format of this string must be such that the `PropertyInfo.scanValue(ReaderTokenizer)` method can scan it.**

There are two kinds of property specifications: combinatorial and probabilistic. As part of the property specification, the user either hardcodes the `Property` values directly to create a combinatorial specification (see Section 5.7.2), or else indicates the probability distribution from which to draw the values for the `Property` to create a probabilistic specification (see Section 5.7.3).

5.4 Combinatorial property specifications

Here, a particular `Property` is associated with a *set* of values, and each value is assigned to the `Property` in turn during successive simulations. Other `Properties` are associated with different value sets in their own combinatorial specification. The `BatchManager` iterates through the values in all combinatorial specifications so as to perform an exhaustive/combinatorial search of the input space: each possible combination of property-value pairs for all combinatorial specifications is examined. Each simulation task consists of exactly one such combination. More formally, say there are N such combinatorial specifications. Then, the general combinatorial algorithm is

```
for value1 in specification1.valueSet()
  for value2 in specification2.valueSet()
    // ...
    for valueN in specificationN.valueSet()
      createSimulationTask(value1, value2, ..., valueN)
    endFor
  // ...
endFor
endFor
```

Note: the actual algorithm is recursive (where `// ...` represents the recursion), but it is easier to understand this “unfolded” version.

Note: mathematically speaking, one simulation task is performed for each element in the N -ary [Cartesian product](#) of all the value sets.

5.5 Probabilistic property specifications

Here, a particular **numeric** `Property` is associated with a *vector* of parametric probability distributions, with the size of the probability distribution vector equal to the size of the `Property` vector. For a scalar numeric `Property`, the probability distribution vector should contain a single entry. Other `Properties` are associated with different probabilistic distribution vectors in their own probabilistic specification. If **only** probabilistic specifications are given, the manager creates simulation tasks by applying the [Monte Carlo](#) method. That is, for each probabilistic specification, all the distributions listed in its associated probability distribution vector are sampled, and a random-value vector is formed from the sampled values. These sampled vectors are then assembled into one simulation task (thus forming one Monte Carlo simulation task), and the process is repeated M times, where M is specified by the user (see the `-m` option of the `BatchManager` in [Section 4.5.1](#)). Formally, the algorithm is

```
for i in 1 to M
  vectors = new List()
  for specification in specificationsList
    sampledVector = new Vector()
    for distribution in specification.distributionVector()
      sampledVector.add(distribution.sample())
    endFor
    vectors.add(sampledVector)
  endFor
  createSimulationTask(vectors)
endFor
```

5.6 Running hybrid combinatorial/probabilistic simulations

If *at least one* probabilistic specification and *at least one* combinatorial specification are given, the combinatorial algorithm described above is used, but augmented with a check for the specification’s type. If the specification is probabilistic, it is treated as if its distribution vector was a single combinatorial value (whose value just happens to be different with each combination). The end result is a hybrid algorithm between combinatorial and Monte Carlo

```

iterator = specificationsList.iterator()
while iterator.hasNext()
  specification1 = iter.next()
  if specification1.type() is COMBINATORIAL
    for value in specification1.valueSet()
      specification2 = iterator.next()
      // ...
      createSimulationTask(value, ...)
    // ...
  endFor
else // specification1.type() == PROBABILISTIC
  sampledVector = new Vector()
  for distribution in specification1.distributionVector()
    sampledVector.add(distribution.sample())
  endFor
  specification2 = iterator.next()
  // ...
  createSimulationTask(sampledVector, ...)
  // ...
endif
endWhile

```

Note: again, the actual algorithm is recursive (where `// ...` represents the recursion), but it is *much* easier to understand this “unfolded” version.

5.7 Property specifications

5.7.1 Property path syntax

Whether combinatorial or probabilistic, each property specification begins with a double-quoted property path

```
"<property_path>"
```

where `<property_path>` must be a valid ArtiSynth property path (see Section 5.2), except for an optional component identifier set extension provided by BatchSim that allows multiple property paths to be grouped into a single specification (see Section 5.7.5).

5.7.2 Combinatorial specification syntax

Each combinatorial specification must adhere to the following general format

```
"<property_path>" = <value_set>
```

where `<value_set>` must be of the form

```
{ <delim><val1><delim> <delim><val2><delim> ... <delim><valN><delim> }
```

where `<delim>` is a chosen value delimiter character (`%` by default; see the `-d` option of the BatchManager in Section 4.5.1). Each `<valX>` consists of the entire string between two consecutive occurrences of `<delim>` (including all whitespace (except the end-of-line character); see Section 5.7.4), and is taken to be a string representation of a (valid) value for the associated Property.

Some examples of valid combinatorial property specifications (when the delimiter character is the default `%`) are

```

"models/0/component/1:someProp.someSubProp" = {%1% %2% %3%}
"color" = {% "red"% % "green"% }
"someVectorProp" = {% [1 2 3]% % [4 5 6]% }

```

Note: the preceding example involving the `color` property path illustrates why the default delimiter character is `%` instead of a more traditional character such as `"` (double quote): if a `Property`'s type is `String`, the `ReaderTokenizer` input to the `PropertyInfo.scanValue(ReaderTokenizer)` method **must** be fed a double-quoted string, which means the values of this property specification would have had to be coded as `"red"` and `"green"` (if `"` was the delimiter character). Clearly, this will not be parsed correctly. Similar arguments apply to using `'` or `,` as the default delimiter character.

It *is* possible to set the delimiter character to `"`, but in this case the double quotes belonging in the value need to be escaped: `"red"` and `"green"`. This also requires passing the `-o` option to the `BatchManager` (see Section 4.5.1).

5.7.3 Probabilistic specification syntax

Each probabilistic specification must adhere to the following general format

```
"<property_path>" ~ <distribution_vector>
```

where `<distribution_vector>` must be of the form

```
[ <distribution1> <distribution2> ... <distributionN> ]
```

where the number of distributions listed between the pair of square brackets must match the size of the associated `Property`'s data type. That is, for scalar-valued `Properties`, a single distribution should be listed, whereas for vector-valued `Properties`, the number of distributions listed should match the vector's size. Each `<distributionX>` must be of the form

```
<Name_of_distribution>(<parameter1_value>, ..., <parameterN_value>)
```

where `<Name_of_distribution>` is the *non-quoted* string name of a probability distribution. To be valid, this name must match `Distribution.toString()` for some `Distribution` value. Following the distribution name, a *comma-separated* list of parameter values (integers or doubles, depending on the distribution) enclosed in one pair of parentheses should appear. The values must appear in the same order and be in the same quantity as specified by the corresponding `Distribution`, and each value must be a valid number for that parameter of that `Distribution` (note that integers can be coerced into doubles).

Some examples of valid probabilistic property specifications are

```
"models/0/component/1:someScalarProp" ~ [ Normal(20.0, 1.5) ]
"colorRGB" ~ [ Uniform(0, 1) Uniform(0, 1) Uniform(0, 1) ]
"someVectorPropWithConstant2ndEntry" ~ [
    T(2)
    Uniform(1, 1)
    Normal(0, 1)
]
```

Aside: in probabilistic specifications, `~` is borrowed from probability theory notation: if a random variable X is distributed according to probability distribution D , then we write $X \sim D$. Here, we augment this notation by using \mathbf{X} (the `Property`) to represent a vector of random variables, and we specify one probability distribution, D_i , for each entry i of \mathbf{X} , notated as $\mathbf{X} \sim [D_1 D_2 \dots D_n]$.

5.7.4 General syntax

Some notes on the syntax (in terms of `tokens`) of property specifications follow. Unless otherwise indicated, syntax rules apply to both combinatorial and probabilistic property specifications.

1. The `#` token begins a comment. All characters between it and the next end-of-line character are ignored.
2. The `<property_path>` token must be double-quoted.

3. All tokens in any specification can be separated by an arbitrary number of whitespace characters. Thus, the following examples are all syntactically valid and semantically equivalent combinatorial property specifications

```
"models:prop" = { %val1% %val2% }
"models:prop" = {%val1% %val2%}
"models:prop"={%val1%%val2%}
"models:prop" = {
    %val1%
    %val2%
}
```

4. The value set of a combinatorial specification *can* be empty: {}. In this case, *but only in this case*, the associated Property will be assigned the value null in *all* simulation tasks.
5. Whitespace appearing in any double-quoted string will be considered part of the string token, **unless** it is an end-of-line character, which will cause the string to be split into multiple word tokens. This applies to both <property_path> and <valX> tokens, and is caused by the [ReaderTokenizer](#) used to parse the input file. One exception to this rule is that non-end-of-line whitespace appearing *within a component identifier set* of a <property_path> token is ignored (see Section 5.7.5).

5.7.5 Component identifier sets

Within a <property_path>, a special set notation can be used as a shorthand to group multiple property paths into a single specification. This is useful when the multiple property paths would otherwise share the same value set (for combinatorial specifications) or distribution vector (for probabilistic specifications).

A target model has components, and all components have either a name or number (called a component identifier).

Within a <property_path>, instead of specifying a single component identifier at each level in the component hierarchy, a *set* of identifiers can be specified (within a pair of curly braces). This set will be “expanded” in place by creating one (implicit) specification for each component identifier in the set. For example, the <property_path>

```
"models/{mySubComponent1 mySubComponent2}:prop"
```

will be expanded into

```
"models/mySubComponent1:prop"
"models/mySubComponent2:prop"
```

This expansion works regardless of the property specification type: the value set (for combinatorial specifications) or distribution vector (for probabilistic specifications) is copied (re-instantiated) for each expanded <property_path>.

Note: the individual identifiers in a component identifier set must be separated by an arbitrary number of non-end-of-line whitespace characters.

Note: since a component identifier set appears within a <property_path>, which is itself quoted by ", the set cannot contain " characters in it.

Note: since components cannot be identified without an identifier, a component identifier set cannot be empty.

When the component is identified by a number, that number is an integer. As such, an *inclusive* range of such integers can be provided as yet another shorthand. For example, the <property_path>

```
"models/{[0-2] [24-25]}:prop"
```

will be expanded into


```
"models/0:prop"
"models/1:prop"
"models/2:prop"
"models/24:prop"
"models/25:prop"
```

Note: an arbitrary number of whitespace characters can separate the individual characters within the range.

Note: the first integer specified in a range must be less than or equal to the second integer in the range, but separate ranges need not be listed in increasing order.

The expansion of a component identifier is done in place. That is, the substring delimited by the curly braces will be exactly replaced by each identifier listed in the set. For example, the `<property_path>`

```
"models/comp{ [0-2] [24-25] }onent:prop"
```

will be expanded into

```
"models/comp0onent:prop"
"models/comp1onent:prop"
"models/comp2onent:prop"
"models/comp24onent:prop"
"models/comp25onent:prop"
```

and the `<property_path>`

```
"models/{ left right top }_comp:prop"
```

will be expanded into

```
"models/left_comp:prop"
"models/right_comp:prop"
"models/top_comp:prop"
```

The identifiers in a component identifier set can in fact be component paths or path fragments as well. For example, the `<property_path>`

```
"models/{ comp0/1 comp2/3 }:prop"
```

will be expanded into

```
"models/comp0/1:prop"
"models/comp2/3:prop"
```

Finally, multiple *sequential* component identifier sets can be specified within a single `<property_path>`, and these will be expanded in a combinatorial fashion. For example, the `<property_path>`

```
"models/comp{0 1}/{2 3}:prop"
```

will be expanded into

```
"models/comp0/2:prop"
"models/comp0/3:prop"
"models/comp1/2:prop"
"models/comp1/3:prop"
```

However, at the time of writing, BatchSim does **not** support nested component identifier sets. For example, the `<property_path>`

```
"models/comp{[0-1] {3 7 9}}:prop"
```

is *not* valid and will not be expanded.

5.7.6 Decorators

Sometimes, it may be useful to specify a property specification as probabilistic, but then pre-sample from its distribution vector a certain number of times, say k , and treat these k values as though they were values in a combinatorial value set. In other words, the property specification is specified as probabilistic, but we sample from it k times, and then treat the resulting values as the k values of the value set of a combinatorial property specification.

Similarly, we may want to specify a property specification as combinatorial, but instead of iterating deterministically through its value set to create simulation tasks, we may want to randomly sample from this set. Furthermore, we may want each value to have a different probability of being sampled. In other words, the property specification is specified as combinatorial, but we treat its value set as the support of a [discrete probability distribution](#), and then sample from this distribution as if the property specification was probabilistic.

BatchSim supports both of these features through [decorators](#). As the name implies, these objects “decorate” a property specification to modify its interpretation. The syntax is as follows.

For combinatorially-decorated probabilistic specifications

```
@COMB(k) <probabilistic_property_specification>
```

leads to sampling the distribution vector of the specification k times.

Note: to have repeatable results, consider fixing the seed of the BatchManager’s random number generator (see `-s` option to the BatchManager in Section 4.5.1).

For probabilistically-decorated combinatorial specifications

```
@PROB <combinatorial_property_specification>
```

leads to uniform sampling of the value set, while

```
@PROB(p1, p2, ..., pN) <combinatorial_property_specification>
```

leads to sampling with value-specific probabilities. Here, the usual constraints on discrete probability distributions, namely $\sum p_i = 1$ and $p_i \geq 0$, $i = 1, 2, \dots, N$, must hold.

Note: the first form is just a shorthand for the second form, with $p_i = \frac{1}{N}$, $i = 1, 2, \dots, N$.

Note: the number of values in the value set **must** match the number of arguments to the @PROB decorator.

Note: at the time of writing, BatchSim does not support the addition of more than one @PROB or @COMB decorator to a single property specification.

Sometimes it may be useful to have a property specification that exists solely for triggering control structures (see Section 5.8), rather than as a simulation input. But property specifications that don’t correspond to a real Property in the target model result in the BatchWorkers throwing a runtime error. The only way around this would be to add a new “phony” Property to the target model. This is obviously not very elegant and needlessly clutters the target model’s code. Instead, a third kind of decorator exists, the @PHONY decorator, which simply flags the property specification as phony. The BatchManager never passes any phony Properties to the BatchWorkers, which resolves the runtime error problem, but phony property specifications can still be used in the input file to trigger control structures.

Note: any property specification (whether otherwise decorated or not) can be **prepended** by *at most one* @PHONY decorator.

5.8 Control structures

At the time of writing, property specifications can be affected by three control structures: `skip` statements, `redef` (i.e. redefinition) statements, and Jython code blocks. These structures are useful for removing certain combinations of input values that would otherwise be discarded as invalid simulations in post-processing. The power of using these structures comes from the fact that they save valuable computation time by not running simulations that will end up being discarded anyways. In addition, they simplify post-processing, because discarding specific pieces of data later on may be non-trivial, and may require manual intervention (although this depends on how the output data is recorded).

Note: because they can interact in complex and very subtle ways, very little checking is done to ensure valid use of these control structures prior to running simulations. For example, multiple `redef` statements can “cascade,” where one is triggered, which redefines some other `Property`, which later triggers yet another. Although this is valid, and even encouraged in many cases, no checks are made to ensure nothing is done incorrectly. Instead, task creation proceeds assuming everything will work out, and only if an error is detected during task creation will a meaningful error message be given.

5.8.1 `skip` statements

A `skip` statement is a list of *combinatorial* property specifications, each of which has to have a property path that has already been defined at some earlier point in the input file (possibly with a different value set; the “identity” of property specifications is tied to the property path). A `skip` statement causes simulations to be skipped. Specifically, it skips all simulations where each `Property` listed in the `skip` statement has a current value equal to any one of the values listed in its value set. That is, multiple property specifications listed together in the same `skip` statement are logically ANDed, and, within each property specification, the values in its value set are logically ORed. This is called a `skip` match. The syntax of a `skip` statement is the keyword `skip` followed by a list of **undecorated, combinatorial** property specifications (called the `skip` block), followed by the keyword `end`. For example

```
"prop1" = {%0% %1% %2%}
"prop2" = {%3% %4% %5%}
skip
  "prop1" = {%1% %2%}
  "prop2" = {%3% %4%}
end
```

will skip all simulations where (`prop1` = 1 OR `prop1` = 2) AND (`prop2` = 3 OR `prop2` = 4), but allow all others (i.e. the pairs (0, 3), (0, 4), (0, 5), (1, 5), and (2, 5) are all allowed).

5.8.2 `redef` statements

`redef` statements are composed of two blocks. The syntax is the keyword `redef` followed by a list of property specifications, which may be combinatorial or probabilistic, and decorated or not (called the `redef` block), followed by the keyword `when`, followed by a list of **undecorated, combinatorial** property specifications (called the `when` block), followed by the keyword `end`.

The `when` block is similar to the `skip` block of the `skip` statement, except that instead of skipping simulation tasks that match the block, it triggers the redefinition of the property specification(s) in the `redef` block. “Redefinition” means that the property path stays the same, and so refers to the same `Property` of the target model, but some other part of the property specification changes. The change can be anything: adding a decorator, changing the values in the value set, changing the probability distributions in a distribution vector, changing the size of the value set or distribution vector, changing the property specification from combinatorial to probabilistic, or anything else.

A list of these redefinable property specifications should appear in the `redef` block. This list is just a shorthand: implicitly, there is one separate and completely independent redefinition for each property specification listed in the `redef` block. As such, it would be equivalent to having one redefinition with the same `when` block for each property specification listed in the `redef` block. One catch: **every** property specification listed in the `redef` block must be initially defined (in the input file) **after** every specification listed in the `when` block.

Aside: we can think of the contents of the input file as a directed graph where each property specification is a node, and where there is a *directed* edge from one property specification to another if the former appears immediately before the latter in this file. Then, a topological sort of the resulting directed acyclic graph (DAG) would place the property specifications in the same order as they appear in the input file. With this in mind, we can think of a `redef` statement as introducing several more directed edges to the DAG. Each such edge starts at a property specification in the `when` block and ends at a property specification in the associated `redef` block. The `redef` statement, then, is valid if and only if the graph that results from the introduction of these new edges is still a DAG (i.e. has no cycles). This is done to prevent cyclical redefinitions, which are usually hard to get right, and often cause infinite loops unless extreme care is taken to avoid them.

`redef` statements become extremely useful when combined with phony property specifications acting as flags. For example

```
@PHONY "prop1" = {%LOW% %HIGH%} # The flag is defined before the "real" properties.
"prop{2 3}" = {} # prop2 & prop3 are initially set to null (to be redefined later).
redef
  "prop2" = {%1% %2%}
  "prop3" = {%value_for_low%}
when
  "prop1" = {%LOW%}
end
redef
  "prop2" = {%9% %10%}
  "prop3" = {%value_for_high%}
when
  "prop1" = {%HIGH%}
end
```

Here, `prop1` acts as a flag which makes some other properties, namely `prop2` and `prop3`, have values that are either low or high in tandem. This effectively skips simulations where one has a low value while the other has a high value, which (in this example) we would want to discard as invalid. Notice that `prop1` is defined before `prop2` and `prop3`. This is to preserve the valid ordering of property specifications, as mentioned above.

Note: since the effect of `skip` and `redef` statements is the same (discarding unwanted simulations), these two structures are effectively equivalent. Indeed, they simply represent different ways of expressing some discard behaviour/logic, and one can always be defined in terms of the other. However, since the logic may be much more complicated to express with one compared to the other, both are made available for convenience.

5.8.3 Jython code blocks

Jython code blocks are different from `skip` and `redef` statements in that they are not standalone statements. Instead, they must occur within a `skip` or `when` block. They act as an additional way in which `skip` or `when` matches can be detected. This can be useful, for example, to match conditional on the value of a probabilistic property specification, which are not allowed in `skip` and `when` blocks. In fact, Jython code blocks can entirely replace the combinatorial matching described above. The advantage of the combinatorial matching, however, is in terms of (1) a convenient syntax, and (2) **noticeably faster computation time** on the part of the `BatchManager` (the `BatchWorkers` are unaffected).

Note: the computational slowdown comes from the fact that, rather than executing everything in Java, the `BatchManager` has to load the Jython code into a Jython interpreter and then relinquish control to the interpreter, which runs the Jython code before eventually returning control to the `BatchManager`.

Although the computational slowdown is noticeable, it is *rarely* significant enough to cause problems. This is because the `BatchManager` creates simulation tasks (to fill its bag of tasks) in a separate thread than the one listening for task requests from the `BatchWorkers`. A slowdown will therefore only affect the overall speed of `BatchSim` if simulation runtimes are faster than task creation times. This is very unlikely, unless the target model is extremely simple and there are a lot of Jython code blocks in the input file.

The syntax of a Jython code block is the keyword `jython` followed by a list of arbitrary lines of Jython code, each of which must be prepended by a `$`, followed by the keyword `end`. Within the Jython code, three special functions are exposed.

`get(string)`

Takes a string representing a property path, and returns the current value of the property specification with that property path (as a string), or `None` if the value is not set.

The value may not be set for one of three reasons.

1. The given property path string does not correspond to a previously defined property specification (e.g. because of a typo). This can happen in both `skip` and `when` blocks.
2. The property specification was previously defined, but the value has not yet been set. This can happen in both `skip` and `when` blocks, and occurs because the recursive algorithms from Sections 5.4 and 5.6 have to run each Jython code block at the beginning of each recursive call in order to ensure the proper functioning of `skip` and `redef` statement logic. In the middle of the recursion, before reaching the base case, only some of the property specifications will be assigned a value.
3. The property specification was previously defined, but the valid ordering of property specifications is not being respected (see Section 5.8.2). This can happen in only in `when` blocks.

`get_if_valid(string)`

Like `get()`, but throws an exception instead of returning `None` if the value is not set. This is useful to prevent silent bugs from occurring in code if the return value is not checked for equality to `None`.

The reason for having both `get()` and `get_if_valid()` is because of the second reason that `get()` can return `None`: a user may want to check for a “partial match” halfway through the recursion without having an exception thrown, and so needs to use `get()` and explicitly check for `None`, rather than use `get_if_valid()`. This is an advantage over using the basic `redef` statement, which always disallows out-of-order property specifications logic in order to provide an “accelerated” (see below) special case.

`return_value(boolean)`

Takes a boolean and returns nothing. `return_value()` is a callback that passes the given value back to the `BatchManager`. If the Jython code block should trigger a match (either in a `skip` or `when` block), then `return_value(True)` should be called. Otherwise, `return_value(False)` should be called.

Note: calling `return_value()` multiple times within a single Jython code block is not permitted.

Note: if a `skip` or `when` block contains both Jython code blocks and combinatorial property specifications, then all of them must match for an overall match to be established.

As an example of using Jython code blocks, consider

```
"prop1" = {%0% %1% %2%}
"prop2" = {%3% %4% %5%}
skip
  jython
    $prop1_val = get("prop1")
    $return_value(prop1_val in ["1", "2"])
  end
  "prop2" = {%3% %4%}
end
```

This is identical to the example `skip` statement shown in Section 5.8.1, **except that it will be computationally slower.**

Note: Although it may not be apparent in this simple example, the (often slight) computation slowdown of the Jython code blocks may well be worth it, since, in more complicated use cases, using only the basic functionality of `skip` or `redef` statements may be extremely cumbersome/bug-prone. In general, the basic `skip` and `redef` statements should be considered “accelerated” special cases provided by `BatchSim`.

In addition to the above Jython code block syntax, Jython one-liners can be specified by appending an additional `$` to the end of the one line of Jython code (the `end` keyword must still be present). For example, the following is identical to the previous example

```
"prop1" = {%0% %1% %2%}
"prop2" = {%3% %4% %5%}
skip
    jython $return_value(get("prop1") in ["1", "2"])$ end
"prop2" = {%3% %4%}
end
```

6 Customizing BatchWorkers

Whereas the `BatchManager`'s behaviour is modified using the input file, the `BatchWorkers` are designed to be customized by subclassing `BatchWorkerBase`.

6.1 What BatchWorkers do

`BatchWorkerBase` is an abstract class that does most of the heavy-lifting required of a `BatchWorker`. Specifically, `BatchWorkerBase` handles the communication between a `BatchWorker` client and a `BatchManager` server. It also handles, upon receiving a simulation task from the `BatchManager`, setting the new value of the appropriate `Properties` of the target model. Finally, it takes care of running all simulations, including potential reruns (see `-e` option the `BatchWorker` in Section 4.5.2).

The abstract `BatchWorkerBase` class uses the [template method pattern](#), which means it leaves “holes” in its methods by calling abstract sub-methods. Concrete (i.e. non-abstract) subclasses should override these sub-methods to add custom behaviour to the `BatchWorker`. The `run()` method is where most of these abstract methods get called. Since the `run()` method is quite complex, a pseudo-code of the implementation is provided here, so that the order in which these abstract methods get called, and what happens between those method calls, is made very clear.

```
run() :
    startLogSession() // For a subclass to override.
    setUpStopConditionMonitor() // For a subclass to override.
    while requestSimTask() != NULL: // While there are simulation tasks left.
        success = false // Set to true once a simulation succeeds (no error occurred).
        myNumSimsAttempted = 0 // For a subclass to know how many reruns happened.
        timeStepsToTry = List(targetModel.getMaxStepSize()) // All the potential ...
        timeStepsToTry.addAll(myRerunList) // ... time steps to try before giving up.
        while !success && !timeStepsToTry.isEmpty():
            myLastStepSizeUsed = timeStepsToTry.removeFirst() // For a subclass to know
            myNumSimsAttempted++ // For a subclass to know.
            reset() // Reset the target model from the previous simulation (if any).
            setMaxStep(myLastStepSizeUsed) // Set the step size for this simulation.
            setPropVals() // Set the property values from the current simulation task.
            preSim() // For a subclass to override.
            play() // Run the simulation.
            waitForStop() // Block the thread, waiting for the simulation to stop.
            postSim() // For a subclass to override.
            if myStopConditionMonitor.hasAnyConditionBeenMet():
                success = true // Simulation stopped appropriately. No error occurred.
            endif
        endwhile // Attempted all potential reruns or some simulation succeeded.
        myCurrentTaskSuccessful = success // For a subclass to know.
        recordSimResults() // For a subclass to override.
        addLogEntry() // For a subclass to override.
    endwhile // Simulation bag of tasks is empty. Nothing left to do.
endMethod
```

Note: since these methods are declared abstract in the abstract `BatchWorkerBase` class, a concrete subclass must provide an implementation of these methods (see Section 6.2). However, most of these methods are not necessary for the proper functioning of the `BatchWorker`, and so can be implemented with an empty body. The **only** `BatchWorkerBase` abstract method that **must** have a non-empty body in order for the `BatchWorker` to function properly is `setUpStopConditionMonitor()` (see Section 6.3).

6.2 Subclassing `BatchWorkerBase`

A number of methods are declared abstract in the abstract `BatchWorkerBase` class, and a concrete subclass must provide an implementation of these methods. This is how customizing `BatchWorkers` is achieved. However, most of these methods are not necessary for the proper functioning of the `BatchWorker`, and so can be implemented with an empty body. For example, `addLogEntry()` can be implemented like this:

```
class MyConcreteBatchWorker extends BatchWorkerBase {
    // ...
    @Override
    public void addLogEntry() {
        // Empty method.
    }
    // ...
}
```

Note: the **only** `BatchWorkerBase` abstract method that **must** have a non-empty body in order for the `BatchWorker` to function properly is `setUpStopConditionMonitor()` (see Section 6.3).

The abstract methods to be overridden are (in alphabetical order)

`void addLogEntry()`

Called in the `run()` method.

At the very end of each simulation task, this method is called to allow for recording of logging/meta-information relating to the current simulation task in the log file (using the `myLogFileWriter`, `myLogComment`, and `mySeparator` instance variables).

Subclasses can optionally override this method to add custom data logging (no logging is otherwise performed). In particular, subclasses are encouraged to make heavy use of the `myCurrentTask`, `myTaskCounter`, `myCurrentTaskSuccessful`, `myLastStepSizeUsed`, and `myNumSimsAttempted` instance variables. These instance variables are set to appropriate values for each new simulation task, and using them in this method is their intended use.

Note: this method should be used to record logging/meta-information, not simulation results, which should instead be done in `recordSimResults()`.

Note: this method is called only **once** per simulation *task*. In particular, if a simulation is retried due to failure, this method only gets called only once, after all simulation attempts have been made.

`void otherParserOptions()`

Called in the `parseArgsAndSetInstanceVars(String[])` method.

This method is called after `BatchWorkerBase` adds its own (default) options, but before argument parsing begins.

Subclasses can optionally override this method to add additional command-line arguments to a `BatchWorker` using the `myParser` instance variable.

Note: care should be taken to avoid name clashes between arguments.


```
void postSim()
```

Called in the `run()` method.

This method is called immediately after the current simulation has stopped (before anything else happens).

Subclasses can optionally override this method to add additional custom post-simulation behaviour. In particular, this method can be used to “undo” any temporary changes made to the target model in the `preSim()` method (if appropriate).

Note: this method is called once for each **attempted** simulation. That is, if a simulation fails, this method **gets called again** immediately after the re-attempted simulation ends.

```
preSim()
```

Called in the `run()` method.

This method is called immediately after the target model’s `Properties` are set according to the current simulation task (through `setPropVals()`), and immediately before the simulation begins.

Subclasses can optionally override this method to add additional custom pre-simulation behaviour. In particular, since this method is called immediately after `setPropVals()`, it can be used by a subclass to query the current `Property` values and tweak them if needed. For example, muscles have an `excitation` property, which should always be between 0 and 1. However, if the `Property` is specified as having, say, a Normal probability distribution (whose support is $-\infty$ to ∞), then even with a mean between 0 and 1 and a very small variance, there is a non-zero probability of sampling values outside the allowable range. These (extremely rare) invalid values can be detected and corrected in this method (e.g. by clamping values between 0 and 1).

Note: in addition to being able to read the current `Property` values from the `Properties` themselves, the `myCurrentTask` instance variable contains the property-value pairs of the current simulation task, and subclasses can access this list of property-value pairs directly.

Note: this method is called once for each **attempted** simulation. That is, if a simulation fails, this method **gets called again** immediately before the re-attempted simulation begins.

```
recordSimResults()
```

Called in the `run()` method.

Near the end of each simulation task (right before `addLogEntry()`), this method is called to allow a recording of the results of the current simulation task in any file or format desired.

Subclasses can optionally override this method to add custom recording of simulation results (no results are otherwise recorded). In particular, subclasses are encouraged to make heavy use of the `myCurrentTask`, `myTaskCounter`, `myCurrentTaskSuccessful`, `myLastStepSizeUsed`, and `myNumSimsAttempted` instance variables. These instance variables are set to appropriate values for each new simulation task, and using them in this method is their intended use.

Note: a number of convenience methods for recording general simulation results already exist in [BatchWorkerBase](#). At the time of writing, these are `recordBinaryWayPoints(boolean)`, `recordStateAsASCII(ModelComponent[], String)`, `recordAllProbes()`, and `writePropVals(boolean)`. Refer to the Javadocs for details.

Note: this method should be used to record simulation results, not logging/meta-information, which should instead be done in `addLogEntry()`.

Note: this method is called only **once** per simulation *task*. In particular, if a simulation is retried due to failure, this method only gets called only once, after all simulation attempts have been made.

```
setUpStopConditionMonitor()
```


Called in the `run()` method.

This method is called **once** by the `BatchWorker` (at the very beginning of the `run()` method, before any simulations have been performed).

Subclasses **must** override this method to add `Conditions` to the `StopConditionMonitor` (see Section 6.3).

Note: `Conditions` can also be dynamically added or removed to the `StopConditionMonitor` on a per-simulation basis in the `preSim()` and `postSim()` methods.

`startLogSession()`

Called in the `run()` method.

This method is called **once** by the `BatchWorker` (at the very beginning of the `run()` method, before any simulations have been performed).

Subclasses can optionally override this method to add additional custom set up of the log file or log file writer (e.g. adding a header line) beyond simply creating either object (as that is already done by `BatchWorkerBase` in `parseArgsAndSetInstanceVars(String [])`).

Note: use the `myLogFileWriter`, `myLogComment`, and `mySeparator` instance variables to achieve desired results in this method.

Note: in addition to these abstract methods to override, there are two instance variables that can optionally be overridden: `myLogComment` and `mySeparator`. Unless you know what you are doing, **all other instance variables can be accessed in `BatchWorkerBase` subclasses, but should be treated as read-only**. Refer to the Javadocs for details.

Note: to see a working example of overriding the abstract `BatchWorkerBase` methods, see `SimpleTimedBatchWorker`. Also, see Section 6.5.

6.3 Conditions and the `StopConditionMonitor`

6.3.1 The need for `Conditions` and the `StopConditionMonitor`

The **only** `BatchWorkerBase` abstract method that **must** have a non-empty body in a subclass in order for the `BatchWorker` to function properly is `setUpStopConditionMonitor()`. This is because, when `BatchWorkerBase` starts a simulation, it doesn't inherently know when the simulation should stop. Therefore, it simply blocks (puts its main thread to sleep), waiting for the simulation (which is running in another thread) to stop.

This blocking behaviour is necessary, but presents two problems. First, since `ArtiSynth` does not require a target model to have any sort of simulation stopping mechanism (e.g. a breakpoint), some simulations will never stop, meaning the `BatchWorker` will block indefinitely. Second, if an error occurs, a simulation may stop prematurely, which the user will likely want to be told about. For example, inverted-element `NumericalExceptions` are frequently thrown during simulations involving finite-element models, rendering the simulation useless.

For `BatchSim`, the solution to these two problems comes in the form of `Conditions` and the `StopConditionMonitor`. Adding a `StopConditionMonitor` with appropriate stopping `Conditions` to a target model helps ensure that all simulations will eventually terminate. It also enables `BatchWorkerBase` to query the `StopConditionMonitor` in order to determine if a particular simulation ended because it reached an appropriate stopping condition, or if it ended due to an error (such as an exception being thrown).

For these reasons, it is crucial that concrete `BatchWorkerBase` subclasses override `setUpStopConditionMonitor()` in a way that is appropriate for the target model being used, as otherwise a simulation may either never stop, or else be inappropriately deemed a success or failure once it does stop.

Note: `Conditions` can also be dynamically added or removed to the `StopConditionMonitor` on a per-simulation basis in the `preSim()` and `postSim()` methods (see Section 6.2).

6.3.2 How Conditions and the StopConditionMonitor work

Although the idea of `Conditions`, `ConditionCheckers`, and `ConditionMonitors` could be widely applicable within `ArtiSynth` in general, the motivation for their development stems from the need for a model-independent way of representing conditions under which a simulation should stop in `BatchSim`, and ensuring that a simulation does stop when such a condition is met.

A `Condition` is a conceptual representation of something that can be either true or false (or equivalently “met” or “not met”) at any given point time, but not both. However, the truth value of a `Condition` is generally expected to change as time progresses (due to changes in some sort of external state).

From an implementation point of view, a class that implements the `Condition` interface represents a type or group of related conditions, and will usually be implemented as an enumeration, where each enumerated value represents one particular value of the associated `Condition` type. For example

```
enum BooleanLiteralCondition implements Condition {
    TRUE, FALSE;
}
```

represents the Boolean literal condition type, which has only two condition values: one condition that is invariably true, and one that is invariably false. This is a simple yet unusual `Condition`, because it’s truth value does not change as time progresses.

A `ConditionChecker` checks whether a particular `Condition` value is met at the moment (i.e. during the time step) the `ConditionChecker`’s `conditionMet()` or `conditionMet(Condition)` method is called.

`ConditionCheckerBase` is a base class that implements the functionality common to all `ConditionCheckers`. In particular, all `ConditionCheckers` are expected to provide two types of public constructors. The first takes a `Condition`, as may be expected. The second takes an additional parameter: another `ConditionChecker`, known as a *nested* `ConditionChecker`. In so doing, `ConditionCheckers` use the [decorator pattern](#) (just as Java’s [I/O Framework](#) does).

Note: a third type of public constructor exists that additionally takes a `String` name for the `ConditionChecker`. Although subclasses are strongly encouraged to implement such a constructor (setting names is useful for retrieving specific `ConditionCheckers` from the `StopConditionMonitor`), doing so is not required, as names can be set using `setName(String)`.

When calling `conditionMet()` or `conditionMet(Condition)`, the return value should be true if, and only if, **both** the outer `ConditionChecker`’s `Condition` *and* its nested `ConditionChecker`’s `Condition` are met. The outer and nested `ConditionCheckers` thus form a compound `ConditionChecker` with a compound `Condition` that is the equivalent of taking the logical AND of the sub-`Conditions`.

Note: `Condition` nesting can be done repeatedly (i.e. one `ConditionChecker` nested inside another that is itself nested inside a third, and so on). This creates a compound `Condition` that is composed of a chain of ANDed sub-`Conditions`.

Note: due to the associativity of logical AND, the order of nesting has no effect on the return value of `conditionMet()` or `conditionMet(Condition)`. It may, however, affect computational efficiency (`Condition` checking is done from the outside in using Java’s built-in logical operation short-circuiting).

A `ConditionMonitor` is an `ArtiSynth` `Monitor` to which `ConditionCheckers` have been added. In its `apply(double, double)` method, a `ConditionMonitor` asks all the outer (i.e. non-nested) `ConditionCheckers` that have been added to it to check whether their respective (possibly compound) `Condition` has been met, each will verify and report its truth value independently of the others. The result of this check can then be queried using the following methods

```
boolean hasAnyConditionBeenMet()
```

Returns true if any `ConditionCheckers`’ `Condition` has been met during the last call to `apply(double, double)`, and false otherwise.

```
boolean hasAnyConditionBeenMet (Condition)
```

Returns true if the given Condition has been met during the last call to `apply(double, double)`, and false otherwise.

```
List<Condition> getMetConditions()
```

Returns a list of all Conditions that have been met during the last call to `apply(double, double)`.

```
List<ConditionChecker<?>> getConditionCheckers()
```

Returns all the ConditionCheckers. This is useful only if `getConditionChecker(String)` fails because multiple ConditionCheckers has the same name or a null name.

```
ConditionChecker<?> getConditionChecker (String)
```

Returns the first ConditionChecker whose name equals the given name. If the given name is null, returns the first ConditionChecker whose name is null.

Note: with the desired ConditionChecker in hand, it can be directly queried using its `conditionMet()` method. This is only necessary in certain niche cases (e.g. when the same Condition could be met according to multiple ConditionChecker, but knowing exactly which ConditionChecker has checked that it has been met is important), and is not the most efficient way to check for met Conditions in general.

Note: whereas nesting ConditionCheckers is equivalent of taking the logical AND of their respective Conditions, some applications may require taking the logical OR of several Conditions, instead. In this case, the Conditions' respective ConditionCheckers should **not** be nested. Rather, they should be separately added to a ConditionMonitor, since the independent checking done by the outer ConditionCheckers in its `apply(double, double)` method is the equivalent of taking the logical OR of the ConditionCheckers' respective Condition. Logical OR can be easily queried using `hasAnyConditionBeenMet()`.

By nesting some ConditionCheckers, but not others, and then adding all the resulting outer ConditionCheckers to a ConditionMonitor, a complex Boolean Conditional expression can be created. When combined with the [NotChecker](#), every possible Boolean expression can be created.

Note: in addition to the [NotChecker](#), have a look at the various commonly-used ConditionChecker subclasses that have already been implemented for convenience, and don't be afraid to implement custom Conditions and ConditionCheckers, as needed. It's easy: with a given Condition, just subclass [ConditionCheckerBase](#), add some constructors, and override the `checkCondition (Condition, double, double)` method (see the Javadocs of [ConditionCheckerBase](#) for details).

Finally, a `StopConditionMonitor` is a `ConditionMonitor` that additionally calls `RootModel.setStopRequest(true)` in its `apply(double, double)` method if, and only if, `hasAnyConditionBeenMet()` returns true.

6.3.3 Adding Conditions to a StopConditionMonitor

Conditions are added to a `StopConditionMonitor` by overriding `setUpStopConditionMonitor()` in a concrete [BatchWorkerBase](#) subclass. [BatchWorkerBase](#) exposes a protected instance variable called `myStopConditionMonitor`, which subclasses should access in this method to add appropriate stopping Conditions.

For example, [SimpleTimedBatchWorker](#) (see Section 6.5) adds a `TimeCondition` that stops the simulation after a certain amount of simulation time has elapsed. To achieve this, we add a `TimeChecker` to `myStopConditionMonitor`, with parameters such that the check is true once the simulation is greater or equal to `myMaxTime` (using the specific `TimeCondition` called `NOT_LESS_THAN_MAX`). `myMaxTime` is an instance variable of [SimpleTimedBatchWorker](#) that stores the maximum allowable simulation time. All this is achieved using the following code

```
@Override
protected void setUpStopConditionMonitor () {
    myStopConditionMonitor.addConditionChecker (
        new TimeChecker (TimeCondition.NOT_LESS_THAN_MAX, 0, myMaxTime));
}
```

6.4 Customizing a Jython driver script

Unlike the `BatchManager`, which is typically run as a standalone application, each `BatchWorker` must run within its own instance of `ArtiSynth` (to be able to run `ArtiSynth` simulations). At the same time, each `BatchWorker` needs to be able to take control of `ArtiSynth`'s simulation interface (play, stop, reset, etc.), and so must be driven from outside the simulation. Ideally, `BatchSim` would be integrated into `ArtiSynth`. Then, one could start `ArtiSynth` with an option such as `-batch` or `-worker` and specify the classpath of a `BatchWorker` to run (see Section 7.4). But this is not likely to happen in the near future. In the mean time, `BatchWorkers` are driven from `ArtiSynth`'s Jython interface, which exists outside the simulation interface. To drive a `BatchWorker` through the Jython interface, `BatchSim` uses a very simple Jython driver script. The template for this script is

```
from <path-to-batch-worker-class-package> import <batch-worker-class>
from jarray import array
import sys

args = array(sys.argv, String)
worker = <batch-worker-class>(args)
worker.run()
quit()
```

In general, the steps required are

1. Import a `BatchWorkerBase` subclass.
2. Import `array` from `jarray`.
3. Import `sys` (for access to `sys.argv`).
4. Convert the command-line arguments passed to the script (but destined for the `BatchWorker`) from a Python list of strings (`sys.argv`) to a Java `String[]`.
5. Instantiate the `BatchWorker` (and pass it the `String[]` of arguments).
6. Run the `BatchWorker`.
7. Quit `ArtiSynth` once the `BatchWorker` has no more simulations to run.

Note: steps (2)-(4) are optional: if no command-line arguments will be passed to the `BatchWorker`, these lines can be omitted, and `None` can be passed to the constructor, such as in `worker = <batch-worker-class>(None)`. Alternatively, if some command-line arguments will *always* be passed to the `BatchWorker`, they can be hard-coded into the script file, such as in `args = array(["-n", "<worker-name>"], String)`.

Note: although such a use case is rare, in principle, custom Jython driver scripts can be made to do anything else the user wants besides simply driving a `BatchWorker`.

Note: in principle, it may be desirable to run() the `BatchWorker` in a separate `Thread` than the main Jython interface `Thread` (that is why `BatchWorkerBase` was made to implement `Runnable`) so that the Jython driver script can do other work in parallel (assuming a complex custom Jython driver script is being used). However, because `ArtiSynth` was never designed with `BatchSim` in mind, there is no easy way to synchronize `BatchSim` and `ArtiSynth`, and running the `BatchWorker` in a separate `Thread` may cause race conditions. At the time of writing, `BatchWorkerBase` only implements `Runnable` in case `ArtiSynth` one day provides better support for `BatchSim`.

For a working example of a Jython driver script, consider the following script used to run `SimpleTimedBatchWorker` (see Section 6.5)

```
from artisynth.tools.batchsim import SimpleTimedBatchWorker
from jarray import array
import sys

args = array(sys.argv, String)
worker = SimpleTimedBatchWorker(args)
worker.run()
quit()
```

6.5 Using the `SimpleTimedBatchWorker`

BatchSim always needs a `BatchWorker`. Most of the time, users need target model-specific outputs to be recorded, so they write their own custom `BatchWorker`. Alternatively, a “default” `BatchWorker` can be used: `SimpleTimedBatchWorker`. This `BatchWorker` has a single stop Condition: a `TimeCondition` that stops the simulation after a certain amount of simulation time has elapsed. After every simulation, `SimpleTimedBatchWorker` does some simple logging and records the end state of the model in binary waypoint files.

Alternatively, for any `BatchWorker` that needs a `TimeCondition` that stops the simulation after a certain amount of simulation time has elapsed, it may be worthwhile to subclass `SimpleTimedBatchWorker` rather than directly subclassing `BatchWorkerBase`. At the time of writing, this is actually quite common.

Note: when using `SimpleTimedBatchWorker`, a default Jython driver script can be used to drive the simulations: `<BatchsimRoot>/batchDriver.py`.

7 Miscellaneous

7.1 Killing a BatchSim session

Sometimes, due to user error (e.g. using the wrong `BatchWorker` class, target model, input file, or `Property`), either the `BatchManager` or some/all `BatchWorkers` (or both) will not shut down and quit properly. In these cases, it can be useful to have a command to quickly kill a running BatchSim session in order to release resources. On a Unix-like system using Bash, this can be achieved with the following command

```
> kill $(ps -u <username> -o pid,command | grep Batch | grep -v grep | \
> sed 's/^[[:space:]]*///' | cut -f 1 -d " ")
```

where `<username>` is the name of the user who owns/started the BatchSim session.

Note: the above command indiscriminately kills all running processes with `Batch` appearing somewhere in the command string. If this is not desired (for example, because there is more than one BatchSim session running, or because the class name of the `BatchWorkers` being used does not contain `Batch` as a substring), then a custom command with finer control should be used.

7.2 Post-processing

Note: since BatchSim (especially with many `BatchWorkers`) creates a lot of output files, it is easiest to put everything into one directory shared by all `BatchWorkers` as well as the `BatchManager` (see `-d` option to the `BatchWorker` in Section 4.5.2) and then operate with that directory as the current working directory.

7.2.1 Merging all `BatchWorker` output files using the `taskno` instance variable

To prevent data corruption, each `BatchWorker` writes to its own file. This implies that data is spread across multiple files whenever more than one `BatchWorker` is used. If the output dataset is very big, this may be desirable as it prevents the creation of one very big file that may not fit in memory when it is later processed. In most use cases, however, merging the separate data files into one big file is desirable as it facilitates further post-processing steps.

If the order of the simulations is important, or if cross-referencing across multiple *types* of output files (from the same `BatchWorker`) is important, record the `taskno` instance variable for each line of each output file (see Section 6.2, especially `recordSimResults()` and `addLogEntry()`), and use that to sort/cross-reference while merging.

Note: `taskno` is an integer (starting at 0 and incremented by 1) that identifies each simulation task uniquely. The `BatchManager` sends `taskno` to each `BatchWorker` together with the corresponding simulation task, and so `taskno` is unique to the entire BatchSim session, not to each `BatchWorker`. That is, when using more than one `BatchWorker`, each `BatchWorker` will likely not receive consecutive simulation tasks, but the `BatchWorkers` as a whole will. This makes sorting by `taskno` ideal when dealing with more than one `BatchWorker`.

Suppose `BatchWorker i` wrote to a file called `<i>_out.csv` (where `<i>` is `i` for each `BatchWorker`). Suppose further that, for each `i`, `<i>_out.csv` is a comma-separated file of outputs where each simulation takes one row in the file and `taskno` is recorded in the first column of each row. Then, on a Unix-like system using Bash, merging these files can be achieved with the following command

```
> cat *_out.csv | sort -n -t , -k 1,1 >> all_out.csv
```

This creates a file called `all_out.csv` that contains all the simulation outputs (sorted by `taskno`).

Once merged, it may be acceptable to remove `taskno` from the resulting file. Alternatively, `taskno` can still be used as a unique identifier for each simulation in further post-processing steps (e.g. to cross-reference a simulation across multiple types of output files from the same `BatchWorker`).

7.2.2 Cleanup

After merging, or if the data is no longer needed, it is easiest to remove all the output files by simply removing the shared output directory.

7.3 Useful diagrams

There are a number of useful diagrams in `<BatchsimRoot>/diagrams` that are available for your reference.

7.4 Ideas for future work

This section includes ideas for many quality-of-life or ease-of-use improvements to `BatchSim` that are not essential to its proper functioning, and, thus, have never been implemented due to lack of time.

1. Use `bm` as an alias for `java artisynth.tools.batchsim.manager.BatchManager`
2. Create a command called `bw` with the following usage pattern

```
> bw <BW_CP> [ <BW_OPT> ] <OTHER_AS_OPT>
```

that takes a `BatchWorker` class name (referred to as `<BW_CP>`), options to the `BatchWorker` (referred to as `<BW_OPT>`), and options to `ArtiSynth` (including the target model class name and any options to the target model; referred to as `<OTHER_AS_OPT>`), and then calls

```
> artisynth <OTHER_AS_OPT> -script fancyJythonDriver.py [ <BW_CP> <BW_OPT> ]
```

where `fancyJythonDriver.py` uses [Java Reflection](#) (or something similar) to call the given `<BW_CP>` constructor and pass it the given `<BW_OPT>`. This implies users no longer have to create their own Jython driver script.

Alternatively, we could add a `-batch` option to `ArtiSynth`, and then start `BatchWorkers` using

```
> artisynth <OTHER_AS_OPT> -batch <BW_CP> [ <BW_OPT> ]
```

This option would internally invoke `fancyJythonDriver.py` (or do something equivalent).

3. Create a program that reads an input file of the form

```
BM_OPT="..."
OTHER_AS_OPT="..."
BW_CP="..."
N_WORKERS="..."
BW_OPT_I="...$i..."
BM_REDIRECT="..."
BW_REDIRECT_I="...$i..."
```

where `BM_OPT` are options to the `BatchManager` (including the input file), `OTHER_AS_OPT` are options to `ArtiSynth` (including the target model class name and any options to the target model), `BW_CP` is a `BatchWorker` class name, `N_WORKERS` is the number of `BatchWorkers` to instantiate, `BW_OPT_I` are options to the `BatchWorker` (in which each `$i` will be replaced in the string by the `BatchWorker`'s name), `BM_REDIRECT` is a file name to which to redirect the `BatchManager`'s console output, and `BW_REDIRECT_I` is a file name to which to redirect the `BatchWorker`'s console output (in which each `$i` will be replaced in the string by the `BatchWorker`'s name).

After reading this input file, the program calls (on a Unix-like system using `Bash`)

```
> bm $BM_OPT >> $BM_REDIRECT 2>&1 &
> for ((i = 0; i < N_WORKERS; i++)); do
>     bw $BW_CP [ $BW_OPT_I ] $OTHER_AS_OPT >> $BW_REDIRECT_I 2>&1 &
> done
```

using the shorthands in (1) and (2) above.

4. Create a program that is essentially just a GUI interface (using buttons, drop-down menus, input boxes, file browsers, etc.) for (3) above. An advanced version of this interface would enable direct input file editing (ideally with syntax highlighting).
5. Try to offer automatic merging of output files based on specifications of which column has the `taskno`, the number of header lines, etc. This would be fine for simple output file formats, but there is really no limit to how complex the output file format can get.
6. Ideally, some or all of the above would be integrated directly into `ArtiSynth`'s UI. For example the `-batch` option mentioned in (2).
7. `BatchSim` is complex enough that it could probably be published in a software engineering journal. If you are interested in pursuing this, please contact me and I'll gladly co-author the paper with you.