Manual for SMAC version vDEV

Frank Hutter & Steve Ramage
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, Canada
{hutter,seramage}@cs.ubc.ca

October 14, 2012

Contents

1 Introduction

This document is the manual for SMAC [2] (an acronym for *Sequential Model-based Algorithm Configuration*). SMAC aims to solve the following *algorithm configuration* problem: Given a binary of a parameterized algorithm $\mathcal{A}$, a set of instances $\mathcal{S}$ of the problem $\mathcal{A}$ solves, and a performance metric m, find parameter settings of $\mathcal{A}$ optimizing m across $\mathcal{S}$.

In slightly more detail, users of SMAC must provide:

- a parametric algorithm $\mathcal{A}$ (an executable to be called from the command line),
- a description of $\mathcal{A}$'s parameters $\theta_1, \ldots, \theta_n$ and their domains $\Theta_1, \ldots, \Theta_n$,
- a set of benchmark instances, Π, and
- the objective function with which to measure and aggregate algorithm preformance results.

SMAC then executes algorithm $\mathcal{A}$ with different *parameter configurations* (combinations of parameters $\langle\theta_1, \ldots, \theta_n\rangle \in \Theta_1 \times \cdots \times \Theta_n$, on instances $\pi \in \Pi$), searching for the configuration that yields overall best performance across the benchmark instances under the supplied objective. For more details please see [2]; if you use SMAC in your research, please cite that article. It would also be nice if you sent us an email – we are always interested in additional application domains.

1.1 License

SMAC is to be released under a dual usage license. Academic & non-commercial usage is permitted free of charge. Please contact us to discuss commercial usage.

1.2 System Requirements

SMAC itself requires only Java 6 or newer to run. The included scripts are currently only available for Unix-compatible operating systems. The included example scenarios require Ruby.

1.3 Version

This version of the manual is for SMAC vDEV-000

2 Differences Between SMAC and ParamILS

There are various differences between SMAC and ParamILS, including the following.

- **Support for continuous parameters:** While ParamILS was limited to categorical parameters, SMAC also natively handles continuous and integer parameters. See Section 4.4.1 for details.
- **Run objectives:** Not all of ParamILS's run objectives are supported at this time. If you require an unsupported objective please let us know.
- **Order of instances:** In contrast to ParamILS, the order of instances in the instance file does not matter in SMAC.
- **Configuration time budget and runtime overheads:** Both ParamILS and SMAC accept a time budget as an input parameter. ParamILS only keeps track of the CPU time the target algorithm reports and terminates once the sum of these runtimes exceeds the time budget; it does *not* take into account overheads due to e.g. command line calls of the target algorithm. In cases where the reported CPU time of each target algorithm run was very small (e.g. milliseconds), these unaccounted overheads could actually dominate ParamILS's wall-clock time. SMAC offers a more flexible management of its runtime overheads through the options **--countSMACTimeAsTunerTime** and **--wallClockLimit**. See Section 3.3 for details on the wall clock time limit.
- **Resuming previous runs:** While this was not possible in ParamILS, in SMAC you can resume previous runs from a saved state. Please refer to Section 3.4 for state restoration and to Section 5.2 for information on the state files SMAC saves.
- **Feature files:** SMAC accepts as an optional input a feature file providing additional information about the instances in the training set; see Section 4.3.

- **Algorithm wrappers:** The wrapper syntax has been extended in SMAC to support additional results in the "solved" field. Specifically, there is a new result **ABORT** signaling that the configuration process should be aborted (e.g. because the wrapper is in an inconsistent state that should never be reached). A similar behaviour is triggered if option **--abortOnFirstRunCrash** is set and the first run returns **CRASHED**.
- **Instance files vs. instance/seed files:** The **instance_file** parameter now auto-detects whether the file conforms to ParamILS's **instance_file** or **instance_seed_file** format. SMAC treats the latter option as an alias for the former. See Section 4.2 for details. While SMAC is backwards compatible with previous (space-separated) files, the prefered format is now `.csv`.

3 Commonly Used Options

To get started with an existing configuration scenerio you simply need to execute smac as follows:

```
./smac --scenarioFile <file> --numRun 0
```

This will execute SMAC with the default options on the scenario specified in the file. Some commonly-used non-default options of SMAC are described in this section. The **--numRun** argument controls the seed and names of output files (to support parallel independent runs)

3.1 ROAR Mode

```
./smac --scenarioFile <file> --executionMode ROAR --numRun 0
```

This will execute the ROAR algorithm, a special case of SMAC that uses an empty model and random selection of configurations. See [2] for details on ROAR.

3.2 Adaptive Capping

```
./smac --scenarioFile <file> --adaptiveCapping true --numRun 0
```

Adaptive Capping (originally introduced for ParamILS [3], but also applicable in SMAC [1]) will cause SMAC to only schedule algorithm runs for as long as is needed to determine whether they are better than the current incumbent. Without this option, each target algorithm runs up to the runtime specified in the configuration scenario file **--cutoffTime**.
NOTE: Adaptive Capping should only be used when the **--runObj** is RUNTIME. Adaptive capping can drastically improve SMAC's performance for scenarios with a large difference between **--cutoffTime** and the runtime of the best-performing configurations. Related configuration options are **--capSlack**, **--capAddSlack**, and **--imputationIterations**.

3.3 Wall-Clock Limit

```
./smac --scenarioFile <file> --runtimeLimit <seconds> --numRun 0
```

SMAC offers the option to terminate after using up a given amount of wall-clock time. This option is useful to limit the overheads of starting target algorithm runs, which are otherwise unaccounted for. This option does not override **--tunerTimeout** or other options that limit the duration of the configuration run; whichever termination criterion is reached first triggers termination.

3.4 State Restoration

```
./smac --scenarioFile <file> --restoreStateFrom <dir>
       --restoreIteration <iteration> --numRun 0
```

SMAC will read the files in the specified directory and restore its state to that of the saved SMAC run at the specified iteration. Provided the remaining options (e.g. **--seed**, **--overall_obj**) are set identicially, SMAC should continue along the same trajectory.

This option can also be used to restore runs from SMAC v1.xx (although due to the lossy nature of Matlab files and differences in random calls you will not get the same resulting trajectory). By default the state can be restored to iterations that are powers of 2, as well as the 2 iterations prior to the original SMAC run stopping. If the original run crashed, additional information is saved, often allowing you to replay the crash.

NOTE: When you restore a SMAC state, you are in essence preloading a set of runs and then running the scenario. In certain cases, if the scenario has been changed in the meantime, this may result in undefined behaivor. Changing something like **--tunerTimeout** is usually a safe bet, however changing something central (such as **--runObj**) would not be.

To check the available iterations that can be restored from a saved directory, use:

```
./smac-possible-restores <dir> --numRun 0
```

To disable saving any state information to disk, use

```
./smac --scenarioFile <file> --stateSerializer NULL --numRun 0
```

3.5 Concurrent Algorithm Execution Requests

```
./smac --scenarioFile <file> --maxConcurrentAlgoExecs <num> --numRun 0
```

In certain circumstances, it may be much faster to allow more than one target algorithm execution at once, (e.g., when multiple cores are available or when actual algorithm execution is I/O bound). To exploit this, you can have SMAC schedule multiple runs at a time. If **--adaptiveCapping** is not set, this will result in the same trajectory as a sequential version (when **--maxConcurrentAlgoExecs** is set to 1). When adaptive capping is enabled, concurrent runs are scheduled with cutoff times as if each were the first of the runs to be scheduled.

3.6 Named Rungroups

```
./smac --scenarioFile <file> --runGroupName <foldername> --numRun 0
```

All output is written to the folder `<foldername>`; runs differing in **--numRun** will yield different output files in that folder.

3.7 Offline Validation

SMAC includes a tool for the offline assessment of incumbents selected during the configuration process. By default, given a test instance file with N instances, SMAC performs $\approx 1\,000$ target algorithm validation runs per configuration (rounded up to the nearest multiple of N).

By default, SMAC limits the number of seeds used in validation runs to 1 000 seeds per instance. This number can be changed by e.g. using

```
./smac --scenarioFile <file> --numSeedsPerTestInstance 50
```

(This parameter does not have any effect in the case of instance/seed files.)

3.7.1 Limiting the Number of Instances Used in a Validation Run

To use only some of the instances or instance seeds specified you can limit them with the **--numTestInstances** parameter. When this parameter is specified, SMAC will only use the specified number of lines from the top of the file, and will keep repeating them until enough seeds are used:

```
./smac --scenarioFile <file> --numTestInstances 10
```

For instance files containing seeds, this option will only use the specified number of instance seeds in the file.

3.7.2 Disabling Validation

Validation can be skipped alltogether as follows:

```
./smac --scenarioFile <file> --skipValidation
```

3.7.3 Standalone Validation

SMAC also includes a method of validating configurations outside of a smac run. You can supply a configuration using the **--configuration** option; apart from this directive the rest of the options are a subset of those from SMAC. Here is an example call:

```
./smac-validate --scenarioFile <file> --numValidationRuns 10000
     --configuration <config string> --maxConcurrentAlgoExecs 8
```

Usage notes for the offline validation tool:

1. If no configuration is specified the default configuration is validated.
2. This validates against the test set only; the training instance set is of no consequence here.
3. By default this outputs into the current directory; you can change the output directory with the option **--runGroupName**.

4 File Format Reference

4.1 Option Files

Option Files are a way of saving a different set of values frequently used with SMAC without having to specify them on every execution. The general format for an option file is the name of the configuration option (without the two dashes), an equal sign , and then the value (for booleans it should be true or false, lowercase). Currently options that take multiple arguments are not supported. Additionally you can not use aliases that are single dashed (*e.g.* to override the Experiment Directory, you must use –experimentDir and not -e)

When using Option Files it is important that no two files (including the Scenario File), specify the same option, the resulting configuration is undefined, and in general this will not throw an error.

4.1.1 Scenario File

The Scenario Option File, or Scenario File, is the recommended way of configuring SMAC [1]. The Scenario Files used in SMAC are backwards compatible with ParamILS and the name of option names here reflect that[2]. NOTE: **cutoff_length** is not currently supported.

algo An algorithm executable or wrapper script around an algorithm that conforms with the input/output format specified in section 4.5. The string here should be invokable via the system shell.

execdir Directory to execute `<algo>` from: (*i.e.* "cd `<execdir>`; `<algo>`")

deterministic A boolean that governs whether or not the algorithm should be treated as deterministic. For backwards compatibility with ParamILS, this option also supports using 0 for false, and 1 for true. SMAC will never invoke the target algorithm more than once for any given instance, seed and configuration. If this is set to `true`, SMAC will never invoke the target algorithm more than once for any given instance and configuration.

run_obj Determines how to convert the resulting output line into a scalar quantifying how "good" a single algorithm execution is, (*e.g.*how long it took to execute, how good of a solution it found, etc...). Currently implemented objectives are the following:

Name	Description
RUNTIME	**The reported runtime of the algorithm.**
QUALITY	**The reported quality of the algorithm.**

overall_obj While **run_obj** defines the objective function for a single algorithm run, **overall_obj** defines how those single objectives are combined to reach a single scalar value to compare two parameter configurations. Implemented examples for this are as follows:

Name	Description
MEAN	**The mean of the values**
MEAN1000	**Unsuccessful runs are counted as 1000 $\times$ cutoff_time**
MEAN10	**Unsuccessful runs are counted as 10 $\times$ cutoff_time**

cutoff_time The CPU time after which a single algorithm execution will be terminated as unsuccess (and treated as a **TIMEOUT**). This is an important parameter: If chosen too high, lots of time will be wasted with unsuccessful runs. If chosen too low the optimization is biased to perform well on easy instances only.

tunerTimeout The limit of the CPU time allowed for configuration (*i.e.*The sum of all algorithm runtimes, and by default the sum of the CPU time of SMAC itself).

paramfile Specifies the file with the parameters of the algorithm. The format of this file is covered in section 4.4

[1]Nothing in general prevents you from specifying non-scenario options in these files, but in general you should restrict your files to these.

[2]Every option name listed here is in fact an alias for an existing option listed in the section 9.4 and it is entirely possible to use SMAC without using Scenario Files.

outdir Specifies the directory SMAC should write its results to.

instance_file Specifies the file containing the list of problem instances (and possibly seeds) for SMAC to use during the *Automatic Configuration Phase*. The ParamILS parameter **instance_seed_file** aliases this one and the format is auto-detected. The format of these files is covered in section 4.2.

test_instance_file Specifies the file containing the list of problem instances (and possibly seeds) for SMAC to use during *Validation Phase*. The ParamILS parameter **test_instance_seed_file** aliases this one and the format is auto-detected. The format of these files is covered in section 4.2.

feature_file Specifies the file with a list of test instances. The format of this files is covered in section 4.3.

4.2 Instance File Format

The files used by the **instance_file** & **test_instance_file** options come in four potential formats, all of which are CSV based[3]. Before specifying the formats it is important to note the three kinds of information that are specified with instances [4].

Instance Name The name of the instance that was selected. This should be meaningful to the target algorithm we are configuring [5].

Instance Specific Information A free form text string (with no spaces or line breaks) that will be passed to the Target Algorithm whenever executed.

Seed A specific seed to use when executing the target algorithm.

The possible formats are as follows, and depend on what information you'd like to specify.

1. Each line specifies only a unique **Instance Name**. No **Instance Specific Information** will be used, and **Seed**'s will be automatically generated.

2. Each line specifies a **Seed** followed by the **Instance Name**. Every line must be unique, but for each **Instance Name** additional seeds will be used in order, when that instance is selected.

3. Each line specifies a **Instance Name** followed by the **Instance Specific Information**. Every **Instance Name** must be unique, **Seed**'s will be automatically generated.

4. Each line specifies a **Seed** followed by the **Instance Name** followed by the **Instance Specific Information**. Every line must be unique, and furthermore, for all **Instance Name**'s the **Instance Specific Information** must be the same for all **Seed** values (*i.e.*You cannot specify different instance specific information that is a function of the seed used).

[3]Specifically each cell should be double-quoted (*i.e.*"), and use a comma as a cell delimiter. SMAC also supports the old method of reading files that use space as a cell delimiter and do not enclose values. However these files cannot handle **Instance Name**'s that contain spaces.

[4]Features which are required for SMAC but not ParamILS are specified in a seperate file see section 4.3.

[5]Generally **Instance Names** reference specific files on disk.

4.3 Feature File Format

The **feature_file** specifies features that are to be used for instances. When executing in *ROAR* mode, feature files are not necessary. Feature Files are specified in CSV format, the first column of every row should list an **Instance Name** as it appears in the **instance_file**. The subsequent columns should list real values specifying a computed continuous feature. By convention the value -512, and -1024 are used to signify that a feature value is missing or not applicable. All instances must have the same number of features.

At the top of the file there MUST appear a header row, the cell that appears above the instance names is unimportant, but for each feature a unique and *non-numeric* feature name must be specified.

4.4 Algorithm Parameter File

The parameter configuration space of your algorithm need to be defined in a file that is specified by the **paramfile** option. Comments in the file begin with a #, and run to the end of the line.

The file consists of three types of statements:

Parameter Declaration Clauses specifies the name of parameters, and their domains.

Conditional Parameter Clauses specify when a parameter is active.

Forbidden Parameter Clauses specify when a combination of parameter settings is illegal and shouldn't be ignored.

4.4.1 Parameter Declaration Clauses

SMAC supports two types of parameters, categorical and numeric. The former is specified as follows:

```
name { value1, ..., value_n } [defaultValue]
```

Example:

```
timeout { 1,5,10,50,100,500,1000,5000,10000 } [500]
```

Here a categorical parameter is declared named `timeout`, its values must be one of the values listed, and it has a default of 500.

Numeric Parameters (both continuous and integral) are specified as follows:

```
name [minValue, maxValue] [defaultValue](i)(l)
```

Example 1:

```
timeout [1, 10000] [500]
```

We have specified timeout as numeric with a default value of 500. Any value is legally permitted so long as it's in the real interval of [1, 10000]. When drawing a random configuration out of this space they are drawn uniformly.

Example 2:

```
timeout [1, 10000] [500]l
```

This example is identical to the previous, except that when drawing random configurations we do so uniformly on a $\log_{10}$ scale (*e.g.* a value between [1, 100] is as likely to be selected as between [100, 10000]).

Example 3:

```
timeout [1, 10000] [500]i
```

In this example the only legal values are integers in the range [1, 10000], we select from these integers uniformly.

Example 4:

```
timeout [1, 10000] [500]il
```

In this example integers in the range [1, 10000] are the only values permitted, and when randomly selecting them we do so on a $\log_{10}$ scale.

Restrictions

Integer Numeric integral parameters must have all values specified as integers, even though strictly speaking the notation should permit fractional values. Additionally the default value must be a integer.

Log Log parameters must start at strictly greater than zero.

4.4.2 Conditional Parameter Clause

Conditional parameter clauses specify when a parameter is active. A parameter is active when for each clause that lists it as a dependent, the independent variable is active and has a value that satisfies the operation [6]. Conditional Parameter Clauses have the following syntax:

```
dependentName | independentName operation { value1, ... , value_n}
```

Example:

```
sort-algo { quick, insertion, merge, heap, stooge, bogo } [ bogo ]
quick-revert-to-insertion { 1,2,4,8,16,32,64 } [16]
quick-revert-to-insertion | sort-algo in { quick }
```

In the above example the `quick-revert-to-insertion` is conditional on the `sort-algo` parameter being set to `quick`, and will be ignored otherwise.

4.4.3 Forbidden Parameter Clauses

Forbidden Parameters are combinations of parameter settings which should not be treated as valid by SMAC. During the search phase, parameters matching a forbidden parameter configuration, will not be explored [7].

The Syntax is as follows:

```
{name1=val1,name2=val2...}
```

Example

```
quick-sort { on, off} [on]
bubble-sort { on, off} [off]
{quick-sort=on, bubble-sort=on}
{quick-sort=off, bubble-sort=off}
```

The above example implements an exclusive-or [8]. The first forbidden parameter clause prevents both sort techniques from being on at the same time. The second ensures that atleast one of them is on. NOTE: The default parameter setting cannot itself be a forbidden parameter setting.

[6]The only supported operation presently is `in`.

[7]Specifying a large number of forbidden parameters may degrade SMAC's performance substantially.

[8]admittedly it could be better modelled with a simple categorical parameter.

4.5 Algorithm executable / wrapper

The target algorithm as specified by the **algo** parameter must obey the following general contracts. While modifying your own code to directly achieve this is one option there are other methods outlined in section 4.5.3.

4.5.1 Invocation

The algorithm must be invokable via the system command-line using the following command with arguments:

```
   <algo_executable> <instance_name> <instance_specific_information> <cutoff_time>
<cutoff_length> <seed> <param> <param> <param>...
```

algo_executable Exactly what is specified in the **algo** argument in the scenario file.

instance_name The name of the problem instance we are executing against.

instance_specific_information An arbitrary string associated with this instance as specified in the **instance_file** . If no information is present then a "0" is always passed here.

cutoff_time The amount of time in seconds that the target algorithm is permitted to run. It is the responsibility of the callee to ensure that this is obeyed. It is not necessary that that the actual algorithm execution time (wall clock time) be below this value (*e.g.*If the algorithm needs to cleanup, or it's only possible to terminate the algorithm at certain stages).

cutoff_length A domain specific measure of when the algorithm should consider itself done.

seed A positive integer that the algorithm should use to seed itself (for reproducability). "-1" is used when the algorithm is **deterministic**

param A setting of an active parameter for the selected configuration as specified in the Algorithm Parameter File. SMAC will only pass parameters that are active. Additionally SMAC is not guaranteed to pass the parameters in any particular order. The exact format for each parameter is:

```
-name 'value'
```

All of the arguments above will always be passed, even if they are inapplicable, in which case a dummy value will be passed.

4.5.2 Output

The Target Algorithm is free to output anything, which will be ignored but must at some point output a line (only once) in the following format[9]

```
   Result for ParamILS: <solved>, <runtime>, <runlength>, <quality>, <seed>,<additi
rundata>
```

[9]ParamILS in not a typo. While other values are possible including SMAC, HAL. ParamILS is probably the most portable. The exact Regex that is used in this version is: **^\s*(Final)?\s*[Rr]esult\s+(?:(for)—(of))\s+(?:(HAL)—(ParamILS)—(SMAC)—(this wrapper))**

solved Must be one of **SAT** (signifying a successful run that was satisfiable), **UNSAT** (signifying a successful run that was unsatisfiable), **TIMEOUT** if the algorithm didn't finish within the allotted time, **CRASHED** if something untoward happened during the algorithm run, or **ABORT** if something prevents the target algorithm for successfully executing and it is believed that further attempts would be futile.

SMAC does not differentiate between **SAT** and **UNSAT** responses, and the primary use of these is to serve as a check that the algorithm is executing correctly by allowing the algorithm to output a yes / no output that can be checked.

SMAC also supports reporting **SATISFIABLE** for **SAT** and **UNSATISFIABLE** for **UNSAT**. NOTE: These are only aliases and SMAC will not preserve which alias was used in the log or state files.

ABORT can be useful in cases where the target algorithm cannot find required files, or a permission problem prevents access to them. This will cause SMAC to stop running immediately. Use this option with care, it should only be reported when the algorithm knows for CERTAIN that subsequent results may fail. For things like sporadic network failures, and other cosmic-ray induced failures, one should consider using **CRASHED** in combination with the **–retryTargetAlgorithmRunCount** and –abortOnCrash options, to mitigate these.

runtime The amount of CPU time used during this algorithm run. SMAC does not measure the CPU time directly, and this is the amount that is used with respect to **tunerTimeout**. You may get unexpected performance degredation when this amount is heavily under reported [10].

NOTE: The **runtime** should always be strictly less than the requested **cutoff_time** when reporting **SAT** or **UNSAT**.

If an algorithm reports **TIMEOUT** or **CRASHED** the algorithm can report the actual CPU time usage, and SMAC will correctly treat it as the cap time.

runlength A domain specific measure of how far the algorithm progressed.

quality A domain specific measure of the quality of the solution.

seed The seed value that was used in this target algorithm execution. NOTE: This seed MUST match the seed that the algorithm was called with. This is used as fail-safe check to ensure that the output we are parsing really matches the call we requested.

Like invocation, all fields are mandatory, when not applicable 0's can be substituted.

4.5.3 Wrappers & Native Libraries

In order to optimize an algorithm, SMAC needs a method of invoking it. While modifying the code to manage the timing and input mechanisms manually is possible, this can sometimes be invasive and difficult to manage. There exist three other methods that one could consider using.

Wrappers Executable Scripts that manage the resource limits automatically and format the specified string into something usable by the actual target algorithm. This approach is probably the most common,

[10]This typically happens when targeting very short algorithm runs with large overheads that aren't accounted for.

but among its drawbacks are the fact that they often rely on third party scripting languages, and for smaller execution times have a large amount of overhead that may not be accounted for as far as the **tunerTimeout** limit is concerned. Most of the examples included in SMAC use this approach, and the wrappers included can be adapted for your own projects.

NOTE: When writing wrappers it's important not to poll the output stream of the target algorithm, especially if there is lots of output. Doing so often results in lock-contention and significantly modifies the runtime performance of the algorithm enough that the resulting configuration is not well tuned to the real algorithm performance.

Native Libraries Augmentation Libraries exist (See: **TBD**) for C and Java currently that facilitate adding the required functionality directly to the code. While parsing the arguments into the necessary data structures is still required, they do manage the timing and output requirements in most cases. Unlike the previous approach however, certain crashes may not allow the the values to be outputted (*e.g.* a segmentation fault occurs).

Target Algorithm Evaluators This is probably the most powerful, but also the most complicated approach. SMAC is architected in a way that makes it fairly simple to replace the mechanism for execution with something completely custom. This can be done without even recompiling SMAC by creating a new implementation of the `TargetAlgorithmEvalutor` interface, which is responsible for converting run requests (`RunConfig` objects) into run results (`AlgorithmRun` objects). Both the input and output objects are simple *Value Objects* so the coupling between SMAC and the rest of your code is almost zero with this approach. For more information see 6.3

5 Interpreting SMAC's Output

SMAC outputs a variety of information to log files, trajectory files, and state files. Most of the files are human readable, and this section describes these files.
NOTE: All output is written to the **outdir** in the **--runGroupName** sub-directory.

5.1 Logging Output

SMAC uses slf4j (http://www.slf4j.org/), a library that allows for abstracting and replacing the logging system with ease, and uses logback (http://logback.qos.ch/) as the default logging system. While there is limited ability to change logging options via the command line (*e.g.*,**--logLevel**,**--consoleLogLevel**,**--logAllCallStrings**,**--logAllProcessOutput**), one can edit `conf/logback.xml`, to get much more control over the logging of SMAC. For more details of how to edit this file consult the logback documentation.

NOTE: If you replace the logger in SMAC or modify the configuration file, the logging command line options may no longer work.

By default SMAC writes the following logging files out to disk (NOTE: The N represents the **--numRun** setting):

log-run$N$.txt A log file that contains a full dump of all the information logged, and where it was logged from.

log-warn$N$.txt Contains the same information as the above file, except only from warning and higher level messages.

log-err$N$.txt Contains the same information as the above file, except only from error messages.

raw-run$N$.txt Only the actual logging messages (this is easier to grep for needed information).

runhashes-run$N$.txt A file that contains only the Run Hash Codes for a given run see the corresponding entry in the **FAQ**.

5.1.1 Interpreting the Log File

SMAC basically goes through three phases when executing:

- Setup Phase Input files are read, and their arguments validated. Everything necessary to execute the Automatic Configuration Phase, is constructed. This phase ends (baring anything that must be lazily loaded), once `Automatic Configurator Started` is logged.
- Automatic Configuration Phase, SMAC is now actively configuring the target algorithm. SMAC will spend most of it's time here, and outputs it's progress. The most important output is the Runtime Statistics which will appear like:

```
[INFO ] *****Runtime Statistics*****
 Iteration: 35
 Incumbent ID: 64 (0x18824F)
 Number of Runs for Incumbent: 70
 Number of Instances for Incumbent: 70
 Number of Configurations Run: 67
 Performance of the Incumbent: 1589.1414639125514
 Total Number of runs performed: 242
 Wallclock time: 18.213 s
 Wallclock time remaining: 2.147483628787E9 s
 Configuration time budget used: 84056.83939320213 s
 Configuration time budget remaining: 2343.160606797872 s
 Sum of Target Algorithm Execution Times \
  (treating minimum value as 0.1): 84036.36939320213 s
 CPU time of Configurator: 20.47 s
 User time of Configurator: 19.6 s
 Total Reported Algorithm Runtime: 84033.27806288192 s
 Sum of Measured Wallclock Runtime: 0.0 s
 Max Memory: 3505.8125 MB
 Total Java Memory: 1249.0625 MB
 Free Java Memory: 719.8940582275391 MB
[INFO ] *********************************************
```

  While most of the fields are self-explanatory some deserve special attention:

  `Incumbent ID`

  The second ID (0x18824F) is a hex-code that represents the configuration anywhere / everywhere it is logged. The first ID, 64, occurs in context where we know the configuration is intended to be run. This ID will corresponding to the ID in the state files The first ID should always appear with the second, but not conversely. The second ID roughly represents the specific configuration in memory [11].

  `Performance of the Incumbent`

  This represents the performance of the incumbent under the given **run_obj** and **overall_obj** on the runs so far.

[11]Specifically every time a configuration is modified, this number is incremented. In cases where the configuration space is small,or we are examining a small part of it, SMAC may end up back at the same configuration again. As far as the behaviour of SMAC is concerned these are identical, the ID is only ever used for logging.

`Configuration time budget used`

The tuner time that has been used so far.

`Sum of Target Algorithm Execution Times`

This represents the contribution of the algorithm runs to the Tuner Time (if applicable), in general each run contributes the minimum of 0.1 and it's reported runtime. This parameter differs from `Sum of Measurement Wallclock Runtime` in that the latter is a direct sum. If you are only running on algorithms with large runtime, this difference may be 0.

- Validation Phase, depending on the options used this can also take a large fraction of SMAC's runtime. The logic here is actually quite simple, as it largely only requires running many algorithm runs and computing the objectives from them.

  At the end of Validation the Runtime Statistics (from the Automatic Configuration Phase) are displayed again, as is the following information

  1. The performance of the incumbent on both the training and test set.
  2. A sample call of the final incumbent (selected configuration)
  3. The complete configuration selected (without inactive conditionals)
  4. The complete configuration selected (with inactive conditionals)
  5. The Return value of SMAC (generally 0 if successful)

5.2 State Files

State files allow you to examine and potentially restore the state of SMAC at a specific point of it's execution. The files are written to the state-run$N$/ sub-directory, where N is the value of **--numRun** option.

All files have the following convention as a suffix either `it` or `CRASH` followed by either the iteration number M, or in some cases `quick` or `quick-bak`.

The state is saved for every iteration m, where $m = 2^n \; n \in \mathbb{N}$, additionally it is saved when SMAC completes whether successfully or due to crash.

The following files are saved in this state directory (ignoring the suffix):

java_obj_dump Stores (Java) serialized versions of the the incumbent and the random object state. In general there is no need to look at this file, and it is not human readable.

paramstrings Stores a human readable setting of each configuration ran, with a prefix of the numeric id of the configuration (as used in the logs, and other state files).

uniq_configurations Stores the configurations ran in a more concise but effectively un-human readable form. The first column again is the numeric id of the configuration (as used in the logs, and other state files).

run_and_results Stores the result of every run of the target algorithm that SMAC has done. The first 13 columns (after the header row are designed to be backwards compatible with SMAC versions 1.xx. Each column is labelled with what data it contains, the following columns deserve some description.

`Instance ID` This is the instance used, and is the n^{th} **Instance Name** specified in the **instance_file** option.

`Response Value(y)` This is the value determined by the **run_obj** on the run.

`Censored`	Indicates whether the `Cutoff Time Used` field is less than the **cutoff_time** in the original run. 0 means `false`, 1 means `true`.
`Run Result Code`	This is a mapping from the `Run Result` to an integer for use with previous versions.

5.3 Trajectory File

SMAC also outputs a trajectory file into identical files `traj-run-`N`.txt` [12] and `traj-run-`N`.csv`. These files outline the incumbent (by id) over the course of execution and it's performance. The first line gives the **–runGroupName**, and then the **–numRun**.

The rest of the file follows the following format:

Column Name	Description
Total Time	Sum of all execution times and CPU time of SMAC
Incumbents Mean Performance	Performance of the Incumbent under the given **–runObjective** and **–overallObjective**
Incumbent's Performance σ	Outputs -1 Currently
Incumbent ID	The ID of the incumbent as listed in the **param_strings** file 5.2.
acTime	CPU Time of SMAC
Remaining Columns	Give a name value mapping for the configuration value as given by the `Incumbent ID` column

5.4 Validation Output

When Validation is completed four files are outputted, (again N is the value of the **–numRun** argument):

1. `rawValidationExecutionResults-run`N`.csv`:

   CSV File containing a list of the configuration, seeds & instance run and the corresponding result and the result of the target algorithm execution. This file is mainly for debugging.

2. `validationInstanceSeedResult-run`N`.csv`:

   CSV File containing a list of seeds & instances and the resulting response value. Again this file is mainly for debugging, but is easier to parse than the previous.

3. `validationResultsMatrix-run`N`.csv`:

   CSV File containing the list of instances on each line, the next column is the aggregation of the remaining columns under the **overall_obj**. Finally there is one additional row that gives the aggregation of all the individual **overall_obj**, aggregated in the same way.

6 Developer Reference

This section is meant as a guide to those who need to modify the SMAC code base for whatever reason.

[12] This file is outputted for backwards compatibility with existing scripts.

6.1 Design Overview

The SMAC Application is broken up into three distinct projects as follows:

SMAC Contains all of the logic that is specific to SMAC, (*e.g.*Validation, the SMAC algorithm, construction of SMAC Objects). In essence it stitches together components of the Automatic Configurator Library. The sources are included in `smac-src.jar`.

Automatic Configurator Library Contains all of the primary abstractions/models used by SMAC (*e.g.*Object representations for Instances, Target Algorithm Configurations & methods for executing algorithms,...). 90% of the code that SMAC uses lives in this library. It also contains code for converting the data from these abstractions into input needed to build the model. These are shipped with SMAC in the `aclib-src.jar` file.

Random Forests The Random Forest model code. The sources are included in `fastrf-src.jar`.

The scope of this document governs only the first two projects. At the time of writing the **Automatic Configurator Library** code base is in good shape, while the **SMAC** code base suffers from two key problems:

- A sizable portion of the 30 or so classes is only to support the porting process of SMAC from MATLAB to Java.
- The bulk of the code necessary to run SMAC lives in three classes `AbstractAlgorithmFramework`, `SequentialModelBasedAlgorithmConfiguration` and finally, `AutomaticConfigurator`. Each of these three classes has problems with poor cohesion (*i.e.*They are all basically doing too much, and could easily be broken up into smaller classes).

As most of the SMAC code is in the **Automatic Configurator Library**, these issues are hardly fatal, and will most likely just be suprising at how disjoint the code bases seem. While the **Automatic Configurator Library** is relatively stable, the **SMAC** portion of the code will be refactored over the coming months to clean up many of its deficiencies.

6.2 Class Overview

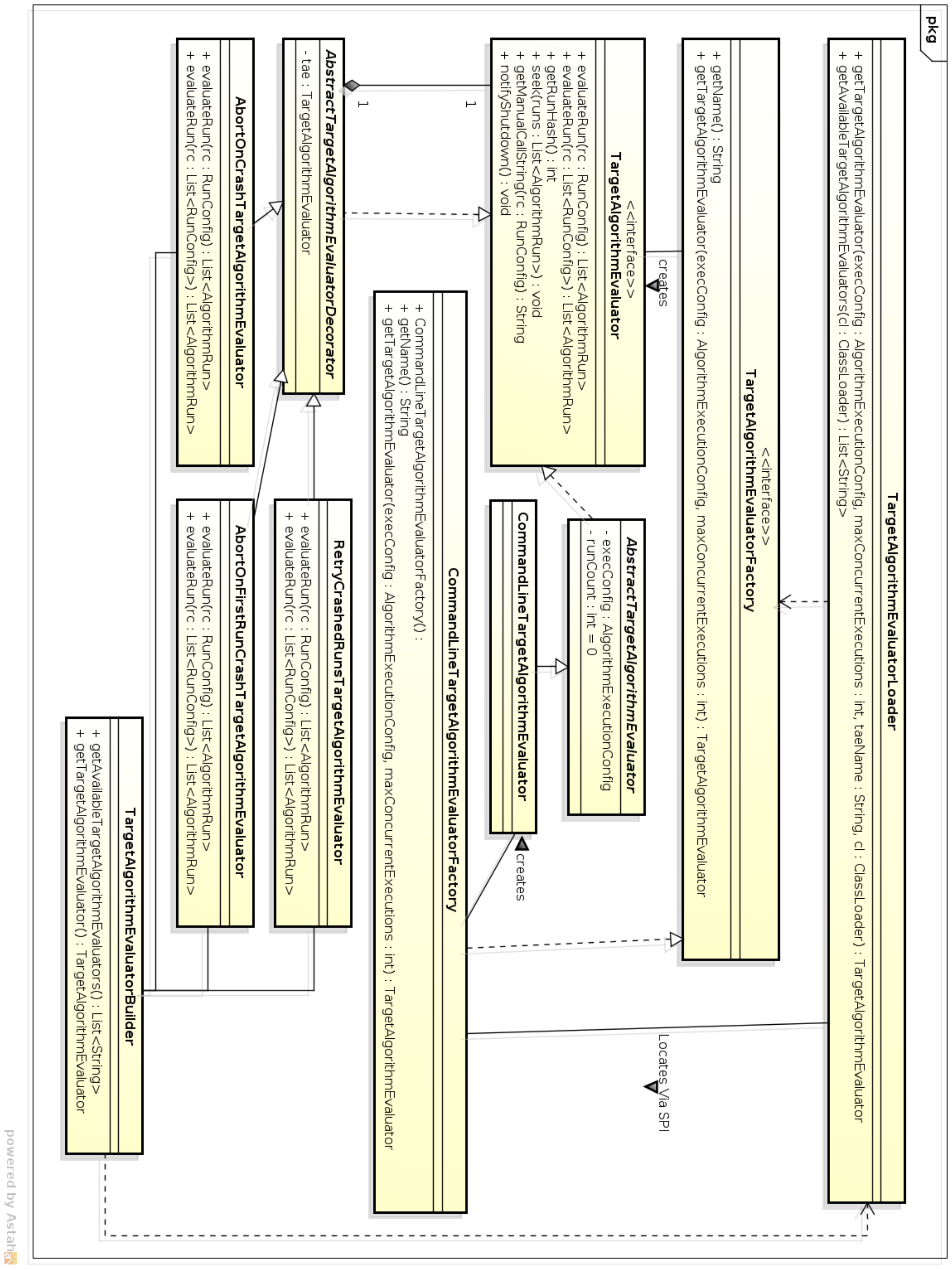
The most important classes to SMAC are as follows:

Automatic Configurator Library Classes	
Name	Description
`AbstractOptions`	Base class for creating new options for SMAC. While not important in and of itself, you will generally be implementing or modifying one of it's subtypes to implement options.
`AlgorithmRun`	Interface that represents the results of a target algorithm run. These are created by a `TargetAlgorithmEvaluator`. Outside of the `TargetAlgorithmEvaluator` these classes are generally immutable.
`AlgorithmExecutionConfig`	Immutable object containing the information required to invoke a target algorithm.
`InstanceSeedGenerator`	Interface that gets seeds for a `ProblemInstance`. These objects are constructed by `ProblemInstanceHelper`
`ModelBuilder`	Interface whose implementations should result in a constructed model.
`OverallObjective`	Aggregates many `RunObjective` values under some statistic (*e.g.*mean), to produce a value to be optimized.
`ParamConfiguration`	Class that represents a specific setting of the target algorithm's parameters. This class also implements the `Map` interface, though does not support all the required operations. The ID associated with is object, is used only for logging and should not be used in the code. Finally although this class is not immutable the general life cycle is that the object is created, given specific values, and then never changed again. In future this may be augmented with the ability to prevent writes. These objects are always constructed via the `ParamConfigurationSpace`.
`ParamConfigurationSpace`	(Almost immutable) class that represents the entire configuration space of a target algorithm. This class is constructed with the **Algorithm Parameter File** described in section 4.4. This class also contains the specifics of each parameter (*e.g.*domains, defaults, etc...). Currently the Random object used is the only portion that is mutable, and this will change in the future.
`ProblemInstance`	Immutable class that represents a specific problem instance, constructed by `ProblemInstanceHelper`.
`ProblemInstanceSeedPair`	Immutable class that represents a problem instance and seed. Decisions of which seed to use when scheduling a run are made in `RunHistory`.
`RunConfig`	Immutable class that represents a problem instance seed pair, and configuration to execute.
`RunHistory`	Interface that saves all the runs performed, and allows various queries against this information.
`RunObjective`	Converts an `AlgorithmRun` into a scalar value for optimization
`SanitizedModelData`	Converts the run data into a format to use when building the model. Other things such as PCA, and other data filtering are done here. This interface and mechanism will likely be refactored in the future as it is brittle at the moment.
`SeedableRandomSingleton`	A global random object whose existence is a convincing case that Singleton's are Anti-Patterns. This will, thankfully, go the way of the dodo bird at some point.
`StateFactory`	Interface that constructs `StateSerializer` & `StateDeserializer` to manage saving and restoring state respectively.
`TargetAlgorithmEvaluator`	Interface whose implementations should be able to run the algorithm (*i.e.* Implementations should convert `RunConfig` objects to `AlgorithmRun` objects). See section 6.3 for more information.

SMAC Library Classes	
Name	Description
`AutomaticConfigurator`	Constructs all objects necessary to execute SMAC (SMAC entrypoint)
`AbstractAlgorithmFramework`	*Non-abstract* class that provides a default Automatic Configurator (ROAR)
`SequentialModelBasedAlgorithmConfiguration`	Class that subtypes `AbstractAlgorithmFramework` and implements the methods required for SMAC
`Validator`	Performs Validation of selected configurations
`ValidatorExecutor`	Entry point to stand alone validation utility

6.3 Target Algorithm Evaluator

The **Target Algorithm Evaluator** subsystem is the part of the code you will be modifying if you would like to change how target algorithms are run. On the next page is a UML class diagram showing most of how this part of the code works.


pkg
TargetAlgorithmEvaluatorLoader
+ getTargetAlgorithmEvaluator(execConfig : AlgorithmExecutionConfig, maxConcurrentExecutions : int, taeName : String, cl : ClassLoader) : TargetAlgorithmEvaluator
+ getAvailableTargetAlgorithmEvaluators(cl : ClassLoader) : List<String>
<<interface>>
TargetAlgorithmEvaluatorFactory
+ getName() : String
+ getTargetAlgorithmEvaluator(execConfig : AlgorithmExecutionConfig, maxConcurrentExecutions : int) : TargetAlgorithmEvaluator
creates
<<interface>>
TargetAlgorithmEvaluator
+ evaluateRun(rc : RunConfig) : List<AlgorithmRun>
+ evaluateRun(rc : List<RunConfig>) : List<AlgorithmRun>
+ getRunHash() : int
+ seek(runs : List<AlgorithmRun>) : void
+ getManualCallString(rc : RunConfig) : String
+ notifyShutdown() : void
1
1
AbstractTargetAlgorithmEvaluatorDecorator
- tae : TargetAlgorithmEvaluator
AbortOnCrashTargetAlgorithmEvaluator
+ evaluateRun(rc : RunConfig) : List<AlgorithmRun>
+ evaluateRun(rc : List<RunConfig>) : List<AlgorithmRun>
CommandLineTargetAlgorithmEvaluatorFactory
+ CommandLineTargetAlgorithmEvaluatorFactory() :
+ getName() : String
+ getTargetAlgorithmEvaluator(execConfig : AlgorithmExecutionConfig, maxConcurrentExecutions : int) : TargetAlgorithmEvaluator
AbstractTargetAlgorithmEvaluator
- execConfig : AlgorithmExecutionConfig
- runCount : int = 0
CommandLineTargetAlgorithmEvaluator
creates
RetryCrashedRunsTargetAlgorithmEvaluator
+ evaluateRun(rc : RunConfig) : List<AlgorithmRun>
+ evaluateRun(rc : List<RunConfig>) : List<AlgorithmRun>
AbortOnFirstRunCrashTargetAlgorithmEvaluator
+ evaluateRun(rc : RunConfig) : List<AlgorithmRun>
+ evaluateRun(rc : List<RunConfig>) : List<AlgorithmRun>
TargetAlgorithmEvaluatorBuilder
+ getAvailableTargetAlgorithmEvaluators() : List<String>
+ getTargetAlgorithmEvaluator() : TargetAlgorithmEvaluator
Locates Via SPI
powered by Astah

Once constructed, the `TargetAlgorithmEvaluator` interface is simple, it simply needs to return a new `AlgorithmRun` object, for each `RunConfig` object passed as input, and in the same order, via the `TargetAlgorithmEvaluator.evaluateRun()` method. The construction of these objects is where the complexity lies and so here is a run through of the construction.

1. When the code starts up, SMAC requests a specific Target Algorithm Evaluator (using some globally unique String as a key), from `TargetAlgorithmEvaluatorBuilder.getTargetAlgorithmEvaluator`
2. This invokes the similarly named method in `TargetAlgorithmEvaluatorLoader`, which uses SPI (see 6.4 for more information on SPI) to find the `TargetAlgorithmEvaluatorFactory` whose `getName()` method returns the matching string. The name MUST NOT have any white space. For reference, the `CommandLineTargetAlgoirthmEvaluatorFactory` returns `CLI`.
3. When an match is found, a no argument constructor (in the diagram this is shown under the `CommandLineTargetAlg` class) is invoked.
4. Next the `getTargetAlgorithmEvaluator()` method is invoked which in the above diagram would return a `CommandLineTargetAlgorithmEvaluator`
5. With this new instance in hand, the `TargetAlgorithmEvaluatorBuilder` then wraps this object with various decorators (*e.g.*RetryCrashedRunTargetAlgorithmEvaluator) depending on the options supplied (not-shown).

The use of SPI allows new implementations to be created without modifying the existing SMAC code, and requires less mantinence to update to newer versions of SMAC. Unfortunately at the time of writing there are two limitations to keep in mind with this approach.

1. You cannot supply options to the user to configure your `TargetAlgorithmEvaluator`.
2. You cannot use this method to add new decorators.

Neither of these seems significant at the current time. If a new decorator is needed, you can hard code the base implementation and return a wrapped instance of it (*i.e.*Create a new `TargetAlgorithmEvaluatorFactory` that returns a wrapped instance of an existing `TargetAlgorithmEvaluator`). Configuration of the `TargetAlgorithmEvaluator` can be done via files at this point.

When using the SPI approach you are strongly encouraged to also implement **Plugin Versioning**; see Section 6.4.

6.4 Plugin Versioning

Any plug-ins or changes to SMAC should contain an implementation of `VersionInfo`, and the implementor should be labelled as a provider of `VersionInfo` via SPI [13].

In essence this interface simply has two getter methods `getProductName()` and `getVersion()`. If everything is done correctly when SMAC starts up you should see the product name and version printed in the logs.

Example:

[13]SPI is the Service Provider Interface, see SPI on Wikipedia (http://en.wikipedia.org/wiki/Service_provider_interface) as well as this utility which simplifies the process drastically (http://code.google.com/p/spi/)

```
[INFO ] Version of Automatic Configurator Library is v1.00.04dev-307
[INFO ] Version of Random Forest Library is v1.04.01dev-50
[INFO ] Version of SMAC is v2.00.01dev-318
[INFO ] Version of Surrogate is v1.01dev-227
```

This can make debugging and managing reproducability much easier.

6.5 Run Hash Codes

A Run Hash Code is a sequence of hashes that represent which runs were scheduled by SMAC. When calling SMAC using
`./smac --scenarioFile <file> --runHashCodeFile <logfile>`,
SMAC logs all Run Hash Codes to <logfile>. This option allows reading of that log file for subsequent runs to ensure that the exact same set of runs is scheduled. This is primarily of use for developers.

7 Acknowledgements

We are indebted to Jonathan Shen for porting our random forest code from C to Java in preparation for a Java port of all of SMAC. We thank Marius Schneider for many valuable bug reports and suggestions for improvements. Thanks also to Chris Thornton for being a secondary beta tester.

8 References

[1] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011a). Bayesian optimization with censored response data. In *2011 NIPS workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits*. Published online.

[2] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011b). Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, LNCS, pages 507–523.

[3] Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2009). ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306.

9 Appendix

9.1 Return Codes

NOTE: All error conditions for besides 255, are fixed. However in future some exceptions that previously reported 255 may be changed to a non 255 value as needed.

9.2 Version History of Java SMAC

Version 2.00 First Java release of SMAC (this is a port and extension of the original Matlab version).

Value	Error Name	Description
0	**Success**	Everything completed successfully
1	**Parameter Error**	There was a problem with the input arguments or files
2	**Trajectory Divergence**	For some reason SMAC has taken a unexpected path (*e.g.* SMAC executes a run that does not match a run in the **–runHashCodeFile**)
3	**Serialization Exception**	A problem occurred when saving or restoring state
255	**Other Exceptions**	Some other error occurred

9.3 Known Issues

1. Trajectory file does not output standard deviation
2. Using any alias for --showHiddenParameters, --help, or --version as values to other arguments (*e.g.* Setting --runGroupName --help) does not parse correctly (This is unlikely to be fixed until someone complains).
3. Using large parameter values in continuous integral parameters, may cause loss of precision, and or crashes if the values are too big.

9.4 Options Reference

9.4.1 SMAC Options

General Options for Running SMAC

--adaptiveCapping Use Adaptive Capping

> **Aliases:** --adaptiveCapping
>
> **Default Value:** Defaults to true when –intraInstanceObjective is RUNTIME, false otherwise
>
> **Domain:** $\{true, false\}$

--capAddSlack amount to increase computed adaptive capping value of challengers by (post scaling)

> **Aliases:** --capAddSlack
>
> **Default Value:** 1.0
>
> **Domain:** $(0, \infty)$

--capSlack amount to scale computed adaptive capping value of challengers by

> **Aliases:** --capSlack
>
> **Default Value:** 1.3
>
> **Domain:** $(0, \infty)$

--consoleLogLevel default log level of console output (this cannot be more verbose than the logLevel)

> **Aliases:** --consoleLogLevel

Default Value: INFO

Domain: $\{TRACE, DEBUG, INFO, WARN, ERROR, OFF\}$

--countSMACTimeAsTunerTime include the CPU Time of SMAC as part of the tunerTimeout

Aliases: --countSMACTimeAsTunerTime

Default Value: true

Domain: $\{true, false\}$

--doValidation perform validation when SMAC completes

Aliases: --doValidation, --validation

Default Value: true

Domain: $\{true, false\}$

--executionMode execution mode of the automatic configurator

Aliases: --executionMode

Default Value: SMAC

Domain: $\{SMAC, ROAR\}$

--expectedImprovementFunction expected improvement function to use during local search

Aliases: --expectedImprovementFunction

Default Value: EXPONENTIAL

Domain: $\{EXPONENTIAL, SIMPLE\}$

--experimentDir root directory for experiments Folder

Aliases: --experimentDir, -e

Default Value: <current working directory>

--help show help

Aliases: --help, -?, /?, -h

--imputationIterations amount of times to impute censored data when building model

Aliases: --imputationIterations

Default Value: 2

Domain: [0, 2147483647]

--intensificationPercentage percent of time to spend intensifying versus model learning

Aliases: --intensificationPercentage, --frac_rawruntime

Default Value: 0.5

Domain: (0, 1)

--logLevel Log Level for SMAC

Aliases: --logLevel

Default Value: DEBUG

Domain: $\{TRACE, DEBUG, INFO, WARN, ERROR, OFF\}$

--maskInactiveConditionalParametersAsDefaultValue build the model treating inactive conditional values as the default value

Aliases: --maskInactiveConditionalParametersAsDefaultValue

Default Value: true

Domain: $\{true, false\}$

--maxIncumbentRuns maximum number of incumbent runs allowed

Aliases: --maxIncumbentRuns, --maxRunsForIncumbent

Default Value: 2000

Domain: (0, 2147483647]

--numChallengers number of challengers needed for local search

Aliases: --numChallengers, --numberOfChallengers

Default Value: 10

Domain: (0, 2147483647]

--numEIRandomConfigs number of random configurations to evaluate during EI search

Aliases: --numEIRandomConfigs, --numberOfRandomConfigsInEI, --numRandomConfigsInEI, --numberOfEIRandomConfigs

Default Value: 10000

Domain: [0, 2147483647]

--numIterations limits the number of iterations allowed during automatic configuration phase

Aliases: --numIterations, --numberOfIterations

Default Value: 2147483647

Domain: (0, 2147483647]

--numPCA number of principal components features to use when building the model

Aliases: --numPCA

Default Value: 7

Domain: (0, 2147483647]

--numRun number of this run (and seed)

REQUIRED

Aliases: --numRun, --seed

Default Value: 0

Domain: [0, 2147483647]

--optionFile read options from file

Aliases: --optionFile

--optionFile2 read options from file

Aliases: --optionFile2

--restoreStateFrom location of state to restore

Aliases: --restoreStateFrom

Default Value: N/A (No state is being restored)

--restoreStateIteration iteration of the state to restore

Aliases: --restoreStateIteration, --restoreIteration

Default Value: N/A (No state is being restored)

--runGroupName name of subfolder of outputdir to save all the output files of this run to

Aliases: --runGroupName

Default Value: RunGroup-<current date and time>

--runtimeLimit limits the total wall-clock time allowed during the automatic configuration phase

Aliases: --runtimeLimit, --wallClockLimit

Default Value: 2147483647

Domain: (0, 2147483647]

--seedOffset offset of numRun to use from seed (this plus --numRun should be less than LONG_MAX)

Aliases: --seedOffset

Default Value: 0

--showHiddenParameters show hidden parameters that no one has use for, and probably just break SMAC (no-arguments)

Aliases: --showHiddenParameters

--stateDeserializer determines the format of the files that store the saved state to restore

Aliases: --stateDeserializer

Default Value: LEGACY

Domain: $\{NULL, LEGACY\}$

--stateSerializer determines the format of the files to save the state in

Aliases: --stateSerializer

Default Value: LEGACY

Domain: $\{NULL, LEGACY\}$

--totalNumRunsLimit limits the total number of target algorithm runs allowed during the automatic configuration phase

Aliases: --totalNumRunsLimit, --numRunsLimit, --numberOfRunsLimit

Default Value: 9223372036854775807

Domain: (0, 9223372036854775807]

--treatCensoredDataAsUncensored builds the model as-if the response values observed for cap values, were the correct ones [NOT RECOMMENDED]

Aliases: --treatCensoredDataAsUncensored

Default Value: false

Domain: $\{true, false\}$

-v print version and exit

Aliases: -v, --version

9.4.2 Scenario Options

Standard Scenario Options for use with SMAC. In general consider using the –scenarioFile directive to specify these parameters and Algorithm Execution Options

--checkInstanceFilesExist check if instances files exist on disk

Aliases: --checkInstanceFilesExist

Default Value: false

Domain: $\{true, false\}$

--cutoffTime CPU time limit for an individual target algorithm run

REQUIRED

Aliases: --cutoffTime, --cutoff_time

Default Value: 0.0

Domain: $(0, \infty)$

--instanceFeatureFile file that contains the all the instances features

Aliases: --instanceFeatureFile, --feature_file

--instanceFile file containing a list of instances to use during the automatic configuration phase (see Instance File Format section of the manual)

REQUIRED

Aliases: --instanceFile, -i, --instance_file, --instance_seed_file

Default Value: null

--interInstanceObj objective function used to aggregate over multiple instances (that have already been aggregated under the Intra-Instance Objective)

Aliases: --interInstanceObj, --inter_instance_obj

Default Value: MEAN

Domain: $\{MEAN, MEAN1000, MEAN10\}$

--intraInstanceObj objective function used to aggregate multiple runs for a single instance

REQUIRED

Aliases: --intraInstanceObj, --overallObj, --overall_obj, --intra_instance_obj

Default Value: null

Domain: $\{MEAN, MEAN1000, MEAN10\}$

--outputDirectory Output Directory

Aliases: --outputDirectory, --outdir

Default Value: <current working directory>/smac-output

--runObj per target algorithm run objective type that we are optimizing for

REQUIRED

Aliases: --runObj, --run_obj

Default Value: null

Domain: $\{RUNTIME, QUALITY\}$

--scenarioFile scenario file

Aliases: --scenarioFile

--testInstanceFile file containing a list of instances to use during the validation phase (see Instance File Format section of the manual)

REQUIRED

Aliases: --testInstanceFile, --test_instance_file, --test_instance_seed_file

Default Value: null

--tunerTimeout limits the total cpu time allowed between SMAC and the target algorithm runs during the automatic configuration phase

Aliases: --tunerTimeout

Default Value: 2147483647

Domain: [0, 2147483647]

-p File containing algorithm parameter space information (see Algorithm Parameter File in the Manual)

REQUIRED

Aliases: -p, --paramFile, --paramfile

Default Value: null

9.4.3 Algorithm Execution Options

Options related to running the target algorithm

--abortOnCrash treat algorithm crashes as an ABORT (Useful if algorithm should never CRASH). NOTE: This only aborts if all retries fail.

Aliases: --abortOnCrash

Default Value: false

Domain: $\{true, false\}$

--abortOnFirstRunCrash if the first run of the algorithm CRASHED treat it as an ABORT, otherwise allow crashes.

Aliases: --abortOnFirstRunCrash

Default Value: true

Domain: $\{true, false\}$

--algoExec command string to execute algorithm with

REQUIRED

Aliases: --algoExec, --algo

Default Value: null

--deterministic treat the target algorithm as deterministic

Aliases: --deterministic

Default Value: false

Domain: $\{true, false\}$

--execDir working directory to execute algorithm in

REQUIRED

Aliases: --execDir, --execdir

Default Value: null

--logAllCallStrings log every call string

Aliases: --logAllCallStrings

Default Value: false

Domain: $\{true, false\}$

--logAllProcessOutput log all process output

Aliases: --logAllProcessOutput

Default Value: false

Domain: $\{true, false\}$

--numConcurrentAlgoExecs maximum number of concurrent target algorithm executions

Aliases: --numConcurrentAlgoExecs, --maxConcurrentAlgoExecs, --numberOfConcurrentAlgoExecs

Default Value: 1

--retryTargetAlgorithmRunCount number of times to retry an algorithm run before eporting crashed (NOTE: The original crashes DO NOT count towards any time limits, they are in effect lost). Additionally this only retries CRASHED runs, not ABORT runs, this is by design as ABORT is only for cases when we shouldn't bother further runs

Aliases: --retryTargetAlgorithmRunCount

Default Value: 0

Domain: [0, 2147483647]

--runHashCodeFile file containing a list of run hashes one per line: Each line should be: "Run Hash Codes: (Hash Code) After (n) runs". The number of runs in this file need not match the number of runs that we execute, this file only ensures that the sequences never diverge. Note the n is completely ignored so the order they are specified in is the order we expect the hash codes in this version. Finally note you can simply point this at a previous log and other lines will be disregarded

Aliases: --runHashCodeFile

--targetAlgorithmEvaluator Target Algorithm Evaluator to use when making target algorithm calls

Aliases: --targetAlgorithmEvaluator, --tae

Default Value: CLI

--targetAlgorithmEvaluatorSearchPath location to look for other target algorithm evaluators [See manual but generally you can ignore this]

Aliases: --targetAlgorithmEvaluatorSearchPath, --taeSP

Default Value: <current working directory>/plugins/ amoung others

9.4.4 Random Forest Options

Options used when building the Random Forests

--fullTreeBootstrap bootstrap all data points into trees

Aliases: --fullTreeBootstrap

Default Value: false

Domain: $\{true, false\}$

--ignoreConditionality ignore conditionality for building the model

Aliases: --ignoreConditionality

Default Value: false

Domain: $\{true, false\}$

--logModel store response values in log-normal form

Aliases: --logModel

Default Value: true

Domain: $\{true, false\}$

--minVariance minimum allowed variance

Aliases: --minVariance

Default Value: 1.0E-14

Domain: $(0, \infty)$

--numTrees number of trees to create in random forest

Aliases: --numTrees, --nTrees, --numberOfTrees

Default Value: 10

Domain: (0, 2147483647]

--penalizeImputedValues treat imputed values that fall above the cutoff time, and below the penalized max time, as the penalized max time

Aliases: --penalizeImputedValues

Default Value: false

Domain: $\{true, false\}$

--preprocessMarginal build random forest with preprocessed marginal

Aliases: --preprocessMarginal

Default Value: true

Domain: $\{true, false\}$

--ratioFeatures ratio of the number of features to consider when splitting a node

Aliases: --ratioFeatures

Default Value: 0.8333333333333334

Domain: (0, 1]

--shuffleImputedValues shuffle imputed value predictions between trees

Aliases: --shuffleImputedValues

Default Value: false

Domain: $\{true, false\}$

--splitMin minimum number of elements needed to split a node

Aliases: --splitMin

Default Value: 10

Domain: [0, 2147483647]

--storeDataInLeaves store full data in leaves of trees

Aliases: --storeDataInLeaves

Default Value: false

Domain: $\{true, false\}$

9.4.5 Validation Options

Options that control validation

--maxTimestamp maximimum relative timestamp in the trajectory file to configure against. -1 means auto-detect

> **Aliases:** --maxTimestamp
>
> **Default Value:** Auto Detect
>
> **Domain:** $[0, \infty) \bigcup \{-1\}$

--minTimestamp minimum relative timestamp in the trajectory file to configure against.

> **Aliases:** --minTimestamp
>
> **Default Value:** 0.0
>
> **Domain:** $[0, \infty)$

--multFactor base of the geometric progression of timestamps to validate (timestamps selected are: maxTime$\times$multFactor^{-n} where n is $\{1, 2, 3, 4...\}$ while timestamp $\geq$ minTimestamp)

> **Aliases:** --multFactor
>
> **Default Value:** 2.0
>
> **Domain:** $(0, \infty)$

--numSeedsPerTestInstance number of test seeds to use per instance during validation

> **Aliases:** --numSeedsPerTestInstance, --numberOfSeedsPerTestInstance
>
> **Default Value:** 1000
>
> **Domain:** (0, 2147483647]

--numTestInstances number of instances to test against (will execute min of this, and number of instances in test Instance File)

> **Aliases:** --numTestInstances, --numberOfTestInstances
>
> **Default Value:** 2147483647
>
> **Domain:** (0, 2147483647]

--numValidationRuns approximate number of validation runs to do

> **Aliases:** --numValidationRuns, --numberOfValidationRuns
>
> **Default Value:** 1000
>
> **Domain:** (0, 2147483647]

--validateOnlyLastIncumbent validate only the last incumbent found

Aliases: --validateOnlyLastIncumbent

Default Value: true

Domain: $\{true, false\}$

--validationHeaders put headers on output CSV files for validation

Aliases: --validationHeaders

Default Value: true

Domain: $\{true, false\}$

--validationRoundingMode selects whether to round the number of validation (to next multiple of numTestInstances

Aliases: --validationRoundingMode

Default Value: UP

Domain: $\{UP, NONE\}$