Manual for SMAC version v2.06.01-master

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

October 19, 2013

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 will 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 [1] or newer to run.

SMAC is primarily intended to run on Unix like platforms, but now includes start up scripts so that it can run on Windows. In all the examples below you should add `.bat` to the end of every executable, for instance `./smac --scenario-file scen.txt --seed 1` becomes `smac.bat --scenario-file scen.txt --seed 1`.

[1]Sun Java version 1.6.0_23 or later recommended

Most of the included scenarios (in `./example_scenarios/` require ruby and Linux 32-bit libraries to run. The scenarios in `./example_scenarios/analytic/` use optimize functions internal to smac and are completely cross platform. There is one scenario for windows available currently in`example_scenarios\saps\SAPS-scenario-windows.txt`.

1.3 Version

This version of the manual is for SMAC v2.06.01-master-622.

Project	Version	Commit	Dirty Flag
ACLib	v2.06.01-master-648	1a58581fbab7e770e7e6994ee48ed5f93a8e376c	0
SMAC	v2.06.01-master-622	e32716a31d3b27a38810ef48906ae50620d5a0c9	0

NOTE: For non-`master` builds these commits may not contain everything in the build. (*i.e.*,non-`master` builds can be built with uncommitted changes). If the dirty flag is 0 that means the commit contains this exact copy, 1 means there were some uncommitted changes, and something else means some other error occurred when we tried to generate this.

2 Differences Between SMAC and ParamILS

There are a number of 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 4.5 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 **--use-cpu-time-in-tunertime** and **--wallclock-limit**. See Section 3.6 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 3.8 for how to use the state restoration feature. Section 6.2 describes the file format for saved states.
- **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** signalling 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 **--abort-on-first-run-crash** is set and the first run returns **CRASHED**. Additionally, the wrapper can also return additional data to SMAC that is associated with the run [2]. For more information see Section 5.1.2.

- **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 preferred format is now `.csv`.

3 Commonly Used Options

3.1 Running SMAC

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

```
./smac --scenario-file <file> --seed 1
```

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 **--seed** argument controls the seed and names of output files (to support parallel independent runs). The **--seed-offset** argument lets you keep the output folders names simple while varying the actual seed of SMAC.

3.2 Testing the Wrapper

SMAC includes a method of Testing Algorithm Execution, via the `algotest` utility. It takes the required scenario options [3]

For example:

```
./algotest --scenario-file <scenario> --instance <instance>
--config <config string> -P[name]=[value] -P[name]=[value]...
```

Some parameters deserve special mention:

1. The config string syntax is a single string with "-name='value' " ... you can also specify `RANDOM` which will generate a random configuration or `DEFAULT` which will generate the default configuration.

2. The `-P` parameters are optional and allow overriding specific values in the configuration (this is useful primarily for RANDOM and DEFAULT, to allow you to set certain values). To set the `sortalgo` parameter to `merge` you would specify `Psortalgo=merge`.

[2]This data will be saved in the run and results file (Section 6.2) that is used in state saving

[3]Unfortunately it cannot read scenario files currently

3.3 Verifying the Scenario

SMAC includes a utility that allows you to test the scenario. It is currently `BETA` but does a bit more sanity checks than SMAC will normally do.

For example:

```
./verify-scenario --scenarios ./scenarios/*.txt --verify-instances true
```

The utility has some limitations however:

1. It currently does not check test instances
2. Scenario files can specify non-scenario options in SMAC (and some of the example scenarios in fact do), this utility is not aware of them, and will report an error.

3.4 ROAR Mode

```
./smac --scenarioFile <file> --exec-mode ROAR --seed 1
```

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.

3.5 Adaptive Capping

```
./smac --scenarioFile <file> --adaptive-capping true --seed 1
```

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 **--algo-cutoff-time**.
NOTE: Adaptive Capping should only be used when the **--run-obj** is RUNTIME. Adaptive capping can drastically improve SMAC's performance for scenarios with a large difference between **--algo-cutoff-time** and the runtime of the best-performing configurations.

3.6 Wall-Clock Limit

```
./smac --scenario-file <file> --wallclock-limit <seconds> --seed 1
```

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 **--tunertime-limit** or other options that limit the duration of the configuration run; whichever termination criterion is reached first triggers termination.

3.7 Change Initial Incumbent

```
./smac --scenario-file <file> --seed 1 --initial-incumbent <config string>
```

SMAC offers the option to specify the initial incumbent, and by default uses the default configuration specified in the parameter file. The argument to **--initial-incumbent** follows the same conventions as in Section 3.2.

3.8 State Restoration

```
./smac --scenario-file <file> --restore-scenario <dir>
 --seed 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 **--tunertime-limit** is usually a safe bet, however changing something central (such as **--run-obj**) would not be.

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

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

3.9 Warm-Starting the Model

```
./smac --scenario-file <file> --seed 0 --warmstart <foldername>
```

Using the same state data as in Section 3.8, you can also just choose to warm-up the model with previous runs. Instead of the **--restore-scenario** option use **--warmstart** instead. SMAC will operate normally, but when building the model the above data will also be used. Please keep in mind the following.

NOTE: If the execution mode is ROAR, this option has no effect.

WARNING: Due to design limitations of the state restoration format in this version of SMAC you cannot / should not have any differences between the instance distribution used to warmstart the model, and the instance distribution we are configuring against. In the best case you will simply get a random exception at some point (perhaps a `NullPointerException`), and in the worst case it will just load the model with junk.

TIP: The included state-merge utility allows you to easily merge a bunch of different runs of SMAC into one state that you can use for a warm start.

3.10 Named Rungroups

```
./smac --scenario-file <file> --rungroup <foldername> --seed 1
```

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

3.11 More Options

By default SMAC only displays BASIC usage options, other options are INTERMEDIATE, ADVANCED, and DEVELOPER. Be warned that there are a bunch of options and some of the more advanced and developer options may cause SMAC to perform very poorly.

```
./smac --help-level INTERMEDIATE
```

3.12 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 as in the following example:

```
./smac --scenario-file <file> --num-seeds-per-test-instance 50
```

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

3.12.1 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 **--num-test-instances** 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 --scenario-file <file> --num-test-instances 10
```

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

3.12.2 Disabling Validation

Validation can be skipped alltogether as follows:

```
./smac --scenario-file <file> --seed 1 --validation false
```

3.12.3 Standalone Validation

SMAC also includes a method of validating configurations outside of a smac run. You can supply a configuration using the **--configuration** option. All scenario options are applicable to the standalone validator, but check the usage screen to see all the options available NOTE: Some options while present are not applicable for validation but are presented anyway.

Here is an example call:

```
./smac-validate --scenario-file <file> --num-validation-runs 100
     --configuration <config string> --cli-cores 8 --seed 1
```

Usage notes for the offline validation tool:

1. This validates against the test set only; the training instance set is not used.

2. By default this outputs into the current directory; you can change the output directory with the option **--rungroup**.

3. You can also validate against a trajectory file issued by **--trajectory-file** option.

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 **–experiment-dir** 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 [4]. The Scenario Files used in SMAC are backwards compatible with ParamILS and the name of option names here reflect that[5]. 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 5.1. 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...). SMAC will attempt to *minimize* this objective. 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:

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

[5]Every option name listed here is in fact an alias for an existing option listed in the section 10.5 and it is entirely possible to use SMAC without using Scenario Files.

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.

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 a file with the features for the instances in the **instance_file** and possibly the **test_instance_file** [6]. The format of this file 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[7]. Before specifying the formats it is important to note the three kinds of information that are specified with instances [8].

Instance Name The name of the instance that was selected. This should be meaningful to the target algorithm we are configuring [9].

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.

[6]The Validator will load features into memory for test instances if they exist.

[7]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.

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

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

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).

4.3 Feature File Format

The **feature_file** specifies features that are to be used for instances. 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 `double` 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* (*i.e.* it must contain atleast one letter) feature name must be specified.

4.4 Algorithm Parameter File

The PCS format requires each line to contain one of the following 3 clauses, or only whitespace/comments.

- **Parameter Declaration Clauses** specify the names of parameters, their domains, and default values.
- **Conditional Parameter Clauses** specify when a parameter is active/inactive.
- **Forbidden Parameter Clauses** specify when a combination of parameter settings is illegal.

Comments are allowed throughout the file; they begin with a #, and run to the end of a line.

4.5 Parameter Declaration Clauses

The PCS format supports two types of parameters: categorical and numeric.

4.5.1 Categorical parameters

Categorical parameters take one of a finite set of values. Each line specifying a categorical parameter should be of the form:

```
<parameter_name> {<value 1>, ..., <value N>} [<default value>]
```

where '`<default value>`' has to be one of the set of possible values.

Example 1:

```
decision-heuristic {1,2,3} [1]
```

This means that the parameter '`decision-heuristic`' can be given one of three possible values, with the default assignment being '`1`'.

Example 2:

```
@1:loops {common,distinct,shared,no}[no]
```

In this example, the somewhat cryptic parameter name '`@1:loops`' is perfectly legal; the only forbidden characters in parameter names are spaces, commas, quotes, and parentheses. Categorical parameter values are also strings with the same restrictions; in particular, there is no restriction for categorical parameter values to be numbers.

Example 3:

```
DS {TinyDataStructure, FastDataStructure}[TinyDataStructure]
```

As this example shows, the parameter values can even be Java class names (to be used, e.g., via reflection).

Example 4:

```
random-variable-frequency {0, 0.05, 0.1, 0.2} [0.05]
```

Finally, as this example shows, numerical parameters can trivially be treated as categorical ones by simply discretizing their domain (selecting a subset of reasonable values).

4.5.2 Numerical parameters

Numerical parameters (both real and integer) are specified as follows:

```
<parameter_name> [<min value>, <max value>] [<default value>] [i] [l]
```

The trailing '`i`' and/or trailing '`l`' are optional. The '`i`' means the parameter is an integer parameter, and the '`l`' means that the parameter domain should be log-transformed for optimization (see Examples 3 and 4 below).

Example 1:

```
sp-rand-var-dec-scaling [0.3, 1.1] [1]
```

Parameter `sp-rand-var-dec-scaling` is real-valued with a default value of 1, and we can choose values for it from the (closed) interval `[0.3, 1.1]`. Note that there may be other parameter values outside this interval that are in principle legal values for the parameter (e.g., your solver might accept any positive floating point value for the parameter). What you specify here is the range that automated configuration procedures should search (i.e., a range you expect a priori to contain good values); of course, every value in the specified range must be legal. There is a tradeoff in choosing the best range size; see Section **??** for some tips on defining a 'good' parameter space.

Example 2:

```
mult-factor [2, 15] [5]i
```

Parameter `mult-factor` is integer-valued, takes any integer value between `2` and `15` (inclusive), and has a default value of 5. Technically, one could also specify this as a categorical parameter with possible values {`2,3,4,5,6,7,8,9,10,11,12,13,14,15`}. However, categorical parameters are not ordered, and using an integer parameter allows the configuration procedure to make use of the natural order relation (this is useful since, a priori, we expect close-by values to yield similar performance).

Example 3:

```
DLSc [0.00001, 0.1] [0.01]l
```

Parameter `DLSc` is real-valued with a default value of `0.01`, and we can choose values for it from the (closed) interval `[0.00001, 0.1]`. The trailing '`l`' denotes that this parameter naturally varies on a log scale. If we were to discretize the parameter, a natural choice would be {`0.00001, 0.0001, 0.001, 0.01, 0.1`}. That means, a priori the distance between parameter values `0.001` and `0.01` is identical to that between `0.01` and `0.1` (after a $\log_{10}$ transformation, `0.001`, `0.01`, and `0.1` become -3, -2, and -1, respectively). We express this natural variation on a log scale by the '`l`' flag. See Section **??** for further tips on transformations.

Example 4:

```
first-restart [10, 1000] [100]il
```

Parameter `first-restart` is integer-valued with a default value of `100`, and we can choose values for it from the (closed) interval `[10, 1000]`. It also varies naturally on a logarithmic scale. For example, due to this logarithmic scale, after the transformation drawing a uniform random value of `first-restart` will yield a number below `100` half the time.

Restrictions

- Numerical integer parameters must have their lower and upper bounds specified as integers, and the default must also be an integer.
- The bounds for parameters with a log scale must be strictly positive.

4.6 Conditional Parameter Clause

Depending on the instantiation of some 'higher-level' parameters, certain 'lower-level' parameters may not be active. For example, the subparameters of a heuristic are not important (i.e., active) if the heuristic is not selected. All parameters are considered to be active by default, and conditional parameter clauses express under which conditions a parameter is not active. The syntax for conditional parameter clauses is as follows:

```
<child name> | <parent name> in {<parent val1>, ..., <parent valK>}
```

This can be read as "The child parameter `<child name>` is only active if the parent parameter `<parent name>` takes one of the K specified values." Parameters that are not listed as a child parameter in any conditional parameter clause are always active. A parameter can also be listed as a child in multiple conditional parameter clauses, and it is only active if the conditions of each such clause are met.

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

4.7 Forbidden Parameter Clauses

Forbidden Parameters are combinations of parameter values which are invalid (e.g., a certain data structure may be incompatible with a lazy heuristic that does not update the data structure, resulting in incorrect algorithm behaviour). Configuration methods should never try to run an algorithm with a forbidden parameter configuration. The syntax for forbidden parameter combinations is as follows:

```
{<parameter name 1>=<value 1>, ..., <parameter name N>=<value N>}
```

Example:

```
DSF {DataStructure1, DataStructure2, DataStructure3}[DataStructure1]
PreProc {NoPreProc, SimplePreproc, ComplexPreproc}[ComplexPreproc]
{DSF=DataStructure2, PreProc=ComplexPreproc}
{DSF=DataStructure3, PreProc=SimplePreproc}
{DSF=DataStructure3, PreProc=ComplexPreproc}
```

In this example, there are different data structures and different simplifications. `DataStructure2` is incompatible with `ComplexPreproc`, and `DataStructure2` is incompatible with both `SimplePreproc` and `ComplexPreproc`. Note that the default parameter setting is not allowed to contain a forbidden combination of parameter values.

5 Wrappers

5.1 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 5.3.

5.1.1 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 reproducibility). "-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.

5.1.2 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[10]:

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

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 historical and serves as a check that the algorithm is executing correctly by outputting whether the instance in question is satisfiable or not. See the **--verify-sat** option for information on how to utilize this feature.
NOTE: SMAC by default crashes if the wrapper ever reports SAT and UNSAT for the same instance across runs. Occasionally edge cases in exposed parameters are tripped and turn a solver buggy, and so this safe guard exists to help detect if this is occurring. To change this behaviour use the **--check-sat-consistency** and **--check-sat-consistency-exception** options.

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.

[10]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+(?:([Ff]or)—([oO]f))\s+(?:(HAL)—(ParamILS)—(SMAC)—([Tt]his [wW]rapper))**

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 **--retry-crashed-count** and **--abort-on-crash** options, to mitigate these.

In other files or the log you may see the following following additional types used. **RUNNING** which signifies a result that is currently in the middle of being run, and **KILLED** which signifies that SMAC internally decided to terminate the run before it finished. These are internal values only, and wrappers are NOT permitted to output these values. If these values are reported by the wrapper, it will be treated as if the run had status **CRASHED**.

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 degradation when this amount is heavily under reported [11].

NOTE:The **runtime** should always be strictly less than the requested **cutoff_time** when reporting **SAT** or **UNSAT**. The runtime must be strictly greater than zero (and not NaN).

If an algorithm reports **TIMEOUT** or **CRASHED** the algorithm can report the actual CPU time used, and SMAC will treat it correctly as a timeout for optimization purposes, but count the actual time for **--tunertime-limit** purposes.

runlength A domain specific measure of how far the algorithm progressed. This value must be from the set: $-1 \cup [0, +\infty]$.

quality A domain specific measure of the quality of the solution. This value needs to be from the set: $(-\infty, +\infty)$.

NOTE: In certain cases, such as when using log transforms in the model, this value must be: $(0, +\infty)$.

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.

additional rundata A string (not containing commas, or newline characters) that will be associated with the run as far as SMAC is concerned. This string will be saved in run and results file (Section 6.2). NOTE:**additional rundata** is not compatible with ParamILS at time of writing, and so wrappers should not include this or the preceding comma if they wish to be compatible.

All fields except for **additional rundata** are mandatory. If the field is not applicable for your scenario a 0 can be substituted.

5.2 Wrapper Output Semantics

As SMAC is entirely insulated from the target algorithm execution by the wrapper it is up to the wrapper to ensure that constraints with respect to the cutoff and runlength are enforced. Occasionally wrappers may not properly enforce these constraints and SMAC will need to somehow handle these cases. The following table

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

outlines how SMAC transforms these values and details what value is used in various parts of SMAC. In future versions some parts of this table may in fact change, and so it is best to ensure that your wrapper is well behaved.

NOTE: The cutoff time in the table is the amount of time SMAC schedules the run for, the scenario cutoff time is denoted as κ_{max}.

solved	**cutoff** (κ)	**runtime** (r)	Tuner Time	PAR10 Score	Model
*	*	$(-\infty, 0)$	**EXCEPTION THROWN**		
`ABORT`	*	$[0, \infty)$	**EXCEPTION THROWN**		
`CRASHED`	*	$[0, \infty)$	r	$10 \cdot \kappa_{max}$	$10 \cdot \kappa_{max}$
`SAT, UNSAT`	$\kappa \leq \kappa_{max}$	$[0, 0.1]$	0.1	r	r
`SAT, UNSAT`	$\kappa \leq \kappa_{max}$	$[0.1, \kappa)$	r	r	r
`SAT, UNSAT`	$\kappa \leq \kappa_{max}$	$[\kappa, \kappa_{max})$	r	r	r
`SAT, UNSAT`	$\kappa \leq \kappa_{max}$	$[\kappa_{max}, \infty)$	r	$10 \cdot \kappa_{max}$	$10 \cdot \kappa_{max}$
`TIMEOUT`	$\kappa < \kappa max$	$[0, \infty)$	r	κ	κ
`TIMEOUT`	$\kappa = \kappa max$	$[0, \infty)$	r	$10 \cdot \kappa_{max}$	$10 \cdot \kappa_{max}$

A description of the locations is as follows:

Tuner Time The amount of time that will be subtracted from the remaining tuner time limit given in the scenario.

PAR10 Score The value that will be used for empirical comparisons between configurations.

Model The value that will be used to build the model.

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, 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 is 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's 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 7.3

6 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 **--rungroup** sub-directory.

6.1 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.*,**--log-level**,**--console-log-level**,**--log-all-call-strings**,**--log-all-process-output**), 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 **--seed** 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.

6.1.1 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 the message `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:

```
*****Runtime Statistics*****
Count: 15
Incumbent ID: 11 (0x1D8A2)
Number of Runs for Incumbent: 33
Number of Instances for Incumbent: 5
Number of Configurations Run: 30
Performance of the Incumbent: 0.04219047619047619
Configuration Time Budget used: 30.038589432000038 s (100%)
Configuration Time Budget remaining: -0.038589432000037505 s
Wall-clock Time Budget used: 36.816 s (0%)
Wall-clock Time Budget remaining: 2.147483610184E9 s
Algorithm Runs used: 104.0  (0%)
Algorithm Runs remaining: 9.223372036854776E18
Model/Iteration used: 14.0  (0%)
Model/Iteration remaining: 2.147483633E9
Configuration Space Searched 0.0 %
Sum of Target Algorithm Execution Times \
(treating minimum value as 0.1): 27.20000000000004 s
CPU time of Configurator: 2.838245764 s
User time of Configurator: 2.8384146279999998 s
Total Reported Algorithm Runtime: 21.05999999999998 s
Sum of Measured Wallclock Runtime: 30.20900000000001 s
Max Memory: 910.25 MB
Total Java Memory: 226.8125 MB
Free Java Memory: 199.03646850585938 MB
```

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 second ID will always associate with a unique first ID, but not conversely. The second ID roughly represents the specific configuration in memory [12].

`Performance of the Incumbent`

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

`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

[12] 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.

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)

6.2 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 **--seed** 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.

param-file If **--save-context** is enabled, a copy of the **paramfile** will be in the state folder

instances If **--save-context** is enabled, a copy of the **instance_file** will be in the state folder

instance-features If **--save-context** is enabled, and SMAC is running with features, then a copy of the **feature_file** will be in the state folder.

scenario If **--save-context** is enabled, and SMAC is using a scenario file, then a copy of the **--scenario-file** will be in the state folder.

6.3 Trajectory File

SMAC also outputs a trajectory file into identical files `traj-run-`N`.txt` [13] 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 **–rungroup**, and then the **–seed**.

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 **–run-obj** and **–overall-obj**
Wallclock Time σ	Time of entry with respect to wallclock time.
Incumbent ID	The ID of the incumbent as listed in the **param_strings** file 6.2.
acTime	CPU Time of SMAC
Remaining Columns	Give a name value mapping for the configuration value as given by the `Incumbent ID` column

6.4 Validation Output

When Validation is completed four files are outputted, (again N is the value of the **–seed** 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.

   `validationResults-run`N`.csv`
   CSV File containing the result of the validation. The columns are defined as follows:

Column Name	Description
Tuner Time	The tuner time when validation occurred
Emperical Performance	The incumbent's performance on the training set
Test Set Performance	The incumbent's performance on the test set
AC Overhead Time	Total CPU Time Used by the Automatic Configurator
Sample Call String	

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

7 Developer Reference

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

7.1 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:

- The bulk of the code necessary to run SMAC lives in four classes `AbstractAlgorithmFramework`, `SequentialModelBasedAlgorithmConfiguration`, `SMACBuilder` and finally, `SMACExecutor`.

7.2 Class Overview

The most important classes to SMAC are as follows:

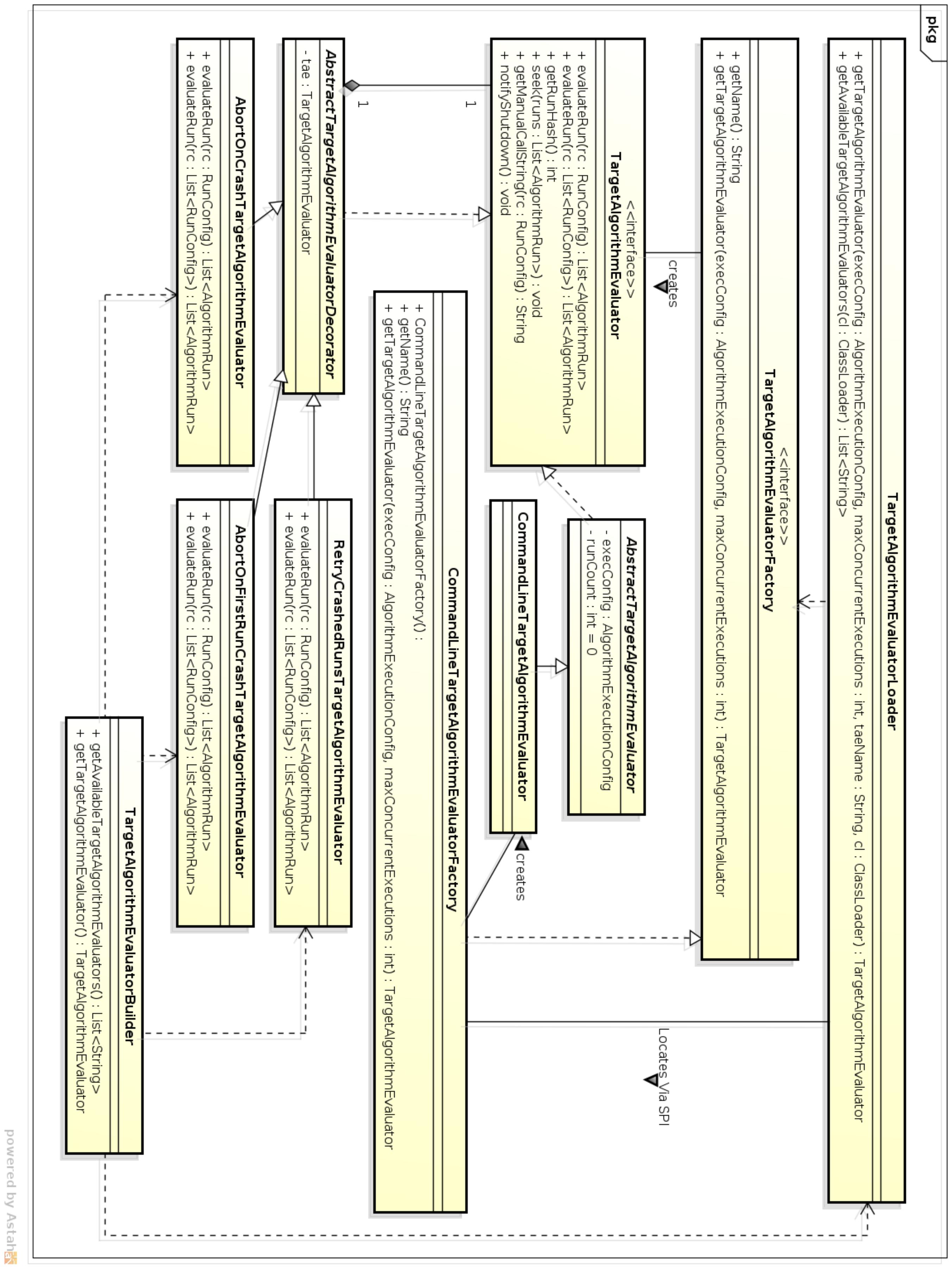
Automatic Configurator Library Classes	
Name	Description
`AbstractOptions`	Base class for creating new options for SMAC. While not important in and of itself, you will generally be implementing or modifying one of it's subtypes to implement options.
`AlgorithmRun`	Interface that represents the results 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 7.3 for more information.

SMAC Library Classes	
Name	Description
`AbstractAlgorithmFramework`	*Non-abstract* class that provides a default Automatic Configurator (ROAR)
`SequentialModelBasedAlgorithmConfiguration`	Class that subtypes `AbstractAlgorithmFramework` and implements the methods required for SMAC
`SMACExecutor`	Parses command line options and creates some of the objects SMAC needs to execute (SMAC entrypoint)
`SMACBuilder`	Takes the options parsed by SMACExecutor or some other utility, and builds everything necessary to create an instance of `AbstractAlgorithmFramework`. If you want to plug smac into your application, you generally want to mimic what SMACExecutor does to invoke SMACBuilder.
`Validator`	Performs Validation of selected configurations
`ValidatorExecutor`	Entry point to stand alone validation utility

7.3 Target Algorithm Evaluator

WARNING: The Target Algorithm Evaluator API has changed a bit since this section was written, you are encouraged to look in the code for an example of how it now works. Much of this is still relevant, and this section will be fixed for v2.08.00, but for v2.06.00 it will contain some inaccuracies.

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
<<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
creates
1
1
AbstractTargetAlgorithmEvaluator
- execConfig : AlgorithmExecutionConfig
- runCount : int = 0
CommandLineTargetAlgorithmEvaluator
creates
CommandLineTargetAlgorithmEvaluatorFactory
+ CommandLineTargetAlgorithmEvaluatorFactory() :
+ getName() : String
+ getTargetAlgorithmEvaluator(execConfig : AlgorithmExecutionConfig, maxConcurrentExecutions : int) : TargetAlgorithmEvaluator
Locates Via SPI
AbstractTargetAlgorithmEvaluatorDecorator
- tae : TargetAlgorithmEvaluator
AbortOnCrashTargetAlgorithmEvaluator
+ evaluateRun(rc : RunConfig) : List<AlgorithmRun>
+ evaluateRun(rc : List<RunConfig>) : List<AlgorithmRun>
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
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 7.4 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 7.4.

7.4 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 [14].

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:

[14]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 v2.06.00-development-583 (2e12acc92f4
[INFO ] Version of Random Forest Library is v1.05.01-development-95 (4a8077e95b21)
[INFO ] Version of SMAC is v2.06.00b-development-561 (abb03ff41e82)
```

This can make debugging and managing reproducibility much easier. Most projects include the first 12 characters of the git commit hash to make it easy to find that commit.

7.5 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.

8 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. Alexandre Fréchette and Chris Thornton for their constant feedback and patches to SMAC. We would also like to thank Marius Schneider for many valuable early bug reports and suggestions for improvements.

9 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.

10 Appendix

10.1 Return Codes

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

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

10.2 Version History of Java SMAC

Version 2.00 (Aug-2012) First Internal Release of Java SMAC (this is a port and extension of the original Matlab version).

Version 2.02 (Oct-2012) First Public Release of SMAC v2 and contained many fixes from the previous release.

Version 2.04 (Dec-2012) Second Release of Java SMAC including the following improvements:

1. Validation file output times consistent with Tuner Times
2. Some **INFO** log statements have been moved to **DEBUG** and some **DEBUG** to **TRACE**
3. Added support for verifying whether responses of SAT and UNSAT are consistent with Instance Specific Information see **--verifySAT** option for more information
4. Added support for the SMAC_MEMORY environment variable to control how much RAM (in MB) SMAC will use when executed via the supplied shell scripts.
5. Context is now added to the state folders to make it easier to debug issues later, to disable consult the **--saveContext** option.
6. Greatly improved memory usage in State Serialization code, and now we free the existing model prior to building a new one, so for some JVMs this may improve memory usage.

Version 2.04.01 (Feb-2013) Minor Bug Fix of Java SMAC

1. Added option to validate over training set instances
2. Can now use <DEFAULT> as a configuration to validate against
3. Fixed bug where **TIMEOUT** runs below our requested cutoff time are not counted properly when considering incumbent changes
4. Can now specify the initial incumbent with the **--initialIncumbent** option.
5. Wallclock time is now saved in the trajectory file instead of -1
6. FAQ Improvements
7. Git commit hash is now documented in Manual, FAQ, and Version strings
8. **(BETA)** Support for bash auto-completion of arguments for `smac` and `smac-validate`. You can load the file by running:

```
. ./util/bash_autocomplete.sh
```

Version 2.04.02 (Aug-2013) Minor Bug Fix of Java SMAC

1. Incumbent Performance now displayed when validation is turned off.
2. **--runtimeLimit** option is no longer just for show.

Version 2.06.00 (Aug-2013) Significant Feature Enhancements

1. New `algo-test` utility allows easy invocation of wrappers.
2. New `verify-scenario` utility preforms extra validation on scenario files.
3. Scenario now ends if the configuration space is exhausted
4. SMAC now lets you search only a subspace for good configurations
5. Validation output formats improved with headers
6. Option to always compare with the initial incumbent (to prevent an early poor choice from derailing the run) (See **--always-run-initial-config**)
7. SMAC reports an error if runs give different answers for **SAT** and **UNSAT** now
8. New **--restore-scenario** option to make restoring scenarios easier
9. New **--warmstart** option makes it possible to preload the model with additional SMAC runs.
10. Can now set seeds to different parts of SMAC using **-S**
11. Runtime Statistics and Termination Reasons now rewritten
12. New validation options **--validate-all**, **--validate-only-if-tunertime-reached** (See the validation options for all of them)
13. SMAC now checks limits *before* scheduling a run, rather than immediately after the run as in previous versions. (This means that if the last run went over, but changed the incumbent it will be logged.)
14. Instances can now be ordered deterministically (that is in the order they are declared in the instance file via **--determinstic-instance-ordering**.
15. Usage improved via new help levels which are displayed with **--help-level** and new usage screens.
16. Improvements to bash auto completion.
17. Target Algorithm Evaluators now take options.
18. Fixes for CPU Time calculation in SMAC.
19. Example scenarios cleaned up, new ones provided.
20. SMAC should be more forgiving with relative paths in a scenario file.
21. Default option files now supported (SMAC will read from `/.aclib/smac.opt`, `/.aclib/tae.opt` and `/.aclib/help.opt`. It will also read from defaults for plugins that are available.
22. Rungroup name is now configurable.
23. Logging of some objects is cleaned up.
24. Windows Startup scripts, and improved Unix start up scripts.
25. Fixed lock-up issue with wrappers launching unterminating subprocesses.
26. Fixed ConvergenceException error message.

27. Options now have a primary non-camel case format.
28. Manual now has a basic options section, before listing all the options.
29. Significant API changes to the Target Algorithm Evaluators so previous plugins will need to be refactored (and another change will come either in v2.06 or v2.08).
30. SMAC will now match capitalization of words in the Result String of wrappers.
31. New **--validation-seed** option should cause the validation at the end of SMAC to behave the same as the stand-alone utility.

Version 2.06.01 (Oct-2013) Minor Bug Fix Release of Java SMAC

1. Fixed a bug introduced in 2.06.00 that caused validation to be performed against the training instance distribution instead of the test instance distribution.
2. Default acquisition function for solution quality optimization is now Expected Improvement (instead of Exponential Expected Improvement).
3. Fixed exception if Scenario file doesn't have extension.
4. New option **--terminate-on-delete** will cause SMAC to abort the procedure before the next set of runs (as if it had hit it's CPU time limit) if the file specified is deleted.
5. New option **--kill-runs-on-file-delete** will cause SMAC to kill any runs in progress . This option should be used with care, as it may cause SMAC to select the wrong incumbent, and it should always be used with **--terminate-on-delete**.
6. New option **--save-runs-every-iteration** will cause SMAC to output the runs and results file necessary to restore state every run. This is useful if your cluster or environment is particularly unreliable. It should NOT be used when runtimes in the scenario can grow very small as the amount of time SMAC will spend writing to disk loosely [15] changes from $O(n)$ to $O(n^2)$, where n is the number of runs it performs.
7. If SMAC is shutting down for an unexpected reason (e.g. `OutOfMemoryError` ,or it received a `SIGTERM`), SMAC will now try its best to write a final batch of state data with the "SHUTDOWN" prefix.
   NOTE: This state may be corrupted for a variety of reasons, and even if it is written correctly you may not be able to restore it properly as the snap shot may be from the middle of an iteration.
8. Fixed typo in error message that mistakenly reported that instances where missing, when in fact it was the test instances that were missing.

10.3 Known Issues

1. The state merge utility may crash if merging runs that don't have a run for every instance.
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.

[15] Assuming the number of iterations scales linearly with the number of runs.

4. On older versions of Java ($<$1.6.0_23), SMAC may get an IOException with Out Of Memory when trying to execute the target algorithms
5. Auto detecting of iterations break if some runs and results files are missing, the work around is to remove some of the .obj files that don't have a runs and results file that is greater than or equal to them. (The quick object files internally store there iteration, and you'd have to check the logs to see what they correspond to).
6. ArrayOutOfBoundsException occurs if not all instances have features

10.4 Basic Options Reference

The following sections outline only the basic options

10.4.1 SMAC Options

General Options for Running SMAC

BASIC OPTIONS

--doubling-capping-challengers Number of challengers to use with the doubling capping mechanism

Default Value: 2

Domain: (0, 2147483647]

--doubling-capping-runs-per-challenger Number of runs each challenger will get with the doubling capping initilization strategy

Default Value: 2

Domain: (0, 2147483647]

--experiment-dir root directory for experiments Folder

Default Value: <current working directory>

--help show help

--help-level Show options at this level or lower

Default Value: BASIC

Domain: {BASIC, INTERMEDIATE, ADVANCED, DEVELOPER}

--num-run number of this run (also used as part of seed)

REQUIRED

Default Value: 0

Domain: [0, 2147483647]

--restore-scenario Restore the scenario & state in the state folder

Default Value: null

Domain: FILES

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

Default Value:

--validation perform validation when SMAC completes

Default Value: true

Domain: {true, false}

--validation-seed Seed to use for validating SMAC

Default Value: 0 which should cause it to run exactly the same as the stand-alone utility.

-v print version and exit

10.4.2 Scenario Options

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

Basic Options

--feature-file file that contains the all the instances features

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

REQUIRED

Default Value: null

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

Default Value: MEAN10

Domain: {MEAN, MEAN1000, MEAN10}

--output-dir Output Directory

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

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

Default Value: RUNTIME

Domain: {RUNTIME, QUALITY}

--scenario-file scenario file

Domain: FILES

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

Default Value: null

10.4.3 Scenario Configuration Limit Options

Options that control how long the scenario will run for

Basic Options

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

Default Value: 2147483647

Domain: [0, 2147483647]

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

Default Value: 2147483647

Domain: (0, 2147483647]

10.4.4 Algorithm Execution Options

Options related to invoking the target algorithm

BASIC OPTIONS

--algo-cutoff-time CPU time limit for an individual target algorithm run

REQUIRED

Default Value: 0.0

Domain: $(0, \infty)$

--algo-deterministic treat the target algorithm as deterministic

Default Value: false

Domain: $\{\mathsf{true}, \mathsf{false}\}$

--algo-exec command string to execute algorithm with

REQUIRED

Default Value: null

--algo-exec-dir working directory to execute algorithm in

REQUIRED

Default Value: null

--param-file File containing algorithm parameter space information in PCS format (see Algorithm Parameter File in the Manual). You can specify ”SINGLETON” to get a singleton configuration space or ”NULL” to get a null one.

Default Value: null

10.5 Complete Options Reference

10.5.1 SMAC Options

General Options for Running SMAC

BASIC OPTIONS

--doubling-capping-challengers Number of challengers to use with the doubling capping mechanism

Aliases: --doubling-capping-challengers
Default Value: 2
Domain: (0, 2147483647]

--doubling-capping-runs-per-challenger Number of runs each challenger will get with the doubling capping initilization strategy

Aliases: --doubling-capping-runs-per-challenger
Default Value: 2
Domain: (0, 2147483647]

--experiment-dir root directory for experiments Folder

Aliases: --experiment-dir, --experimentDir, -e
Default Value: <current working directory>

--help show help

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

--help-level Show options at this level or lower

Aliases: --help-level
Default Value: BASIC
Domain: {BASIC, INTERMEDIATE, ADVANCED, DEVELOPER}

--num-run number of this run (also used as part of seed)

REQUIRED
Aliases: --num-run, --numrun, --numRun, --seed
Default Value: 0
Domain: [0, 2147483647]

--restore-scenario Restore the scenario & state in the state folder

Aliases: --restore-scenario, --restoreScenario
Default Value: null
Domain: FILES

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

Aliases: --rungroup, --rungroup-name, --runGroupName
Default Value:

--validation perform validation when SMAC completes

Aliases: --validation, --doValidation

Default Value: true
Domain: {true, false}

--validation-seed Seed to use for validating SMAC
Aliases: --validation-seed
Default Value: 0 which should cause it to run exactly the same as the stand-alone utility.

-v print version and exit
Aliases: -v, --version

INTERMEDIATE OPTIONS

--adaptive-capping Use Adaptive Capping
Aliases: --adaptive-capping, --ac, --adaptiveCapping
Default Value: Defaults to true when –runObj is RUNTIME, false otherwise
Domain: {true, false}

--always-run-initial-config if true we will always run the default and switch back to it if it is better than the incumbent
Aliases: --always-run-initial-config, --alwaysRunInitialConfiguration
Default Value: false
Domain: {true, false}

--console-log-level default log level of console output (this cannot be more verbose than the logLevel)
Aliases: --console-log-level, --consoleLogLevel
Default Value: INFO
Domain: {TRACE, DEBUG, INFO, WARN, ERROR, OFF}

--deterministic-instance-ordering If true, instances will be selected from the instance list file in the specified order
Aliases: --deterministic-instance-ordering, --deterministicInstanceOrdering
Default Value: false
Domain: {true, false}

--exec-mode execution mode of the automatic configurator
Aliases: --exec-mode, --execution-mode, --executionMode
Default Value: SMAC
Domain: {SMAC, ROAR}

--initial-challenger-runs initial amount of runs to request when intensifying on a challenger
Aliases: --initial-challenger-runs, --initialN, --initialChallenge
Default Value: 1
Domain: (0, 2147483647]

--initial-incumbent Initial Incumbent to use for configuration (you can use RANDOM, or DEFAULT as a special string to get a RANDOM or the DEFAULT configuration as needed). Other configurations are specified as: -name 'value' -name 'value' ... For instance: --quick-sort 'on'
Aliases: --initial-incumbent, --initialIncumbent

Default Value: DEFAULT

--initial-incumbent-runs initial amount of runs to schedule against for the default configuration

Aliases: --initial-incumbent-runs, --initialIncumbentRuns, --defaultConfigRuns
Default Value: 1
Domain: (0, 2147483647]

--log-level messages will only be logged if they are of this severity or higher.

Aliases: --log-level, --logLevel
Default Value: DEBUG
Domain: {TRACE, DEBUG, INFO, WARN, ERROR, OFF}

--num-challengers number of challengers needed for local search

Aliases: --num-challengers, --numChallengers, --numberOfChallengers
Default Value: 10
Domain: (0, 2147483647]

--num-ei-random number of random configurations to evaluate during EI search

Aliases: --num-ei-random, --numEIRandomConfigs, --numberOfRandomConfigsInEI, --numRandomConfigsInEI, --numberOfEIRandomConfigs
Default Value: 10000
Domain: [0, 2147483647]

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

Aliases: --num-pca, --numPCA
Default Value: 7
Domain: (0, 2147483647]

--save-runs-every-iteration if true will save the runs and results file to disk every iteration. Useful if your runs are expensive and your cluster unreliable, not recommended if your runs are short as this may add an unacceptable amount of overhead

Aliases: --save-runs-every-iteration
Default Value: false
Domain: {true, false}

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

Aliases: --seed-offset, --seedOffset
Default Value: 0

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

Aliases: --show-hidden, --showHiddenParameters

--warmstart location of state to use for warm-starting

Aliases: --warmstart, --warmstart-from
Default Value: N/A (No state is being warmstarted)

Advanced Options

--ac-add-slack amount to increase computed adaptive capping value of challengers by (post scaling)

Aliases: --ac-add-slack, --capAddSlack
Default Value: 1.0
Domain: $(0, \infty)$

--ac-mult-slack amount to scale computed adaptive capping value of challengers by

Aliases: --ac-mult-slack, --capSlack
Default Value: 1.3
Domain: $(0, \infty)$

--acq-func acquisition function to use during local search

Aliases: --acq-func, --acquisition-function, --ei-func, --expected-improvement-function, --expectedImprovementFuncti
Default Value: null
Domain: {EXPONENTIAL, SIMPLE, LCB, EI}

--clean-old-state-on-success will clean up much of the useless state files if smac completes successfully

Aliases: --clean-old-state-on-success, --cleanOldStateOnSuccess
Default Value: true
Domain: {true, false}

--config-tracking Take measurements of configuration as it goes through it's lifecycle and write to file (in state folder)

Aliases: --config-tracking
Domain: {true, false}

--help-default-file file that contains default settings for SMAC

Aliases: --help-default-file, --helpDefaultsFile
Default Value: /.aclib/help.opt
Domain: FILES

--imputation-iterations amount of times to impute censored data when building model

Aliases: --imputation-iterations, --imputationIterations
Default Value: 2
Domain: [0, 2147483647]

--init-mode Initialization Mode

Aliases: --init-mode, --initialization-mode, --initMode, --initializationMode
Default Value: CLASSIC
Domain: {CLASSIC, ITERATIVE_CAPPING}

--intensification-percentage percent of time to spend intensifying versus model learning

Aliases: --intensification-percentage, --intensificationPercentage, --frac_rawruntime
Default Value: 0.5
Domain: (0, 1)

--iterativeCappingBreakOnFirstCompletion In Phase 2 of the initialization phase, we will abort the first time something completes and not look at anything else with the same kappa limits

Aliases: --iterativeCappingBreakOnFirstCompletion
Default Value: false
Domain: {true, false}

--iterativeCappingK Iterative Capping K

Aliases: --iterativeCappingK
Default Value: 1

--mask-censored-data-as-kappa-max Mask censored data as kappa Max

Aliases: --mask-censored-data-as-kappa-max, --maskCensoredDataAsKappaMax
Default Value: false
Domain: {true, false}

--mask-inactive-conditional-parameters-as-default-value build the model treating inactive conditional values as the default value

Aliases: --mask-inactive-conditional-parameters-as-default-value, --maskInactiveConditionalParametersAsDefaultValu
Default Value: true
Domain: {true, false}

--max-incumbent-runs maximum number of incumbent runs allowed

Aliases: --max-incumbent-runs, --maxIncumbentRuns, --maxRunsForIncumbent
Default Value: 2000
Domain: (0, 2147483647]

--option-file read options from file

Aliases: --option-file, --optionFile
Domain: FILES

--option-file2 read options from file

Aliases: --option-file2, --optionFile2, --secondaryOptionsFile
Domain: FILES

--print-rungroup-replacement-and-exit print all the possible replacements in the rungroup and then exit

Aliases: --print-rungroup-replacement-and-exit
Default Value: false
Domain: {true, false}

--restore-iteration iteration of the state to restore, use "AUTO" to automatically pick the last iteration

Aliases: --restore-iteration, --restoreStateIteration, --restoreIteration
Default Value: N/A (No state is being restored)

--restore-state-from location of state to restore

Aliases: --restore-state-from, --restoreStateFrom
Default Value: N/A (No state is being restored)

--save-context saves some context with the state folder so that the data is mostly self-describing (Scenario, Instance File, Feature File, Param File are saved)

Aliases: --save-context, --saveContext, --saveContextWithState

Default Value: true

Domain: {true, false}

--smac-default-file file that contains default settings for SMAC

Aliases: --smac-default-file, --smacDefaultsFile

Default Value: /.aclib/smac.opt

Domain: FILES

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

Aliases: --state-deserializer, --stateDeserializer

Default Value: LEGACY

Domain: {NULL, LEGACY}

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

Aliases: --state-serializer, --stateSerializer

Default Value: LEGACY

Domain: {NULL, LEGACY}

--treat-censored-data-as-uncensored builds the model as-if the response values observed for cap values, were the correct ones [NOT RECOMMENDED]

Aliases: --treat-censored-data-as-uncensored, --treatCensoredDataAsUncensored

Default Value: false

Domain: {true, false}

--warmstart-iteration iteration of the state to use for warm-starting, use ”AUTO” to automatically pick the last iteration

Aliases: --warmstart-iteration

Default Value: AUTO (if being restored)

-S Sets specific seeds (by name) in the random pool (e.g. -SCONFIG=2 -SINSTANCE=4). To determine the actual names that will be used you should run the program with debug logging enabled, it should be output at the end.

Aliases: -S

10.5.2 Random Forest Options

Options used when building the Random Forests

INTERMEDIATE OPTIONS

--rf-log-model store response values in log-normal form

Aliases: --rf-log-model, --log-model, --logModel

Default Value: true if optimizing runtime, false if optimizing quality

Domain: {true, false}

--rf-num-trees number of trees to create in random forest

  Aliases: --rf-num-trees, --num-trees, --numTrees, --nTrees, --numberOfTrees
  Default Value: 10
  Domain: (0, 2147483647]

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

  Aliases: --rf-ratio-features, --ratioFeatures
  Default Value: 0.8333333333333334
  Domain: (0, 1]

ADVANCED OPTIONS

--rf-full-tree-bootstrap bootstrap all data points into trees

  Aliases: --rf-full-tree-bootstrap, --fullTreeBootstrap
  Default Value: false
  Domain: {true, false}

--rf-ignore-conditionality ignore conditionality for building the model

  Aliases: --rf-ignore-conditionality, --ignoreConditionality
  Default Value: false
  Domain: {true, false}

--rf-impute-mean impute the mean value for the all censored data points

  Aliases: --rf-impute-mean, --imputeMean
  Default Value: false
  Domain: {true, false}

--rf-min-variance minimum allowed variance

  Aliases: --rf-min-variance, --minVariance
  Default Value: 1.0E-14
  Domain: $(0, \infty)$

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

  Aliases: --rf-penalize-imputed-values, --penalizeImputedValues
  Default Value: false
  Domain: {true, false}

--rf-shuffle-imputed-values shuffle imputed value predictions between trees

  Aliases: --rf-shuffle-imputed-values, --shuffleImputedValues
  Default Value: false
  Domain: {true, false}

--rf-split-min minimum number of elements needed to split a node

  Aliases: --rf-split-min, --split-min, --splitMin
  Default Value: 10
  Domain: [0, 2147483647]

DEVELOPER OPTIONS

--rf-preprocess-marginal build random forest with preprocessed marginal

Aliases: --rf-preprocess-marginal, preprocessMarginal
Default Value: true
Domain: {true, false}

--rf-store-data store full data in leaves of trees

Aliases: --rf-store-data, --rf-store-data-in-leaves, --storeDataInLeaves
Default Value: false
Domain: {true, false}

--rf-subsample-memory-percentage when free memory percentage drops below this percent we will apply the subsample percentage

Aliases: --rf-subsample-memory-percentage, --freeMemoryPecentageToSubsample
Default Value: 0.25
Domain: (0, 1]

--rf-subsample-percentage multiply the number of points used when building model by this value

Aliases: --rf-subsample-percentage, --subsamplePercentage
Default Value: 0.9
Domain: (0, 1]

--rf-subsample-values-when-low-on-memory subsample model input values when the amount of memory available drops below a certain threshold (see --subsampleValuesWhenLowMemory) (Not Tested)

Aliases: --rf-subsample-values-when-low-on-memory, --subsampleValuesWhenLowOnMemory, --subsampleValuesWhenLowMemory
Default Value: false
Domain: {true, false}

10.5.3 Scenario Options

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

BASIC OPTIONS

--feature-file file that contains the all the instances features

Aliases: --feature-file, --instanceFeatureFile, --feature_file

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

REQUIRED
Aliases: --instance-file, --instanceFile, -i, --instance_file, --instance_seed_file
Default Value: null

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

Aliases: --intra-obj, --intra-instance-obj, --overall-obj, --intraInstanceObj, --overallObj, --overall_obj, --intra_instance_obj

Default Value: MEAN10
Domain: {MEAN, MEAN1000, MEAN10}

--output-dir Output Directory
Aliases: --output-dir, --outputDirectory, --outdir
Default Value: <current working directory>/smac-output

--run-obj per target algorithm run objective type that we are optimizing for
Aliases: --run-obj, --runObj, --run_obj
Default Value: RUNTIME
Domain: {RUNTIME, QUALITY}

--scenario-file scenario file
Aliases: --scenario-file, --scenarioFile
Domain: FILES

--test-instance-file file containing a list of instances to use during the validation phase (see Instance File Format section of the manual)
Aliases: --test-instance-file, --testInstanceFile, --test_instance_file, --test_instance_seed_file
Default Value: null

ADVANCED OPTIONS

--check-instances-exist check if instances files exist on disk
Aliases: --check-instances-exist, --checkInstanceFilesExist
Default Value: false
Domain: {true, false}

--inter-obj objective function used to aggregate over multiple instances (that have already been aggregated under the Intra-Instance Objective)
Aliases: --inter-obj, --inter-instance-obj, --interInstanceObj, --inter_instance_obj
Default Value: MEAN
Domain: {MEAN, MEAN1000, MEAN10}

10.5.4 Scenario Configuration Limit Options

Options that control how long the scenario will run for

BASIC OPTIONS

--tunertime-limit limits the total cpu time allowed between SMAC and the target algorithm runs during the automatic configuration phase
Aliases: --tunertime-limit, --tuner-timeout, --tunerTimeout
Default Value: 2147483647
Domain: [0, 2147483647]

--wallclock-limit limits the total wall-clock time allowed during the automatic configuration phase
Aliases: --wallclock-limit, --runtime-limit, --runtimeLimit, --wallClockLimit

Default Value: 2147483647
Domain: (0, 2147483647]

INTERMEDIATE OPTIONS

--iteration-limit limits the number of iterations allowed during automatic configuration phase

Aliases: --iteration-limit, --numIterations, --numberOfIterations
Default Value: 2147483647
Domain: (0, 2147483647]

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

Aliases: --runcount-limit, --totalNumRunsLimit, --numRunsLimit, --numberOfRunsLimit
Default Value: 9223372036854775807
Domain: (0, 9223372036854775807]

ADVANCED OPTIONS

--max-norun-challenge-limit if the parameter space is too small we may get to a point where we can make no new runs, detecting this condition is prohibitively expensive, and this heuristic controls the number of times we need to try a challenger and get no new runs before we give up

Aliases: --max-norun-challenge-limit, --maxConsecutiveFailedChallengeIncumbent
Default Value: 1000

--terminate-on-delete Terminate the procedure if this file is deleted

Aliases: --terminate-on-delete
Default Value: null

--use-cpu-time-in-tunertime include the CPU Time of SMAC as part of the tunerTimeout

Aliases: --use-cpu-time-in-tunertime, --countSMACTimeAsTunerTime
Default Value: true
Domain: {true, false}

10.5.5 Algorithm Execution Options

Options related to invoking the target algorithm

BASIC OPTIONS

--algo-cutoff-time CPU time limit for an individual target algorithm run

REQUIRED
Aliases: --algo-cutoff-time, --cutoff-time, --cutoffTime, --cutoff_time
Default Value: 0.0
Domain: $(0, \infty)$

--algo-deterministic treat the target algorithm as deterministic

Aliases: --algo-deterministic, --deterministic
Default Value: false
Domain: {true, false}

--algo-exec command string to execute algorithm with

REQUIRED
Aliases: --algo-exec, --algoExec, --algo
Default Value: null

--algo-exec-dir working directory to execute algorithm in

REQUIRED
Aliases: --algo-exec-dir, --exec-dir, --execDir, --execdir
Default Value: null

--param-file File containing algorithm parameter space information in PCS format (see Algorithm Parameter File in the Manual). You can specify "SINGLETON" to get a singleton configuration space or "NULL" to get a null one.

Aliases: --param-file, --pcs-file, -p, --paramFile, --paramfile
Default Value: null

ADVANCED OPTIONS

--continous-neighbours Number of neighbours for continuous parameters

Aliases: --continous-neighbours, --continuous-neighbors, --continuousNeighbours
Default Value: 4

DEVELOPER OPTIONS

--search-subspace Only generate random and neighbouring configurations with these values. Specified in a "name=value,name=value,..." format (Overrides those set in file)

Aliases: --search-subspace, --searchSubspace
Default Value: null

--search-subspace-file Only generate random and neighbouring configurations with these values. Specified each parameter on each own line with individual value

Aliases: --search-subspace-file, --searchSubspaceFile
Default Value: null
Domain: FILES

10.5.6 Target Algorithm Evaluator Options

Options that describe and control the policy and mechanisms for algorithm execution

INTERMEDIATE OPTIONS

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

Aliases: --abort-on-crash, --abortOnCrash
Default Value: false
Domain: $\{\mathsf{true}, \mathsf{false}\}$

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

Aliases: --abort-on-first-run-crash, --abortOnFirstRunCrash
Default Value: true
Domain: {true, false}

--bound-runs [DEPRECATED] (Use the option on the TAE instead if available) if true, permit only --cores number of runs to be evaluated concurrently.

Aliases: --bound-runs, --boundRuns
Default Value: false
Domain: {true, false}

--check-sat-consistency Ensure that runs on the same problem instance always return the same SAT/UNSAT result

Aliases: --check-sat-consistency, --checkSATConsistency
Default Value: true
Domain: {true, false}

--check-sat-consistency-exception Throw an exception if runs on the same problem instance disagree with respect to SAT/UNSAT

Aliases: --check-sat-consistency-exception, --checkSATConsistencyException
Default Value: true
Domain: {true, false}

--cores [DEPRECATED] (Use the TAE option instead if available) maximum number of concurrent target algorithm executions

Aliases: --cores, --numConcurrentAlgoExecs, --maxConcurrentAlgoExecs, --numberOfConcurrentAlgoExecs
Default Value: 1

--kill-run-exceeding-captime Attempt to kill runs that exceed their captime by some amount

Aliases: --kill-run-exceeding-captime
Default Value: true
Domain: {true, false}

--kill-run-exceeding-captime-factor Attempt to kill the run that exceed their captime by this factor

Aliases: --kill-run-exceeding-captime-factor
Default Value: 10.0
Domain: $(1, \infty)$

--retry-crashed-count number of times to retry an algorithm run before reporting 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: --retry-crashed-count, --retryCrashedRunCount, --retryTargetAlgorithmRunCount
Default Value: 0
Domain: [0, 2147483647]

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

Aliases: --tae, --targetAlgorithmEvaluator
Default Value: CLI
Domain: $\{$ANALYTIC, BLACKHOLE, CLI, CONSTANT, PRELOADED, RANDOM$\}$

--track-scheduled-runs If true outputs a file in the output directory that outlines how many runs were being evaluated at any given time

Aliases: --track-scheduled-runs
Default Value: false
Domain: $\{$true, false$\}$

--verify-sat Check SAT/UNSAT/UNKNOWN responses against Instance specific information (if null then performs check if every instance has specific information in the following domain SAT, UNSAT, UNKNOWN, SATISFIABLE, UNSATISFIABLE

Aliases: --verify-sat, --verify-SAT, --verifySAT
Default Value: null
Domain: $\{$true, false$\}$

ADVANCED OPTIONS

--log-requests-responses If set to true all evaluation requests will be logged as they are submitted and completed

Aliases: --log-requests-responses
Default Value: false
Domain: $\{$true, false$\}$

--log-requests-responses-rc-only If set to true we will only log the run configuration when a run completes

Aliases: --log-requests-responses-rc-only, --log-requests-responses-rc
Default Value: false
Domain: $\{$true, false$\}$

--observer-walltime-delay How long to wait for an update with runtime information, before we use the walltime. With the 5 seconds and an scale of 0.95, it means we will see 0,0,0,0...,4.95...

Aliases: --observer-walltime-delay
Default Value: 5.0
Domain: $(0, \infty)$

--observer-walltime-if-no-runtime If true and the target algorithm doesn't update us with runtime information we report wallclock time

Aliases: --observer-walltime-if-no-runtime
Default Value: true
Domain: $\{$true, false$\}$

--observer-walltime-scale What factor of the walltime should we use as the runtime (generally recommended is the 0.95 times the number of cores)

Aliases: --observer-walltime-scale
Default Value: 0.95
Domain: $(0, \infty)$

--tae-default-file file that contains default settings for Target Algorithm Evaluators

Aliases: --tae-default-file
Default Value: /.aclib/tae.opt
Domain: FILES

DEVELOPER OPTIONS

--check-for-unclean-shutdown If true, we will try and detect an unclean shutdown of the Target Algorithm Evaluator

Aliases: --check-for-unclean-shutdown
Default Value: true
Domain: {true, false}

--check-for-unique-runconfigs Checks that all submitted Run Configs in a batch are unique

Aliases: --check-for-unique-runconfigs
Default Value: true
Domain: {true, false}

--check-for-unique-runconfigs-exception If true, we will throw an exception if duplicate run configurations are detected

Aliases: --check-for-unique-runconfigs-exception
Default Value: true
Domain: {true, false}

--check-result-order-consistent Check that the TAE is returning responses in the correct order

Aliases: --check-result-order-consistent, --checkResultOrderConsistent
Default Value: false
Domain: {true, false}

--exception-on-prepost-command Throw an abort

Aliases: --exception-on-prepost-command, --exceptionOnPrePostCommand
Domain: {true, false}

--kill-runs-on-file-delete All runs will be forcibly killed if the file is deleted. This option may cause the application to enter an infinite loop if the file is deleted, so care is needed. As a rule, you need to set this and some other option to point to the same file, if there is another option, then the application will probably shutdown nicely, if not, then it will probably infinite loop.

Aliases: --kill-runs-on-file-delete
Default Value: null

--post-scenario-command Command that will run on shutdown

Aliases: --post-scenario-command, --postScenarioCommand, --post_cmd

--pre-scenario-command Command that will run on startup

Aliases: --pre-scenario-command, --preScenarioCommand, --pre_cmd

--prepost-exec-dir Execution Directory for Pre/Post commands

Aliases: --prepost-exec-dir, --prePostExecDir

Default Value: Current Working Directory
Domain: {readabledirectories}

--prepost-log-output Log all the output from the pre and post commands
Aliases: --prepost-log-output, --logOutput
Domain: {true, false}

--run-hashcode-file 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: --run-hashcode-file, --runHashCodeFile
Domain: FILES

--skip-outstanding-eval-tae If set to true code, the TAE will not be wrapped by a decorator to support waiting for outstanding runs
Aliases: --skip-outstanding-eval-tae
Default Value: false
Domain: {true, false}

--track-scheduled-runs-resolution We will bucket changes into this size
Aliases: --track-scheduled-runs-resolution
Default Value: 10.0
Domain: $(0, \infty)$

10.5.7 Transform Target Algorithm Evaluator Decorator Options

This Target Algorithm Evaluator Decorator allows you to transform the response value of the wrapper according to some rules. Expressions that can be used by exp4j (http://www.objecthunter.net/exp4j/), can be specified and will cause the returned runs to be transformed accordingly. The variables in the expression can be S which will be -1 if the run was UNSAT, 1 if SAT, and 0 otherwise, R which is the original reported runtime, Q which is the original reported quality, and C which was the requested cutoff time. Care should be taken when transforming values to obey wrapper semantics. If you don’t know what you are doing, we recommend that SAT and UNSAT values should be kept in the range between 0 and cutoff, and the TIMEOUT value shouldn’t be transformed at all. A very special thanks to the original author Alexandre Fréchette.

ADVANCED OPTIONS

--tae-transform Set to true if you’d like to transform the result, if false the other transforms have no effect
Aliases: --tae-transform
Default Value: Identity transform.
Domain: {true, false}

--tae-transform-SAT-quality Function to apply to an algorithm run’s quality if result is SAT.
Aliases: --tae-transform-SAT-quality
Default Value: Identity transform.

Domain: Calculable string using a run's associated variables: S run result (SAT=1,UNSAT=-1,other=0), R runtime, Q quality, C cutoff.

--tae-transform-SAT-runtime Function to apply to an algorithm run's runtime if result is SAT.

Aliases: --tae-transform-SAT-runtime

Default Value: Identity transform.

Domain: Calculable string using a run's associated variables: S run result (SAT=1,UNSAT=-1,other=0), R runtime, Q quality, C cutoff.

--tae-transform-TIMEOUT-quality Function to apply to an algorithm run's quality if result is TIMEOUT.

Aliases: --tae-transform-TIMEOUT-quality

Default Value: Identity transform.

Domain: Calculable string using a run's associated variables: S run result (SAT=1,UNSAT=-1,other=0), R runtime, Q quality, C cutoff.

--tae-transform-TIMEOUT-runtime Function to apply to an algorithm run's runtime if result is TIMEOUT.

Aliases: --tae-transform-TIMEOUT-runtime

Default Value: Identity transform.

Domain: Calculable string using a run's associated variables: S run result (SAT=1,UNSAT=-1,other=0), R runtime, Q quality, C cutoff.

--tae-transform-UNSAT-quality Function to apply to an algorithm run's quality if result is UNSAT.

Aliases: --tae-transform-UNSAT-quality

Default Value: Identity transform.

Domain: Calculable string using a run's associated variables: S run result (SAT=1,UNSAT=-1,other=0), R runtime, Q quality, C cutoff.

--tae-transform-UNSAT-runtime Function to apply to an algorithm run's runtime if result is UNSAT.

Aliases: --tae-transform-UNSAT-runtime

Default Value: Identity transform.

Domain: Calculable string using a run's associated variables: S run result (SAT=1,UNSAT=-1,other=0), R runtime, Q quality, C cutoff.

--tae-transform-other-quality Function to apply to an algorithm run's quality if result is not SAT, UNSAT or TIMEOUT.

Aliases: --tae-transform-other-quality

Default Value: Identity transform.

Domain: Calculable string using a run's associated variables: S run result (SAT=1,UNSAT=-1,other=0), R runtime, Q quality, C cutoff.

--tae-transform-other-runtime Function to apply to an algorithm run's runtime if result is not SAT, UNSAT or TIMEOUT.

Aliases: --tae-transform-other-runtime

Default Value: Identity transform.

Domain: Calculable string using a run's associated variables: S run result (SAT=1,UNSAT=-1,other=0), R runtime, Q quality, C cutoff.

DEVELOPER OPTIONS

--tae-transform-valid-values-only If the transformation of runtime results in a value that is too large, the cutoff time will be returned, and the result changed to TIMEOUT. If the result is too small it will be set to 0

Aliases: --tae-transform-valid-values-only
Default Value: true
Domain: {true, false}

10.5.8 Validation Options

Options that control validation

INTERMEDIATE OPTIONS

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

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

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

Aliases: --min-timestamp, --minTimestamp
Default Value: 0.0
Domain: $[0, \infty)$

--num-validation-runs approximate number of validation runs to do

Aliases: --num-validation-runs, --numValidationRuns, --numberOfValidationRuns
Default Value: 1000
Domain: [0, 2147483647]

--save-state-file Save a state file consisting of all the runs we did

Aliases: --save-state-file, --saveStateFile
Default Value: false
Domain: {true, false}

--validate-by-wallclock-time Use wallclock times

Aliases: --validate-by-wallclock-time, --validateByWallClockTime
Default Value: false
Domain: {true, false}

--validate-only-if-tunertime-reached If the walltime in the trajectory file hasn't hit this entry we won't bother validating

Aliases: --validate-only-if-tunertime-reached, --validateOnlyIfTunerTimeReached
Default Value: 0.0

Domain: [0, ∞)

--validate-only-if-walltime-reached If the walltime in the trajectory file hasn't hit this entry we won't bother validating

Aliases: --validate-only-if-walltime-reached, --validateOnlyIfWallTimeReached

Default Value: 0.0

Domain: [0, ∞)

--validate-only-last-incumbent validate only the last incumbent found

Aliases: --validate-only-last-incumbent, --validateOnlyLastIncumbent

Default Value: true

Domain: $\{\mathsf{true}, \mathsf{false}\}$

ADVANCED OPTIONS

--mult-factor 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: --mult-factor, --multFactor

Default Value: 2.0

Domain: (0, ∞)

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

Aliases: --num-seeds-per-test-instance, --numSeedsPerTestInstance, --numberOfSeedsPerTestInstance

Default Value: 1000

Domain: (0, 2147483647]

--num-test-instances number of instances to test against (will execute min of this, and number of instances in test instance file). To disable validation in SMAC see the --doValidation option

Aliases: --num-test-instances, --numTestInstances, --numberOfTestInstances

Default Value: 2147483647

Domain: (0, 2147483647]

--output-file-suffix Suffix to add to validation run files (for grouping)

Aliases: --output-file-suffix, --outputFileSuffix

--validate-all Validate every entry in the trajectory file (overrides other validation options)

Aliases: --validate-all, --validateAll

Default Value: false

Domain: $\{\mathsf{true}, \mathsf{false}\}$

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

Aliases: --validation-rounding-mode, --validationRoundingMode

Default Value: UP

Domain: $\{\mathsf{UP}, \mathsf{NONE}\}$

--write-configuration-matrix Write the configuration matrix

Aliases: --write-configuration-matrix, --writeConfigurationMatrix, --writeThetaMatrix

Default Value: false
Domain: {true, false}

DEVELOPER OPTIONS

--validation-headers put headers on output CSV files for validation

Aliases: --validation-headers, --validationHeaders
Default Value: true
Domain: {true, false}

10.5.9 Analytic Target Algorithm Evaluator Options

This Target Algorithm Evaluator uses an analytic function to generate a runtime. Most of the function definitions come from Test functions for optimization needs, by Marcin Molga, Czesaw Smutnicki (http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf). NOTE: Some functions have been shifted vertically so that there response values are always positive.

INTERMEDIATE OPTIONS

--analytic-function Which analytic function to use

Aliases: --analytic-function
Default Value: CAMELBACK
Domain: {ZERO, ADD, CAMELBACK, BRANINS}

DEVELOPER OPTIONS

--analytic-observer-frequency How often to notify observer of updates (in milli-seconds)

Aliases: --analytic-observer-frequency
Default Value: 100
Domain: (0, 2147483647]

--analytic-scale-simulate-delay Divide the simulated delay by this value

Aliases: --analytic-scale-simulate-delay
Default Value: 1.0
Domain: $(0, \infty)$

--analytic-simulate-cores If set to greater than 0, the TAE will serialize requests so that no more than these number will execute concurrently.

Aliases: --analytic-simulate-cores
Default Value: 0
Domain: [0, 2147483647]

--analytic-simulate-delay If set to true the TAE will simulate the wallclock delay

Aliases: --analytic-simulate-delay
Default Value: false
Domain: {true, false}

10.5.10 Blackhole Target Algorithm Evaluator Options

This Target Algorithm Evaluator simply never returns any runs

DEVELOPER OPTIONS

--blackhole-warnings Suppress warning that is generated

Aliases: --blackhole-warnings
Default Value: true
Domain: {true, false}

10.5.11 Command Line Target Algorithm Evaluator Options

INTERMEDIATE OPTIONS

--cli-concurrent-execution Whether to allow concurrent execution

Aliases: --cli-concurrent-execution
Default Value: true
Domain: {true, false}

--cli-cores **Aliases:** --cli-cores
Default Value: 1
Domain: (0, 2147483647]

--cli-log-all-call-strings log every call string

Aliases: --cli-log-all-call-strings, --log-all-call-strings, --logAllCallStrings
Default Value: false
Domain: {true, false}

--cli-log-all-process-output log all process output

Aliases: --cli-log-all-process-output, --log-all-process-output, --logAllProcessOutput
Default Value: false
Domain: {true, false}

ADVANCED OPTIONS

--cli-default-file file that contains default settings for CLI Target Algorithm Evaluator (it is recommended that you use this file to set the kill commands)

Aliases: --cli-default-file
Default Value: /.aclib/cli-tae.opt
Domain: FILES

--cli-listen-for-updates If true will create a socket and set environment variables so that we can have updates of CPU time

Aliases: --cli-listen-for-updates
Default Value: true
Domain: {true, false}

--cli-pg-force-kill-cmd Command to execute to try and ask the process group to terminate nicely (generally a SIGKILL in Unix). Note

Aliases: --cli-pg-force-kill-cmd

Default Value: bash -c ”kill -s KILL -

--cli-pg-nice-kill-cmd Command to execute to try and ask the process group to terminate nicely (generally a SIGTERM in Unix). Note

Aliases: --cli-pg-nice-kill-cmd

Default Value: bash -c ”kill -s TERM -

--cli-proc-force-kill-cmd Command to execute to try and ask the process to terminate nicely (generally a SIGTERM in Unix). Note

Aliases: --cli-proc-force-kill-cmd

Default Value: kill -s KILL

--cli-proc-nice-kill-cmd Command to execute to try and ask the process to terminate nicely (generally a SIGTERM in Unix). Note

Aliases: --cli-proc-nice-kill-cmd

Default Value: kill -s TERM

DEVELOPER OPTIONS

--cli-observer-frequency How often to notify observer of updates (in milli-seconds)

Aliases: --cli-observer-frequency

Default Value: 750

Domain: (0, 2147483647]

10.5.12 Constant Target Algorithm Evaluator Options

Parameters for the Constant Target Algorithm Evaluator

DEVELOPER OPTIONS

--constant-additional-run-data Additional Run Data to return

Aliases: --constant-additional-run-data

--constant-run-length Runlength to return

Aliases: --constant-run-length

Default Value: 0.0

--constant-run-quality Quality to return

Aliases: --constant-run-quality

Default Value: 0.0

--constant-run-result Run Result To return

Aliases: --constant-run-result

Default Value: SAT

Domain: $\{\mathsf{TIMEOUT}, \mathsf{SAT}, \mathsf{UNSAT}, \mathsf{CRASHED}, \mathsf{ABORT}, \mathsf{RUNNING}, \mathsf{KILLED}\}$

--constant-runtime Runtime to return

Aliases: --constant-runtime

Default Value: 1.0

10.5.13 Preloaded Response Target Algorithm Evaluator

Target Algorithm Evaluator that provides preloaded responses

DEVELOPER OPTIONS

--preload-additional-run-data Additional Run Data to return

Aliases: --preload-additional-run-data

--preload-quality Quality to return on all values

Aliases: --preload-quality
Default Value: 0.0

--preload-response-data Preloaded Response Values in the format [SAT,UNSAT,...=x], where x is a runtime (e.g. [SAT=1],[UNSAT=1.1]...

Aliases: --preload-response-data, --preload-responseData

--preload-run-length Runlength to return on all values

Aliases: --preload-run-length, --preload-runLength
Default Value: -1.0

10.5.14 Random Target Algorithm Evaluator Options

This Target Algorithm Evaluator randomly generates responses from a uniform distribution

DEVELOPER OPTIONS

--random-additional-run-data Additional Run Data to return

Aliases: --random-additional-run-data

--random-max-response The maximum runtime we will generate

Aliases: --random-max-response
Default Value: 10.0
Domain: $[0, \infty)$

--random-min-response The minimum runtime we will generate (values less than 0.01 will be rounded up to 0.01)

Aliases: --random-min-response
Default Value: 0.0
Domain: $[0, \infty)$

--random-observer-frequency How often to notify observer of updates (in milli-seconds)

Aliases: --random-observer-frequency
Default Value: 100
Domain: (0, 2147483647]

--random-sample-seed Seed to use when generate random responses

Aliases: --random-sample-seed
Default Value: Current Time in Milliseconds

--random-scale-simulate-delay Divide the simulated delay by this value

Aliases: --random-scale-simulate-delay

Default Value: 1.0

Domain: $(0, \infty)$

--random-simulate-cores If set to greater than 0, the TAE will serialize requests so that no more than these number will execute concurrently.

Aliases: --random-simulate-cores

Default Value: 0

Domain: [0, 2147483647]

--random-simulate-delay If set to true the TAE will simulate the wallclock delay

Aliases: --random-simulate-delay

Default Value: false

Domain: $\{\mathsf{true}, \mathsf{false}\}$

--random-trend-coefficient The Nth sample will be drawn from Max(0,Uniform(min,max) + N$\times$(trend-coefficient)) distribution. This allows you to have the response values increase or decrease over time.

Aliases: --random-trend-coefficient

Default Value: 0.0