Advances in Meta-algorithmic Software Libraries for Distributed Automated Algorithm Configuration

by

Stephen Edward Andrew Ramage

B.Sc, The University of British Columbia, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

April 2015

© Stephen Edward Andrew Ramage, 2015
Abstract

A meta-algorithmic procedure is a computer procedure that operates upon another algorithm and its associated design space to produce another algorithm with desirable properties (e.g., faster runtime, better solution quality, ...; see e.g., Hoos [2008]). Many meta-algorithmic procedures have runtimes that are dominated by the runtime of the algorithm being operated on. This holds in particular for automatic algorithm configurators, such as ParamILS, SMAC, and GGA, which serve to optimize the design (expressed through user settable parameters) of an algorithm under certain use cases. Consequently, one can gain improved performance of the meta-algorithm if evaluations of the algorithm under study can be done in parallel. In this thesis, we explore a distributed version of the automatic configurator, SMAC, called pSMAC, and the library, AEATK, that it was built upon, which has proved general and versatile enough to support many other meta-algorithmic procedures.
Preface

I participated in the design of pSMAC along with my co-collaborators Frank Hutter, Kevin Leyton-Brown and Holger H. Hoos. For AEATK, I primarily designed and implemented it, but it’s evolution and development were driven by requirements by myself and other users in the lab, especially Alexandre Fréchette, and Frank Hutter.

Some other contributions of note:

1. Daniel Geschwender worked on implementing some features for the MySQL and HAL Target Algorithm Evaluators.

2. Alexandre Fréchette helped implement several features, and many bug fixes.

3. Christopher Thornton provided several bug fixes.

4. Frank Hutter actually provided the initial code that everything grew out of, and most of the abstractions within it are actually more concrete formalisms of his previous work on SMAC and ParamILS. Additionally some text in here has been lifted on documents were wrote together.

5. Christopher Fawcett provided some bug fixes, as well as provided some guidance based on his experience with HAL.
# Table of Contents

Abstract .......................................................... ii

Preface ............................................................... iii

Table of Contents ................................................... iv

List of Tables ......................................................... vii

List of Figures ......................................................... ix

Acknowledgements .................................................... xii

1 Introduction ......................................................... 1
   1.1 Contributions ....................................................... 1
   1.2 Thesis Organization .............................................. 2

2 Background ......................................................... 3
   2.1 Introduction ....................................................... 3
   2.2 Meta-Algorithmic Libraries ...................................... 4
   2.3 Algorithm Configuration ......................................... 5
       2.3.1 Existing Approaches ........................................ 7
       2.3.2 Sequential Model Based Optimization Approaches .......... 8

3 AEATK ............................................................... 12
   3.1 Introduction ...................................................... 12
       3.1.1 Motivation & Related Work ................................ 12
       3.1.2 Design Goals ................................................ 13
       3.1.3 Algorithm Execution Model ................................ 14
       3.1.4 Sample Use Cases for Target Algorithm Execution ....... 17
3.2 Domain Objects .............................................. 18
  3.2.1 Parameter Configuration Spaces ..................... 18
  3.2.2 ProblemInstance & ProblemInstanceSeedPair ........... 19
  3.2.3 AlgorithmExecutionConfiguration .................... 19
  3.2.4 AlgorithmRunConfiguration ......................... 20
  3.2.5 AlgorithmRunResult ................................. 21
  3.2.6 Differences Between the Object and Conceptual Models ... 21
3.3 Target Algorithm Evaluator API .............................. 22
  3.3.1 Execution Options ................................... 22
3.4 Local Execution ............................................. 25
  3.4.1 Target Algorithm Invocation ......................... 26
  3.4.2 Concurrent Execution ................................. 28
  3.4.3 Target Algorithm Observation & Termination ......... 28
3.5 Distributed Execution ....................................... 30
  3.5.1 Architecture ........................................ 30
  3.5.2 Distributed vs Local Execution Performance .......... 33
3.6 Modification & Customization ............................... 38
  3.6.1 New Implementations ................................. 38
  3.6.2 Existing Implementations ......................... 40
  3.6.3 Execution Customization ............................ 41
3.7 Easy Distribution ........................................... 43
  3.7.1 Command-line Interface .............................. 44
  3.7.2 Output ............................................. 45
3.8 Conclusions and Limitations ................................ 45

4 SMAC in AEATK .................................................. 48
  4.1 Introduction ............................................. 48
  4.2 Algorithm Overview ...................................... 48
  4.3 AEATK Implementation ................................... 49
  4.4 Differences Between MATLAB and Java Implementations ... 51
  4.5 Experimental Comparison ................................ 52
    4.5.1 Computational Environment ....................... 52
    4.5.2 Results ........................................... 53
    4.5.3 Conclusions & Future Work ....................... 53
5 Parallel Automatic Configuration

5.1 Introduction

5.1.1 Parallel Independent Runs

5.1.2 Parallel Automatic Configurator

5.1.3 Parallel Dependent Runs

5.2 pSMAC

5.3 Experiments

5.3.1 Experiment I - Performance Over Time

5.3.2 Results

5.4 Experiment II - CSSC Performance

5.4.1 Experimental Setup

5.4.2 Results

5.5 Future Extensions

Bibliography

A AEATK Decorators

A.1 Debug Decorators

A.2 Functionality Decorators

A.3 Helper Decorators

A.4 Resource Decorators

A.5 Safety Decorators
List of Tables

Table 3.1  System configuration for performance comparison between distributed and local execution. 34

Table 4.1  Configuration scenarios for MATLAB and Java SMAC Implementations. Additionally, all of the multi-instance scenarios used the continuous parameter configuration spaces. For single instance scenarios involving the SPEAR solver, the configuration space was discretized. 56

Table 4.2  System configuration for all configuration scenarios 56

Table 4.3  Median training performance of MATLAB and Java implementations of SMAC, along with the default performance for single-instance scenarios from Hutter et al. [2011b]. The number of independent runs for each is denoted in the # column. Some runs of MATLAB did not complete successfully. Bold-faced entries represent configurator performance that is significantly better under a Mann-Whitney U Test with significance level of 0.01 (a Bonferroni correction was applied so the individual p-values are less than $6.667 \times 10^{-4}$). 57

Table 4.4  Median performance of MATLAB and Java implementations of SMAC, along with the default performance for multi-instance scenarios from Hutter et al. [2011b]. The number of independent runs for each is denoted in the # column. Some runs of MATLAB did not complete successfully. Bold-faced entries represent configurator performance that is significantly better under a Mann-Whitney U Test with a significance level of 0.01 (a Bonferroni correction for 15 samples was applied so the individual p-values are less than $6.667 \times 10^{-4}$). 57
Table 5.1  Median training performance of SMAC and pSMAC implementations. For Best of SMACx4, we performed bootstrap samplings of 4 runs from the # column, computed the minimum, and then the medium of 1000 samples. For the pSMACx4 run, we took grouped runs into batches of 4 for sharing runs, and took # independent samples from this. Bold-faced entries represent configurator performance that is significantly better under a Mann-Whitney U Test with significance level of 0.01 (because of the repeated testing we applied a Bonferroni correction and adjusted the individual p-values used for significance of the test to less than $6.667 \times 10^{-4}$).

Table 5.2  Median training performance of SMAC and pSMAC implementations. For Best of SMACx8, we performed bootstrap sampling of 8 runs from the # column, computed the minimum, and then the medium of 1000 samples. For the pSMACx8 run, we took grouped runs into batches of 8 for sharing runs, and took # independent samples from this. Bold-faced entries represent configurator performance that is significantly better under a Mann-Whitney U Test with significance level of 0.01 (because of the repeated testing we applied a Bonferroni correction and adjusted the individual p-values used for significance of the test to less than $6.667 \times 10^{-4}$).
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Goals of Selected Meta-Algorithmic Software Libraries</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Classes related to the configuration space of a target algorithm</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Classes that represent Problem Instances</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Class that represents the physical information needed to execute an algorithm</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>Class that represents all information needed to execute a target algorithm</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Class that represents the result of an target algorithm execution</td>
<td>21</td>
</tr>
<tr>
<td>3.7</td>
<td>The TargetAlgorithmEvaluator interface</td>
<td>22</td>
</tr>
<tr>
<td>3.8</td>
<td>Standard Execution Sequence Diagram</td>
<td>23</td>
</tr>
<tr>
<td>3.9</td>
<td>UML Diagram for Observer class</td>
<td>24</td>
</tr>
<tr>
<td>3.10</td>
<td>Sequence diagram outlining how observation interacts between classes</td>
<td>24</td>
</tr>
<tr>
<td>3.11</td>
<td>UML Diagram describing Callback class</td>
<td>24</td>
</tr>
<tr>
<td>3.12</td>
<td>Sequence diagram outlining asynchronous callbacks</td>
<td>25</td>
</tr>
<tr>
<td>3.13</td>
<td>TAE Queue Facade UML</td>
<td>26</td>
</tr>
<tr>
<td>3.14</td>
<td>TAE Queue Facade Sequence Diagram</td>
<td>26</td>
</tr>
<tr>
<td>3.15</td>
<td>Components involved in executing a target algorithm</td>
<td>27</td>
</tr>
<tr>
<td>3.16</td>
<td>Target algorithm execution with runsolver</td>
<td>28</td>
</tr>
<tr>
<td>3.17</td>
<td>MySQL Target Algorithm Evaluator architecture</td>
<td>31</td>
</tr>
<tr>
<td>3.18</td>
<td>Sharing of runs between applications</td>
<td>31</td>
</tr>
<tr>
<td>3.19</td>
<td>Processing time versus cutoff time for the TargetAlgorithmEvaluator</td>
<td>34</td>
</tr>
<tr>
<td>3.20</td>
<td>Processing time versus cutoff time for the MySQLTargetAlgorithmEvaluator</td>
<td>35</td>
</tr>
<tr>
<td>3.21</td>
<td>Overhead for the TargetAlgorithmEvaluator</td>
<td>36</td>
</tr>
<tr>
<td>3.22</td>
<td>Overhead for the MySQLTargetAlgorithmEvaluator</td>
<td>37</td>
</tr>
</tbody>
</table>
List of Algorithms

4.1 ............................................................... 55
Acknowledgements

There are many people that helped me get to the end of this process and consequently that I am indebted to. In no particular order, I would like to thank my supervisors: Holger H. Hoos, Kevin Leyton-Brown and especially my unofficial third supervisor Frank Hutter. Secondly, my lab mates Chris Fawcett, Alexandre Fréchette, and Chris Thornton. Next I’d like to thank my instructors and the department as a whole for being supportive. Fourthly, my friends: Elisabeth, Karioka, Marcin, Marek, Pawel, Sarah, and Susan. Lastly, my parents and grandmother.
Chapter 1

Introduction

A meta-algorithmic procedure is an algorithm that uses other algorithms as input. Many meta-algorithmic procedures are focused on optimizing algorithms or a set of algorithms for a specific task. One such class of meta-algorithmic procedures are those that configure or tune an algorithm’s parameters to minimize some objective function across a set of inputs or instances (e.g., finding the algorithm that can solve a set of problems with minimized runtime). Produces for solving this algorithm configuration problem are called automatic configurators like ParamILS, GGA, irance, and SMAC.

Automatic configurators have already made significant strides, in some cases improving algorithm perform times by several orders of magnitude (see e.g., Hutter et al. [2010a]).

This work extends previous work by describing an advancement in automatic configuration in parallel environments, that is, an automatic configurator that works in a distributed and parallel setting, pSMAC.

One model for meta-algorithmic procedures is that of an algorithm that emits a series of requests for runs of an algorithm, perhaps varying some parts or some input, and then operates on the result of the algorithm run, to make more decisions. This model indeed captures many meta-algorithmic procedures, not simply automatic configuration. Another contribution of this thesis is a new toolkit, called AEATK, which is described separately.

1.1 Contributions

The contribution of this thesis is twofold: A toolkit for abstracting algorithm executions, the second is a new automatic configurator that supports distributed configuration.
1.2 Thesis Organization

This thesis is organized as follows, Chapter 2 provides a survey of existing work on meta-algorithmic procedures in general, and development of (distributed) automatic configurators in particular. Chapter 3 provides a detailed description of the Algorithm Execution and Abstraction Toolkit (AEATK) including its design goals, the abstractions it provides, and the various mechanisms of execution available. Chapter 4 provides an empirical validation of a configurator implemented in AEATK. Finally, chapter 5 provides an overview of the design of pSMAC.
Chapter 2

Background

2.1 Introduction

Meta-algorithmic procedures (see Nell et al. [2011]) are algorithms that operate on or use other algorithms as part of their procedure. A simple example of a meta-algorithm would be given an instance of the Boolean satisfiability (SAT) problem (see, e.g., Gomes et al. [2008]), and a series of heuristic solvers each of which may solve it within a given time budget. Provided that there is some variation between which instances can be solved by each heuristic solver, and that none of them simply dominates the remainder, one can construct a simple more powerful meta-algorithmic solver that simply runs all of these algorithms in parallel, and terminates once any of them successfully solves it, returning the model from the successful solver. The above example is an example of a parallel algorithm portfolio. Other types of meta-algorithms may speculatively choose a particular solver which it feels has the best chance of solving an instance, this type of algorithm is called a algorithm selector, another type of meta-algorithm automatically “tunes” a heuristic algorithm with the objective of having it perform better (this is an instance of the algorithm configuration problem). In both cases, these meta-algorithms need to interact with the target algorithms, and Section 2.2 outlines existing software libraries that provide this functionality. In Section 2.3, we define the algorithm configuration problem more formally before surveying the literature on this topic.
2.2 Meta-Algorithmic Libraries

Most meta-algorithmic procedures, a procedure that operates on algorithms, perform runs on the algorithm under study. A standard and portable approach is to execute the algorithm via the shell or \texttt{system()} system call. This allows the meta-algorithmic procedure to treat the algorithm that is being used as a black box. The simplest of meta-algorithmic procedures use the result of the algorithm’s computation directly (e.g., whether a particular instance of the Boolean satisfiability problem is satisfiable or unsatisfiable), while other meta-algorithmic procedures take measurements of the algorithm performance, such as its runtime.

On the surface, executing target algorithm runs may seem trivial. As a result, designers of meta-algorithms tend to simply develop their own custom solution when they need to interact with target algorithms. However, there are in fact a number of issues related to algorithm execution for other users or during large scale experiments, such as:

- Measuring the runtime of an algorithm reliably.
- Terminating an algorithm run reliably.
- Policing algorithms that exceed resource limits (e.g., runtime and/or memory limits).
- Caching and reusing a previous algorithm execution (when possible) which allows for improved utilization of scarce resources. This can be extremely helpful in debugging, by minimizing the time needed to reproduce an error state in a meta-algorithm.
- Dealing with crashed runs and/or erroneous results.
- Executing algorithms in parallel.
- Distributing the execution of algorithms on a cluster.
- Managing the number of concurrent algorithm executions that happen at any given time.

In settings where a number of meta-algorithmic procedures have a similar purpose, the fact that they each execute the algorithm differently can be an annoying liability. For instance, GGA (Ansotegui et al. [2009]), ParamILS (Hutter et al. [2007b]), and
SMAC (Hutter et al. [2011b]) all solve the algorithm configuration problem, but each has their own method of interacting with the given target algorithm. In some cases, different executions can even make the results incomparable; for example, GGA executes and measures an algorithm runtime directly and is unable to determine if the algorithm terminated early due to an error. For these reasons, there is a need for a well-developed application programming interface (API) for executing target algorithm runs.

There are currently two projects that implement such an API as part of a more ambitious project. The first API is the Experiment Design and Administration for Computer Clusters (EDACC) from Balint et al. [2010] & Balint et al. [2011]. Originally conceived as a tool for managing algorithms in the context of SAT solver competitions, it evolved into a general experimental platform. It follows a master/slave architecture, in which the master defines the experiment and the runs needed, and then the slaves execute the target algorithm executions that are requested. To police algorithm resource usage, the runsolver tool is used (Roussel [2011]). EDACC also has support for running solver competitions, up to and including generating various plots regarding the experiments. The standard user interface is very limited in terms of possible experiments, but the API is rich and some example implementations based on SMAC are available publicly. Unfortunately, this aspect of EDACC seems largely undocumented, and development appears to have stalled.

Nell et al. [2011] developed another meta-algorithmic framework, the High-performance Algorithm Laboratory (HAL) which has similar goals as EDACC. It differs, however, in that it provides a rich structure and set of abstractions, in an attempt at being able to describe all meta-algorithmic procedures. Once described, meta-algorithmic procedures for certain problems, such as the algorithm configuration problem can have many implementations, such as ParamILS or GGA. The web-based interface seems to provide much more flexibility and control over experiments than EDACC’s user interface does. Both EDACC and HAL have the ability to directly interact with a computer cluster management system such as Torque or Sun Grid Engine, to schedule algorithm executions.

2.3 Algorithm Configuration

One of the most widely studied and useful meta-algorithmic design procedures are automated algorithm configurators, which are able to significantly improve algorithm performance, in some cases by several orders of magnitude (see, e.g., Hutter et al.
It is widely believed that a large class of algorithmic problems (e.g., verifying the 
existence of a solution to Boolean formula, or determining the shortest path that 
visits all point on a graph) features worst case runtimes that at least with current 
computer architectures make them theoretically intractable (Garey and Johnson 
[1990]). Empirically, however, acceptable performance is often attainable on instances 
of practical relevance (see, e.g., Leyton-Brown et al. [2014]). For example, while 
satisfiability problems generated randomly with certain parameters may be very 
difficult (Johnson [2002]), instances that occur in practice are often computationally 
feasible, such as those from hardware or software verification (see, e.g., Prasad et al. 
[2005]).

For NP-hard problems especially, better than brute-force performance on specific 
types of instances can be attained through the use of carefully designed heuristic 
methods. The performance of these heuristics may rely on implicit assumptions about 
design choices, such as parameter values that may either be incorrectly set, or may 
depend on the actual problem instance, or distribution of problem instances being 
examined. Additionally, as more of these heuristics are added, these may interact in 
unexpected and complex ways.

Algorithm configuration techniques (see, e.g., Hoos [2012b]) are able to explore the 
design space of an algorithm and optimize some objective such as algorithm runtime 
or solution quality.

Definition 1 The algorithm configuration problem can be defined as a 4-tuple 
$\langle A, \Theta, \Pi, m \rangle^1$, where:

- $A$ is a parameterized algorithm;
- $\Theta = \{\theta_1, \theta_2, \ldots\}$ is the possibly infinite set of allowed configurations of $A$ (i.e., a 
complete assignments of values to all parameters of $A$), each $\theta_i = \langle \theta_1, \theta_2, \ldots, \theta_k \rangle$, 
is an ordered $k$-tuple where each element $\theta_i$ has its own domain coming from 
either a finite set, or a non-empty interval over $\mathbb{R}$ or $\mathbb{N}$;
- $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ is a set of problem instances that $A$ will be run on;
- $m$ is a objective function that measures the performance of each $\theta$ across the 
instance set $\Pi$ for algorithm $A$. For instance the objective function might be 
the mean solution quality.

\footnote{This definition is slightly modified from Hoos [2012b].}
2.3.1 Existing Approaches

There are a number of existing approaches to solving the algorithm configuration problem. Hutter et al. [2007b] developed ParamILS, an automatic configurator that uses a local search optimization strategy. The local-search-based approach involves selecting configurations within the one-exchange neighbourhood (under which two configurations are neighbours if they are identical except for one parameter). The configuration space that ParamILS searches also supports the concept of conditional parameters, that is parameters which are only active or meaningful if another parameter is set to a certain value. ParamILS differs from many other approaches in that it is able to optimize multi-instance scenarios (i.e., many approaches can only optimize an algorithms performance for a particular instance, not over a set of instances). BasicILS, the simplest conceptual variant of ParamILS, evaluates each configuration on the entire instance distribution. FocusedILS, the variant that typically exhibits much better performance, initially starts with a small number of runs for a challenger (i.e., a new configuration which will be inspected to see if it’s better than the best configuration found so far), and then increases it so long as the empirical estimate of the objective function suggests it is better than the current incumbent (i.e., the best configuration found so far). FocusedILS also occasionally gives the current incumbent more runs, and supports random restarts in the configuration space. Finally, ParamILS introduced the adaptive capping mechanism in runtime optimization, where an algorithm run of a challenger is given a cutoff time which is set adaptively to the time it would take for the challenger to be demonstratively worse than the incumbent (e.g., if $t_{\text{MAX}}$ is 5 seconds, but the incumbent has runtime performance of 3 seconds on an instance, then the challenger need only be run for $3 + \epsilon$ seconds, where $\epsilon > 0$) (Hutter et al. [2009b]).

Another approach for solving the algorithm configuration problem is realised in the irace package by López-Ibáñez et al. [2011]. Their racing approach traces its origins to earlier method called F-Race (Birattari et al. [2002]). In F-Race, a set of candidate configurations are repeatedly run against each other until a statistical test reveals which configuration is statistically significantly better than the others. This method can only select a configuration from a pre-existing list of configurations. Balaprakash et al. [2007] modified F-Race, and created I/F-Race which runs multiple rounds of F-Race, and between each rounds samples new configurations with parameter settings biased towards the values of the promising configurations in the last round. The reference implementation for the I/F-Race algorithm is called irace (López-Ibáñez [2011]).
et al. [2011]), and includes I/F-Race as well as supporting other variants (such as changing the statistical tests used).

An entirely different approach is the Gender-based Genetic Algorithm (GGA) automatic configurator (Ansotegui et al. [2009]). GGA uses techniques from evolutionary computation to find the most promising configuration. GGA maintains two sets (genders) of configurations; the first is a set of promising configurations, the second is a set of other configurations of unknown performance. Each round, some fixed fraction of the promising configurations is selected to mate with one of the others (i.e., a new parameter setting is created by taking parts of each parent). GGA allows the designer of the algorithm to specify an And/Or tree encoding of the configuration space, which allows different parameters to be grouped together as a unit (genes) so that when mating, these groups of parameters can be selected together. The authors compare GGA against a non-gender based genetic algorithm, and find that GGA outperformed it. They note that the non-promising configurations allow for a wider variety of genes to be maintained beyond the number that could be actually evaluated. Additionally, it allows for good genes to stay around, even if they only occurred in configurations along with some other bad genes. They also evaluated the benefit provided by the grouping of parameters into genes and demonstrated significant improvements. Finally, they compared GGA against ParamILS on three scenarios (Hutter et al. [2007b]) and found that it generally tends to perform better, however, a later, thorough study by Hutter et al. [2011b] found that the ParamILS with adaptive capping performed better than GGA on these and other scenarios.

2.3.2 Sequential Model Based Optimization Approaches

Sequential Model-based Optimization (SMBO) techniques, which have a grounding in the statistics literature, are also suitable for solving the algorithm configuration problem.

The prominent EGO (Efficient Global Optimization) algorithm (Jones et al. [1998] & Jones [2001]) lays the ground work for model-based approaches to automatic algorithm configuration. In EGO, an arbitrary black box function is optimized by repeatedly fitting a model to the observed data, using that model to select a new input to evaluate, and then sampling the black box function at that point. The first model is built from data sampled in the same way as by Sacks et al. [1989]. Instead of optimizing the mean predicted value, EGO will optimize a more complex acquisition function, that takes both the mean and the uncertainty about the prediction into account. This prevents the procedure from selecting points too close to the best point.
found so far, but also still searches in regions in which we expect to find better inputs with high probability.

One limitation of EGO is that the black box function being optimized is presumed to be noiseless. Huang et al. [2006] extend EGO to create Sequential Kriging Optimization (SKO). In SKO, response values are treated as samples from a posterior distribution of the true value. To account for this, the acquisition function from Jones [2001] is augmented to support uncertainty and previously sampled points and to allow SKO to potentially resample points to obtain a better estimate as needed.

Bartz-Beielstein et al. [2005] adapted EGO to the optimization of algorithms, with their Sequential Parameter Optimization (SPO) method. SPO uses the same acquisition functions as EGO, but a slightly different model, which includes a second order polynomial fit as well as the standard Gaussian process model. Unlike EGO, their approach is able to deal with random response values through a continual resampling of the best observed points using a doubling strategy, which allows the estimate to converge to the true value over time. Finally, as opposed to fitting the model with each sample point individually, as done by SKO, SPO merges the samples for each point into a better estimate of the objective at that point, and then fits the model on these merged estimates.

Hutter et al. [2009a] directly compared SPO and SKO and their suitability for algorithm configuration. They found that SPO in general outperformed SKO on the algorithms they studied. They also introduced SPO+, which introduced some modifications to the original algorithm. First, they used log-transformed response values, which significantly improved the model’s predictive performance. To account for this transformation, they introduced a new acquisition function, log expected improvement. Finally, they noticed that SPO could be misled by a small number of runs on a challenger, and so introduced an invariant from their previous work (Hutter et al. [2009b]), which requires that for a challenger to be an incumbent, it must have at least as many samples as the incumbent.

This work was followed up by Hutter et al. [2010b], who improved upon SPO+ to form TB-SPO. TB-SPO differs from SPO+ in a few key ways. The first is that TB-SPO allows the procedure to meaningfully execute within a specific CPU time budget, as opposed to a fixed number of algorithm evaluations. The authors noted that runtime optimization of algorithms is conceptually different from ordinary expensive black-box function optimization, in that the cost of each measurement can vary significantly, and becomes cheaper as the optimization procedure makes progress. Therefore, instead of limiting the number of function evaluations that a procedure may take, it makes more
sense to limit the amount of time it takes. This is further true, because often building the model can take a disproportionate amount of time. The authors report that in some cases, previous experiments showed that the actual execution of the target algorithm only accounted for approximately 3% of the actual CPU time used, while the rest dealt with the overhead of building the models. The first change TB-SPO makes is that it drops the expensive initial design; instead, it interleaves random configurations with configurations selected by the model as the configuration procedure executes. The second is to alter the intensification strategy to be sensitive to the overhead of building the model: in TB-SPO many promising configurations are retrieved from the model (and mixed with randomly generated configurations), and these are executed in sequence until they use the same amount of CPU time as the last model build, thus ensuring that the algorithm gets to be executed at least 50% of the time. Finally, the authors replace the noise-free Gaussian process models with computationally cheaper approximations called projected process models.

Further advances to TB-SPO came with Hutter et al. [2011b], which also borrowed several ideas from ParamILS and defined the new automatic configurator SMAC. Unlike TB-SPO, SMAC was the first SMBO technique that was directly able to reason about configuration across multiple instances, in a manner similar to ParamILS. SMAC also replaced the previous projected process models with Random Forests (Breiman [2001]). Random Forests allow SMAC to optimize categorical parameters like ParamILS, but also continuous parameters like TB-SPO easily. Instead of treating each run of the target algorithm on a given instance as the same design point, the model is given both the used configurations and, if available, a vector of features describing each used instance features. To predict the performance of a configuration, the instance features are marginalized out of the model. As in previous SMBO approaches, promising configurations are selected from the model via random sampling, but in addition, SMAC also uses a local search procedure to optimize the acquisition function, afterwards merging the selected configurations and selecting them in order of the most promising acquisition function value. Hutter et al. [2011a] extended SMAC to add support for the adaptive capping mechanism from ParamILS, which required changes to the model building procedure (to account for the fact that response values now in some cases represented a lower bound on the performance) as well as to the intensification mechanism.

Finally Hutter et al. [2012] extended SMAC to work in a distributed setting, and also showed that SMAC can yield state of the art performance for solution quality optimization. D-SMAC modifies the general SMBO framework used in SMAC such
that the model building step and the evaluations happen in parallel. In D-SMAC, many configurations are chosen to be challenged against the current incumbent in parallel. Each configuration undergoes intensification to generate a set of runs if appropriate, or to terminate the challenge. Runs generated this way are placed in a queue; periodically, runs from this queue are distributed to workers synchronously, and the D-SMAC algorithm waits for all to be completed. If the size of the queue drops below a certain threshold, a new model is built and new configurations are generated. The authors demonstrated that for solution quality optimization, D-SMAC generates the best configurations and in general, as a function of wall-clock time, dominated sequential approaches, or approaches based on taking the best from a set of parallel independent runs of sequential approaches. D-SMAC, however, is largely limited to solution quality optimization, since it dispatches many runs to workers synchronously and then must wait for the final run to finish. A large variation of runtime would lead it to poorly utilizing the available resources, and it was thus not evaluated for runtime optimization.
Chapter 3

AEATK

3.1 Introduction

In this chapter, we outline the Algorithm Execution & Abstraction Toolkit (AEATK). After providing the motivation, an overview of the design goals, and some example applications, we provide an overview of the AEATK API, with an eye towards how these design goals are implemented.

3.1.1 Motivation & Related Work

In Section 2.2, we described two existing meta-algorithm libraries, and it is not clear why a third is needed. The *Gang of Four* (Gamma et al. [1994]) design patterns book makes a distinction between three types of software: applications, toolkits, and frameworks. A *toolkit* is a set of reusable components that provide useful features and functions. They do not “impose a design on the application”, and in general cannot make assumptions about how their functionality will be used. Alternatively a *framework* essentially provides a developer with an almost completely ready-made application and requires them to fill in the missing bits. Conceivably this makes it very quick to develop new applications, but requires the framework designer to “gamble” that their chosen architecture will work for all for all targeted applications. HAL and EDACC most closely fall into framework category and our experience using the former suggests it is very much an all-or-nothing activity. Additionally, our anecdotal experience developing meta-algorithmic procedures suggested that the architecture introduced by HAL suffers from several key limitations:

\footnote{An examination of the EDACC documentation suggests these would also be true of EDACC.}
Table 3.1: Goals of Selected Meta-Algorithmic Software Libraries

<table>
<thead>
<tr>
<th>Feature</th>
<th>HAL</th>
<th>EDACC</th>
<th>AEATK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Algorithm Execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distributed Target Algorithm Execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Experiment Specification &amp; Execution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Experiment Analysis</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Management of Solver Competitions</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Rich Representation of Meta-algorithmic Concepts</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Cluster Management</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Easily Modified Execution Architecture</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Easily Distributed Meta-algorithmic Procedures</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 3.1: Goals of Selected Meta-Algorithmic Software Libraries

1. All meta-algorithmic procedures require the availability of a properly configured MySQL server.

2. Users of a meta-algorithmic procedure would primarily interact with it via HAL’s web interface.

3. Meta-algorithmic procedures are tightly coupled to HAL and its API.

Historically AEATK “fell out” of the process of porting SMAC from MATLAB to Java. There was a strong desire to ensure that SMAC would be available for anyone who wished to configure an algorithm. While a direct implementation of HAL would allow users to easily manage experiments, and facilitate comparisons with comparable meta-algorithmic procedures, this was not viewed as necessary for the simplest of users, and most likely would be overwhelming and discourage uptake. Thus a lightweight abstraction layer was created to allow SMAC to be operated independently and directly via the command line, and it subsequently proved flexible enough to be adapted for other uses.

### 3.1.2 Design Goals

AEATK primarily evolved organically in response to my own needs, as well as the needs of other users. Nevertheless, there have been several unifying design goals that governed its evolution. If a new feature aligned with these goals, it would be added to AEATK, otherwise it would have to be implemented elsewhere. The central design goals of AEATK are the following:
1. A simple API should exist that allows developers to invoke a target algorithm and obtain a response.

2. It should be trivial to have target algorithm execution take place in a parallel and distributed manner. In particular, users should be shielded as much as possible from having to deal with the standard issues related to concurrent programming in Java, such as thread synchronization, data-races, or correct adherence to the Java memory model.

3. Target algorithm execution should be extensible, and policies (e.g., how target algorithm evaluations should be cached, or how errors should be handled) should be customizable, both for meta-algorithmic designers (e.g., the developer of a new automatic configurator), and for individual users of the meta-algorithm (e.g., users that would apply the new automatic configurator to a new target algorithm).

4. The meta-algorithmic procedures themselves should be easy to redistribute. In other words, they should be portable, and lend themselves to easy execution.

As AEATK is a toolkit and not a framework, and thus using it is not an all-or-nothing proposition. Indeed, while the central work in AEATK has been focused on target algorithm execution, at least one project (Zilla) which has no need for algorithm execution uses it for the last reason alone. Figure 3.1 outlines AEATK’s goals and contrasts them with both HAL and EDACC. One final note is that AEATK is not exclusive to the other approaches, in fact it can be complementary. The abstractions in AEATK are flexible and cohesive enough to lend themselves to working with both HAL and EDACC. Indeed, a number of meta-algorithmic procedures, including SMAC, can be used in both, provided that the meta-algorithmic procedure is properly encapsulated in Java (i.e., is not just implemented in the main() method) and that a TargetAlgorithmEvaluator (discussed in Section 3.3) is implemented in the other meta-algorithmic library. At the time of this writing, an implementation exists for HAL, and SMAC is available as a plugin in HAL.

3.1.3 Algorithm Execution Model

At a conceptual level, the model of an algorithm execution in AEATK is that of a function invocation: \( f : P \rightarrow R \). In this model \( f \) is the target algorithm
under study, $P$ is an ordered tuple $< \pi, \kappa, s, \theta >$, and $R$ is the ordered tuple $< \text{status}, \text{satisfiability}, \text{runtime}, \text{quality}, \text{data} >$.

The elements of $P$ are:

$\pi$ - The problem instance to be invoked.

$\kappa$ - The time (must be greater than or equal to zero) that the algorithm is permitted to execute for.

$s$ - A seed for the pseudo-random number generator used by the algorithm.

$\theta$ - The configuration of the algorithm.

The elements of $R$ are:

$\text{status}$ - One of \{SUCCESS, TIMEOUT, CRASHED, ABORT, KILLED\}. ‘SUCCESS’ indicates that the algorithm completed successfully. ‘TIMEOUT’ indicates that the algorithm did not complete within the time budget $\kappa$. ‘CRASHED’ indicates that the algorithm exhibited some unexpected failure. ‘ABORT’ also signifies some failure, but in this case the error is expected to be permanent\(^2\). ‘KILLED’ indicates that the target algorithm did not complete successfully, because the meta-algorithm asked for the run to be cancelled prematurely, later in Section 3.1.4 we will see examples where this is useful.

$\text{satisfiability}$ - One of \{SAT, UNSAT\}. This is useful when the target algorithm is a decision problem (for instance graph colouring, or boolean satisfiability), in which case ‘SAT’ indicates that the instance specified by $\pi$ in $P$ has a solution and ‘UNSAT’ indicates that it does not (e.g., this could indicate that a graph may be three-colourable or not three-colourable respectively). Other problems such as optimization problems typically have no use for this parameter.

$\text{runtime}$ - How long the took algorithm took to execute in seconds.

$\text{quality}$ - A problem specific measure of how good the run was. For instance if the target algorithm were a heuristic algorithm for the travelling salesman problem, this field might report the distance it found on a graph.

\(^2\)For example, imagine if the target algorithm is accessed by directly executing it, and the executable is not found. Users doing a single run of a meta-algorithmic procedure, or others doing large scale experiments are more likely to notice premature termination of the procedure than to examine the output of an on-going procedure. Additionally these types of errors typically invalidate the experiment anyway, so no benefit is derived by fixing this while the procedure is running.
**data** - Additional details on the run; for example, it might report the certificate to an NP-hard decision problem that answered ‘SAT’ for satisfiability above, other use cases have been debug information about the target algorithm execution. Typically general meta-algorithmic procedures do not rely on this field, and only specialized uses cases exist for this.

Additionally, we make the following assumptions:

1. As $f$ is viewed as a function, repeated invocations of $f(P)$ conceptually should result in identical output. The seed parameter is meant to control the randomization inherent in the algorithm itself. One limitation of the current model is that it is not possible to measure the distribution of runtime due to say cache effects or the operating system. It would be fairly easy to add another parameter to the tuple $P$, a sample index, that would allow multiple invocations of a specific point in the function easily.

2. If status is ‘TIMEOUT’ for some $\kappa$ and ‘SUCCESS’ for a $\kappa + \epsilon$ where ($\epsilon > 0$), then it should be the case that status is ‘TIMEOUT’ for all $y' < \kappa$, and ‘SUCCESS’ for all $\hat{y} > \kappa + \epsilon$. Additionally, if $\kappa$ is 0, then status will equal ‘TIMEOUT’.

3. The satisfiability parameter is only meaningful when status is ‘SUCCESS’, and it should be a function of the instance, $\pi$. In other words it should not be the case that for a particular $\pi$ this parameter is sometimes ‘SAT’, and other times ‘UNSAT’. In the example of graph colouring, a particular graph is either three-colourable or it is not. It does not depend on which graph colouring algorithm is used, or the configuration that algorithm.

AEATK expects the algorithm to satisfy these assumptions, which can facilitate improvements in the efficiency of handling algorithm execution through the use of caching and/or simplified logic. For instance, the first property allows for runs to be cached easily (i.e., if we previously have evaluated a particular $P$ we do not need to re-evaluate it, and can use the previous result), and the second allows us to improve on caching or skip execution entirely (if $\kappa = 0$). The third property is used for error detection in the target algorithm as occasionally different configurations can trigger bugs that cause the algorithm to produce incorrect results. Later, we will examine how this conceptual model is implemented and represented, and in some cases, the representation will deviate slightly from this model.
3.1.4 Sample Use Cases for Target Algorithm Execution

For the purposes of this discussion it will be helpful to have some example use cases in mind, consequently I will mention several use cases here:

**Online Single Execution** In this use case a meta-algorithm makes a decision about which target algorithm executions it would like to do, gets the result of the execution, and repeats. SMAC is a simple example which follows this work-flow. SMAC repeatedly executes the target algorithm comparing a new configuration’s (challenger) estimated performance versus that of the best found configuration (incumbent) thus far. After doing a target algorithm evaluation, it must decide whether to continue evaluating the challenger to get a better estimate of the performance, try a different challenger instead, or take another sample of the incumbent to improve its empirical estimate of the incumbents performance.

**Offline Batch Execution** In this use case, a meta-algorithm knows a priori exactly which invocations of target algorithms it would like to do and will do some computation when the results are completed. An example of this is the validation utility included in SMAC, which measures the performance of a configuration on a set of test instances. Another example is determining the runtime virtual best solver (VBS) for a set of instances and a set of solvers. This requires evaluating every solver on every instance, to determine which solver solved each instance in the least amount of time.

**Online Batch Execution** In this use case, many invocations of a target algorithm are needed before the meta-algorithm can continue, and (potentially) parallel resources are available to execute them. For example, imagine we have 10 cores and an automatic configurator knows that the best configuration (incumbent) found so far has an average time of 5 seconds over a set of 10 instances; to evaluate this incumbent in the conceptual model of Section 3.1.3 would require invoking the algorithm 10 times, each with a $\kappa$ (cutoff time) of 50 seconds, since any one of the runs could be the longest and they will be executed in parallel. Instead AEATK supports looking at the runs as it executes them, and capping them dynamically, thus only determining a lower bound on execution time. Another use case of this would be again determining the runtime VBS, where once the first solver solves an instance, the remaining solver executions can be terminated once they are known to take longer.

**Online Concurrent Execution** This use case is almost identical to the previous
one, except that the meta-algorithm is multi-threaded and these threads do not coordinate amongst themselves individual target algorithm execution. An example use case for this would be in parallelizing FocusedILS, by evaluating neighbours in parallel by a set of worker threads.

### 3.2 Domain Objects

In this section we outline the domain objects used in the AEATK API. These are objects which will correspond to the conceptual model outlined in Section 3.1.3 (e.g., The $P$ and $R$ tuple, configurations, instances, etc...). Most of the objects are very simple POJO objects with a minimal of logic, and in most cases immutable.

#### 3.2.1 Parameter Configuration Spaces

The `ParameterConfigurationSpace` and `ParameterConfiguration` classes (shown in Figure 3.2) allows representation of the design space or configuration space of a target algorithm. This space is exactly the space as introduced in SMAC (see Hutter et al. [2011b]). Notably, a parameter space of an algorithm can consist of two kinds of parameters. Parameters can be categorical or numeric. Additionally, certain
parameters can be active or inactive, depending on the value of other parameters. Finally combinations of parameters can be forbidden.

ParameterConfigurationSpace objects are constructed by supplying a string either from file or in memory containing a description of the space in PCS format (Hutter and Ramage [2013]). ParameterConfigurationSpace objects are immutable (i.e., stateless).

ParameterConfiguration objects can only be obtained from the space, via getRandomConfiguration() or getDefaultConfiguration(), or by supplying a string encoded version of a configuration to the getConfigurationFromString() method. ParameterConfiguration objects are mutable, but typically are modified only once in their life cycle.

3.2.2 ProblemInstance & ProblemInstanceSeedPair

A ProblemInstance class (shown in figure 3.3) represent the abstract problem instance that an algorithm will be run on. A ProblemInstanceSeedPair object is a container for a ProblemInstance, along with the seed the target evaluation should use, if it is a randomized algorithm. A ProblemInstance object can also contain an associated set of features. Both of these objects are immutable, are instantiable directly, and are simple objects with no complicated logic. In typical use cases, the instance will represent a file on disk, but nothing in AEATK requires this to be so.

3.2.3 AlgorithmExecutionConfiguration

AlgorithmExecutionConfiguration (Figure 3.4) are objects which represent the physical information about executing an algorithm, namely where the algorithm is, what its configuration space is, etc. It is an immutable, and again a simple object that can be constructed directly. The semantics of the individual fields vary depending on exactly how the algorithm will be executed (see Section 3.3). In the simplest use
Figure 3.4: Class that represents the physical information needed to execute an algorithm

```java
class AlgorithmExecutionConfiguration {
    + getAlgorithmExecutable() : String
    + getAlgorithmExecutionDirectory() : String
    + getParamFile() : ParameterConfigurationSpace
    + isDeterministicAlgorithm() : boolean
    + getAlgorithmCutoffTime() : double
    + getTargetAlgorithmExecutionContext() : Map<String, String>
}
```

Figure 3.5: Class that represents all information needed to execute a target algorithm case, the field names correspond to what one would expect. Additionally a `Map` object is included that may contain custom key/value pairs that can be used to customize algorithm execution.

### 3.2.4 AlgorithmRunConfiguration

An `AlgorithmRunConfiguration` (Figure 3.5) object includes everything necessary to execute a target algorithm through a command line, and is a merely a container for
the previously defined objects. Its constructor is directly accessible. This corresponds to the ordered tuple $P$ in our conceptual model from Section 3.1.3.

### 3.2.5 AlgorithmRunResult

An AlgorithmRunResult object (Figure 3.6) represents the result of executing a target algorithm with the given AlgorithmRunConfiguration. The details of how this is done will be discussed in Section 3.3. This object corresponds to the ordered tuple $R$ in the conceptual model from Section 3.1.3.

### 3.2.6 Differences Between the Object and Conceptual Models

There are two differences between the conceptual model and the object model. The first is that the function $f$ is encoded with the parameters $P$. The second is that in the AlgorithmRunResult the RunResult field combines the status and satisfiability fields in $R$, this is for historical reasons. Finally, the RunStatus can also take the value of RUNNING, which is used only in certain cases, when the meta-algorithm is exposed to the lower bound of runtime performance as discussed in the conceptual model.
3.3 Target Algorithm Evaluator API

Most of the value provided by AEATK lies within its ability to execute target algorithms, and specifically the `TargetAlgorithmEvaluator` interface shown in Figure 3.7. This interface provides a clear separation between the meta-algorithm and the target algorithm and allows meta-algorithm designers the ability to design their algorithms ignoring most of the details of actual execution. `TargetAlgorithmEvaluator` instances are thread-safe, and so client programs can schedule runs from multiple threads.

3.3.1 Execution Options

Depending on the meta-algorithmic procedure, different execution modes may be required, for example, fast concurrent performance can require the use of callback and asynchronous methods, but at the same time we do not want to require users to deal with more complexity than they need.

*Standard Execution* By default the simplest execution option is via the `evaluateRun()` method, which allows the meta-algorithm designer to supply a list of `AlgorithmRunConfiguration` objects to be evaluated (the second argument is optional, and its semantics will be covered in the *Observation* section). The method will return when all the runs are evaluated. This execution method corresponds to the Online Single Execution Case in Section 3.1.4. For completeness, a sequence diagram has been included to show this in Figure 3.8.

*Observation* During algorithm execution, additional information may become available that would cause one to change the parameters of that execution. For instance,
in the Online Batch Execution in Section 3.1.4, the determination of a VBS may decide, once an instance is solved by one target algorithm, to terminate other target algorithms runs that are longer. As a further example, consider a situation where the time budget on an experiment might run out, and it is desirable to end the procedure as quickly as possible. TargetAlgorithmEvaluator objects implement an observer (Gamma et al. [1994]) pattern that allows a given set of runs to be monitored. Depending on the TargetAlgorithmEvaluator, the currentStatus() method of the user-supplied observer may be invoked, and the implementation may request that the run be terminated by invoking the kill() method. Importantly, the termination mechanism is strictly advisory\(^3\), and the TargetAlgorithmEvaluator may or may not honour the request. This provides several benefits, including allowing the TargetAlgorithmEvaluator to ignore this request when inconvenient and discourages users from writing code that is prone to data races (e.g., between an algorithm being completed and requesting that it is terminated). The class structure for this is outlined in Figure 3.9, and a sequence diagram outlining the interaction is shown in Figure 3.10. The instances of the AlgorithmRunResult objects that the observer sees are snapshots and do not provide live updates.

**Asynchronous Execution** High-performing meta-algorithmic procedures may require the ability to execute runs asynchronously, and the evaluateRunsAsync() method can be used to achieve this purpose. As well as supplying the run, one supplies a callback object (typically an anonymous implementation of the

---

\(^3\)Other examples of advisory mechanisms in Java are invoking the method `System.gc()` for garbage collection, or `Thread.yield()` to suspend a thread. The Java virtual machine is free to ignore these requests, as these methods have no testable semantics (Bloch [2008]).
TargetAlgorithmEvaluatorCallback interface shown in Figure 3.11 that will be notified when the runs have been completed, as in Figure 3.12. Asynchronous execution is not actually guaranteed, and this method may block until resources become available within the TargetAlgorithmEvaluator to execute the request. For instance, the CommandLineTargetAlgorithmEvaluator will block after all CPU cores have been used. This provides a way for meta-algorithms to avoid generating new runs, when they cannot be run instantly. This mechanism provides a natural method of
controlling the rate at which runs are requested, as the blocking will act like a back pressure (see Welsh et al. [2001]) regulating the meta-algorithm.

**Single-Threaded Asynchronous Execution**  Conceptually, this mechanism provides for a high-performance version of the Online Concurrent Execution example in Section 3.1.4. One limitation, however, is that the callbacks are handled in a different thread than the caller, which can introduce issues of data races and memory visibility issues to the meta-algorithm designer. To minimize this, AEATK also includes a queue facade that allows for executions to be done asynchronously and concurrently, but which places the results in a queue (along with perhaps user supplied context), and retrieved by the submitting thread at some later point in time. This allows the meta-algorithm designer to gain most of the benefits of concurrent execution, while avoiding the drawbacks. Figure 3.13 shows the interface, and a sequence diagram illustrates how the client code would interact with it in Figure 3.14.

### 3.4 Local Execution

The preceding section outlined the TargetAlgorithmEvaluator interface, but did not provide any details regarding the implementations. AEATK includes a number of implementations of this interface, including an implementation that defers execution to HAL. In this section we outline how execution is done locally and discuss several challenges and how they were overcome. The implementation of the TargetAlgorithmEvaluator interface in AEATK that schedules runs locally is the CommandLineTargetAlgorithmEvaluator.
3.4.1 Target Algorithm Invocation

Typically, most executions are done by executing the algorithm through a standard command line interface, which traces its origins to ParamILS (the format is best documented in Hutter and Ramage [2013]). In short, the algorithm is invoked by executing the following in a shell:

```
<algo> <instance_name> <instance_specific_info> <cutoff_time> <cutoff_length> <seed> --<paramname>='<paramvalue>', --<paramname>='paramvalue' ...
```

Eventually, at some point the algorithm being executed must output the following:

Result of algorithm run: <status>, <runtime>, <runlength>, <quality>,...
When this is output, the various parameters are parsed, and a corresponding `AlgorithmRunResult` object is returned to the user. It should be clear how the above parameters and arguments map to the conceptual model in Section 3.1.3, and to the objects outlined in Section 3.2.

In practice, it is rare for the target algorithm to directly implement the interface needed by AEATK. Unlike HAL, which allows the user to instruct it exactly how to interact with the target algorithm, AEATK mandates this calling structure. To allow arbitrary algorithms to be executed, typically a wrapper is employed, which takes the call generated by AEATK and then adapts the call above to the calling structure of the algorithm. Figure 3.15 shows the typical components involved between the meta-algorithm or application and the target algorithm. Due to the difficulty associated with correctly measuring CPU time (see e.g., Chapter 9 of Bryant and O’Hallaron [2002]), typically these wrappers delegate this task to another special-purpose utility, called `runsolver` Roussel [2011], which then invokes the algorithm directly, as is shown in Figure 3.16.

The fact that the `CommandLineTargetAlgorithmEvaluator` delegates this entirely to the wrapper is a design choice that serves several purposes:

1. It ensures that runtime measurements are done consistently across meta-algorithm implementations, not just those using AEATK.

2. Java itself has no portable way of measuring runtimes, although libraries do exist, such as SIGAR.

3. It allows for more control over what exactly is being measured by the algorithm, for example, one use case we have experienced is that prior to the target algorithm being run, the instance is unzipped from a file and placed in a temporary directory. This can be specifically excluded from the processing time.
3.4.2 Concurrent Execution

The CommandLineTargetAlgorithmEvaluator may be configured to enable multiple simultaneous executions of a target algorithm. The management of physical CPU resources is left entirely to the discretion of the target algorithm; however, each concurrent run will be assigned an ID that will be exposed to it via the environment variable: AEATK_CONCURRENT_TASK_ID. The target algorithm may then choose to bind to only a specific subset of available cores (e.g., on a 16-core machine, if the environment variable was set to 4, and the target algorithm is dual-core, then it might limit the CPU affinity to cores 7 & 8). As mentioned in the previous section, the runsolver utility is often used to measure and enforce CPU time limits. AEATK includes a customized build of runsolver that will automatically set the affinity of the executing process when the environment variable is detected.

3.4.3 Target Algorithm Observation & Termination

One aspect of the conceptual model from Section 3.1.3 is the notion that function evaluations take time, and it can be useful to obtain a lower bound on runtime performance, as mentioned the examples from Section 3.1.4. The CommandLineTargetAlgorithmEvaluator maintains a thread pool and occasionally notifies any supplied observer of runs. To get an update of the current runtime, the CommandLineTargetAlgorithmEvaluator listens for a UDP message on a specific port passed as an environment variable (AEATK_PORT) to the target algorithm when invoked. Again, because runsolver is commonly used to measure and enforce runtime limits, AEATK’s custom version of it automatically does the notification upon detecting the environment variable. Figure 3.16 outlines the typical relation between the various components involved in physically executing a target algorithm, and the UDP message.
from runsolver to the CommandLineTargetAlgorithmEvaluator.

Termination of a target algorithm had historically been deferred to the wrapper, which typically defers to runsolver to ensure that all processes are terminated. Experimentally, we have found that this mechanism is insufficient. On shared multi-user clusters, where memory usage was strictly enforced externally, we noticed that occasionally processes would leak over time, eventually this would culminate in the job being terminated by the cluster for exceeding its resource limits. This occurred due to several reasons:

1. In several cases, runsolver was susceptible to segfaults, and when this occurred, the wrapper would complete, but the process would continue executing.

2. runsolver manages which processes to terminate by keeping track of a tree of processes and their children. With unfortunate timing and the parent process terminating, it is possible that runsolver may not detect a child process being spawned, if the parent process terminates too quickly.

3. The Java call to terminate a subprocess is the Process.destroy() method, which terminates the direct parent, not the process tree (in Figure 3.16, only the Wrapper process would be terminated, runsolver and the actual Target Algorithm would continue executing. This means that a kill() method which relied on this mechanism would inherently leak processes overtime in most cases.

After trying several mechanisms (including process groups), our design of AEATK eventually settled on the following mechanism, which was inspired by the mechanism that our local cluster used to track processes. When AEATK executes, it sets an environment variable AEATK_EXECUTION_UUID to a randomly generated Universally Unique Identifier (UUID). When the kill() method is invoked or the parent process completes execution, AEATK will find any process with the environment variable and value set appropriately and terminate that process. In almost all cases, the environment variables are inherited by subprocesses unmolested. This method has anecdotally proven to be very reliable in terms of preventing process leakage. The environment variable name may change, if the CommandLineTargetAlgorithmEvaluator detects that this name is already set in the currently running process, as this implies that we are already running as a child to some other TargetAlgorithmEvaluator.

Since Java does not provide an API that allows the inspection of other processes environment variables, to implement this functionality, the CommandLineTargetAlgorithmEvaluator delegates to a script, which takes the name
of the environment variable and the value as arguments and is responsible for ter-
minating all matching processes. In Linux this can be done by traversing the `/proc`
filesystem, and theoretically, the same could be done in Java; however benchmarking
suggested that it was on the order of 10 to 100 times slower to traverse the file system
in Java than it was to simply call a shell script which performed the same task via
`fgrep`. A portable way to do this on POSIX systems (including BSD and MacOS X)
seems to be through the output of `ps eww`, and a preliminary investigation suggested
that there exists a way to inspect the environment data of other processes via the
Win32 API. Conceivably, AEATK could automatically delegate to the appropriate
method, but in practice so far, no one has needed anything but Linux support for this
mechanism.

3.5 Distributed Execution

One goal outlined in Section 3.1.2 is support for leveraging distributed re-
sources in a straightforward manner. To meet this goal, AEATK provides
another implementation of the `TargetAlgorithmEvaluator` interface called the
`MySQLTargetAlgorithmEvaluator`. This section outlines the details of the imple-
mentation.

3.5.1 Architecture

The `MySQLTargetAlgorithmEvaluator` uses a master-slave architecture, in which
`AlgorithmRunConfiguration` objects are written to the database by the master, and
then processed on a corresponding slave/worker on another machine typically, but not
necessarily, by its `CommandLineTargetAlgorithmEvaluator`, as in Figure 3.17.

The `MySQLTargetAlgorithmEvaluator` is designed to be a lightweight and efficient
mechanism for performing large amounts of target algorithm evaluations. It is also
designed to be conceptually simple to manage, with most user intervention taking
place directly in the table structure. In contrast with HAL, there is no grouping of runs
by experiments. Conceptually, the table structure consists of a table that stores dis-
tinct `AlgorithmExecutionConfiguration` and `ParameterConfigurationSpace` ob-
jects, another table that stores the rest of the `AlgorithmRunResult` object as well as

---

Technically, this implementation is in a separate plugin as outlined in Section 3.6.1, but
this is largely to avoid requiring the extra space for database support in all programs.
At the code level, the only difference between the `CommandLineTargetAlgorithmEvaluator` and
`MySQLTargetAlgorithmEvaluator` is that the latter exists in a separate project; both are used and
interacted with the same way.
Figure 3.17: MySQL Target Algorithm Evaluator architecture

Figure 3.18: Sharing of runs between applications
indicating the status of whether the run has been completed by a worker, and a third
table that provides information about the workers and allow changing of some worker
parameters (e.g., how long to stay alive for, how often to poll the database, how many
runs to retrieve from the database at a single time, etc...).

In many use cases, applications will share a pool of workers, which allows for
improved efficiency if desired, see Figure 3.18. The database structure very much
follows the conceptual model outlined in Section 3.1.3; in particular, the table or
pool serves simply as a cache of runs, that are resolved by workers over time. In
fact, the resolution of jobs by workers is entirely anonymous to the clients of the
MySQLTargetAlgorithmEvaluator, and there is no connection between the application
and the worker executing it. The application merely polls the database periodically
to check for run status, and the workers poll the database to check for new runs to
execute. Additionally, the workers can automatically adapt the number of runs they
poll from the database in a single trip, if they detect that the runs are very short, and
\textit{push back} unstarted runs to the database if other workers are idle.

It should be noted that to distribute runs requires no changes to the application
code, and in some circumstances (discussed more in Section 3.7.1), the only changes
needed for users to turn a locally executing meta-algorithm to a distributed one is
to change a single argument on the command line and to schedule workers on their
cluster. To run a worker, the user merely executes the \texttt{mysql-worker} script that is
provided with the MySQLTargetAlgorithmEvaluator. To have the worker execute in
a distributed way, the user simply schedules the execution of this command on their
cluster using the appropriate mechanisms\textsuperscript{5}.

One additional benefit concerns the Offline Batch Execution example in
Section 3.1.4. The TargetAlgorithmEvaluator interface contains a method
\texttt{areRunsPersisted()} which will return \texttt{true} if the application can exit such that runs
will still be processed. This allows for a simplified workflow in larger experiments
that primarily follow the Offline Batch Execution workflow. For example, if one needs
to compute the VBS over 10 different instance sets, instead of having to start and
potentially background 10 different processes, one can execute them serially, each
inserting the necessary runs into the database. Later when the runs are done, the
user can re-execute the processes serially, and as expected by our conceptual model,
the runs will be done now and can be processed, as if they were cached. This is the

\textsuperscript{5}This is another instance where AEATK operates differently than HAL, in that it does not do any
cluster management, and the user is responsible for scheduling the workers. This has not historically
been an impediment, as most clusters provide an example script for executing commands and these
are sufficient to distribute the workers.
standard workflow for the smac-validate utility.

3.5.2 Distributed vs Local Execution Performance

At an API level, the distinction between local and distributed target algorithm execution is non-existent. However, in terms of actual meta-algorithmic performance, we would certainly expect a difference between local and distributed execution. We feel that the two most pertinent parameters of interest are the amount of time that individual target algorithm evaluations take (the cutoff time), as well as the number of runs scheduled at any one time to the Target Algorithm Evaluator (the batch size).

A batch size of 1 captures the Online Single Execution use case described in Section 3.1.4, in which the application submits runs to the Target Algorithm Evaluator one by one. As the batch size gets larger we expect it more closely follows Online Batch Execution, before finally being more similar to the Offline Batch Execution, where a large number of runs is scheduled in a single batch.

In this experiment, we examined how long 128 runs take to execute for cutoff times from \(2^{-6}, 2^{-5}, ..., 2^1\) seconds and batch size from \(1, 2, 4, ..., 128\). A sample point consists a combination of cutoff time and batch size, and we measured the time immediately before the first `evaluateRun()` call, to when the last call to `evaluateRun()` returns (corresponding to all the runs being completed). Each combination of settings was measured 20 times, for a total of 1,980 total sample points. We compared the differences between the `CommandLineTargetAlgorithmEvaluator` detailed in Section 3.4 versus the time taken for the `MySQLTargetAlgorithmEvaluator` detailed in Section 3.5. The `CommandLineTargetAlgorithmEvaluator` was configured to permit up to 16 runs to be executed concurrently. The `MySQLTargetAlgorithmEvaluator` used a distinct pool for each sample, and simultaneously had 16 workers scheduled to process the jobs. Finally, for the `MySQLTargetAlgorithmEvaluator` version, we set the client poll frequency to 0.5 seconds and the worker poll frequency to 1 second. The target algorithm being executed is a python application that busy-waits until the required cutoff time has passed. Target algorithm runs are limited by wall-time rather than system time, because at the smaller scales system time is a relatively coarse measure (see Bryant and O’Hallaron [2002]); furthermore, at these small scales ( \(\sim 15\) ms), the interference from polling every 500 ms is expected to be negligible.

Each sample was scheduled on a cluster in shuffled order, and then processed in FIFO order. The same database and file system is used for all points, and there is a potentially for interaction effects to occur, but we expect that these effects would be minor and mitigated by the randomized order. This experiment was executed on
Table 3.1: System configuration for performance comparison between distributed and local execution.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Ubuntu Server 14.04.1 LTS x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2×Intel Xeon Processor E5-2650 v2 (8 cores / CPU)</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GiB</td>
</tr>
<tr>
<td>Kernel Version</td>
<td>3.13.0-24-generic</td>
</tr>
<tr>
<td>Java Version</td>
<td>1.7.0_45</td>
</tr>
</tbody>
</table>

Figure 3.19 on page 34 outlines the processing time versus the cutoff time for a number of different batch sizes. Several aspects of the graph deserve to be mentioned:

1. For the CommandLineTargetAlgorithmEvaluator the overhead of target algorithm execution seems to be about on the order of $2^{-4}$ or 0.06 seconds, judging
Figure 3.20: Processing time versus cutoff time for the MySQLTargetAlgorithmEvaluator

by how the processing time levels out for smaller values.

2. There is negligible performance increase in scheduling more jobs than the available number of cores with the CommandLineTargetAlgorithmEvaluator, judging by the fact that the processing time only decreases up until 16 runs are scheduled concurrently.

Figure 3.20 on page 35 shows the outcome of an analogous experiment for the MySQLTargetAlgorithmEvaluator from which we make the following observations:

1. The overhead of target algorithm execution seems to be on the order of $2^{-2}$ seconds or 0.25 seconds.

2. Processing time shows a similar improvement as the batch size increases, but the improvements continue past 16, especially for small cutoff times. This is to be expected, and with a batch size of 16, when the worker completes a run and polls for a next, there are not any waiting as the client must poll for completion and
then submit the next run. With larger batch sizes, the utilization improvement continues. Additionally, the workers will automatically adapt the number of runs they poll from the database in a single trip, if they detect that the runs are very short.

3. The processing time for the **MySQLTargetAlgorithmEvaluator** has much more variance than that for the **CommandLineTargetAlgorithmEvaluator**.

Another way of looking at the same data is to consider the overhead of processing the runs, defined as: \( \frac{\text{Actual Time}}{\text{Expected Time}} \). In this calculation, we account for the 16 available cores, so for example, assuming the cutoff time is 1 second, the expected time to complete 128 runs with a batch size of 2 is 64 seconds, a batch size of 16 is 8 seconds, and a batch size of 64 is 8 seconds.

Figure 3.21 on page 36 shows the overhead as batch size increases. From this data we notice that, except for the 0.0625 line, which has a lot of variation, the overhead is independent of batch size and is less than \( 2 \times \) for \( \geq 125 \) ms.

The corresponding plot for the **MySQLTargetAlgorithmEvaluator** is shown in Figure 3.22 on page 37. We notice that in this case the overhead is far greater, in
some cases over $10\times$ as high in the slowest case. Here, batch size does seem to affect the overhead significantly; unexpectedly, up until a size of 16 the overhead seems to consistently increase, and afterwards it decreases as expected.

In general, the results from these experiments suggest that for very short runtimes there is a considerable amount of overhead introduced by the MySQLTargetAlgorithmEvaluator, but this can be mitigated by increasing batch size.

In practice, better performance than seen in our experiments is achievable, especially when it becomes possible to increase the batch size further (as is fairly common when using the smac-validate utility to submit $> 100\,000$ runs in a single batch). To a limited degree, it is also possible to improve performance by decreasing the polling intervals, but this comes at the cost of increased load on the database. The MySQLTargetAlgorithmEvaluator is not ideal for Online Concurrent workloads with small cutoff times, and another distributed approach that does not use polling mechanisms would be preferred. Anecdotally, however, the fact that all of this is done via polling of the database has provided a considerable advantage to meta-algorithm designers, as this approach provides a transparent and easily modified view to the
3.6 Modification & Customization

In this section we outline how TargetAlgorithmEvaluator usage can be customized and changed to suite the particular needs of the meta-algorithm designer and the user. In Section 3.6.1, we outline how new TargetAlgorithmEvaluator implementations can be created by users and slotted into existing applications using AEATK. In Section 3.6.2, we provide an overview of other TargetAlgorithmEvaluator implementations, and finally, in Section 3.6.3, we outline other parts of AEATK that provide common functionality needed by all implementations.

3.6.1 New Implementations

There are a number of reasons why a new implementation of a TargetAlgorithmEvaluator may be necessary:

1. It can be useful to debug an algorithm by, for instance, having a specific sequence of results returned.

2. In some cases, the overheads associated with existing implementation may be very large. For instance, for very short runtimes, it can be more efficient to directly execute the target algorithm within the same process instead of doing a system() call\(^6\).

3. The target algorithm may not lend itself easily to wrapper execution; for instance, it might be a remote process, or something that prompts the user for input.

4. To interface with other meta-algorithmic libraries.

AEATK supports a plugin style architecture based on the Service Provider Interface (SPI) in Java. To get an instance of a TargetAlgorithmEvaluator in AEATK, one invokes the TargetAlgorithmEvaluatorBuilder.getTargetAlgorithmEvaluator() method and supplies a custom name, for instance CLI for the implementation discussed in Section 3.4 or MYSQLDB for the implementation discussed in Section 3.5. SPI is used to map this name to the specific implementation, which is then loaded dynamically. Figure 3.23 shows the required classes (on the right) needed to create a new implementation. The actual implementation is shown in the first row. The second row

\(^6\)This is only applicable if the target algorithm is written in Java and can be evaluated directly, or is executable in C/C++ using the Java Native Interface (JNI)
Figure 3.23: UML for new TargetAlgorithmEvaluator implementations

is an implementation for an Abstract Factory design pattern (Gamma et al. [1994]), which can instantiate the TargetAlgorithmEvaluator implementation. The third row allows the user to specify options applicable to the TargetAlgorithmEvaluator implementation, for instance in the CommandLineTargetAlgorithmEvaluator an option exists to set the number of concurrent executions.

As will be discussed in Section 3.7.1, this name can be specified in the command line. This allows someone to implement a new TargetAlgorithmEvaluator and then leverage existing meta-algorithmic techniques without needing to make any modifications to source code.

Supporting all of the features and mechanisms outlined in Section 3.3 such as asynchronous execution, observation, and early termination of runs can appear daunting, but AEATK provides mechanisms for simplifying this task. One mechanism that AEATK provides to simplify the task of implementing new TargetAlgorithmEvaluator is by supplying useful base types which can be extended (others will be discussed in Section 3.6.3). First note that a new TargetAlgorithmEvaluator need only implement one of evaluateRunAsync() and evaluateRun(). If evaluateRunAsync() is implemented, then evaluateRun() can always simply call evaluateRunAsync() and wait on a semaphore that will be triggered in a callback’s onSuccess() method on completion. Alternatively, if evaluateRun() is implemented, evaluateRunAsync() can simply start an-
other thread and in this thread call `evaluateRun()`\(^7\). AEATK provides the abstract subtype `AbstractSyncTargetAlgorithmEvaluator`, which can be extended and only requires the `evaluateRun()` method to be implemented, and the abstract subtype `AbstractAsyncTargetAlgorithmEvaluator`, which only requires the `evaluateRunAsync()` method to be implemented.

Additionally note that our conceptual model from Section 3.1.3 allowed us to view a run as a function evaluation, and it should be obvious that meta-algorithms would generally be agnostic to how long the evaluation took (This would not be true for instance if the API returned a token, and expected the meta-algorithm to poll this token for updates, in this world the meta-algorithm is very much aware of time.) If the evaluation were instantaneous there would be no reason to notify the observer, and consequently notifying the observer is optional. Additionally, as mentioned in Section 3.3, the `kill()` method is advisory and an implementation may chose to implement this method only partially\(^8\), if at all. Since implementations may at their discretion skip observer functionality and `kill()` method, and can rely on the library to handle asynchronous requests new implementations can be as simple as implementing a single procedural method. Consequently, while AEATK supports a flexible execution framework, users of custom meta-algorithms are not exposed to complexity beyond that needed for their purposes.

### 3.6.2 Existing Implementations

Beyond local execution outlined in Section 3.4 and distributed execution in Section 3.5, there are several additional implementations of the `TargetAlgorithmEvaluator` interface available in AEATK.

There are a number of `TargetAlgorithmEvaluator` implementations that are useful for debugging (names in brackets). For instance, there are implementations that return a constant response (`CONSTANT`), allow the user to specify a predefined sequence of responses (`PRELOADED`), or return a randomized response (`RANDOM`). Another implementation (`BLACKHOLE`) simply drops asynchronous requests and is useful for unit testing.

A number of `TargetAlgorithmEvaluator` have more general applicability. The `TargetAlgorithmEvaluator` (HAL) is used by meta-algorithm implementations when

---

\(^7\)It should be noted that this solution is generally not scalable to a large number of asynchronous threads.

\(^8\)An example of a partial implementation, is a `TargetAlgorithmEvaluator` that cannot actually terminate active runs, but can terminate those runs in a batch that have not actually physically started executing.
running from within HAL. Another implementation (ANALYTIC) contains some closed-form analytic test functions in Java, for instance Branins’ function (see e.g., Molga and Smutnicki [2005]). Finally, an implementation exists (IPC) that can communicate with another process via UDP or TCP/IP; typically, this is used as a way of interacting with other languages, or across the network. Finally another implementation (SURROGATE) allows users to build a model of their target algorithms performance and use the model in place of the actual algorithm, similar to work done in Eggensperger et al. [2015].

### 3.6.3 Execution Customization

When actually performing target algorithm evaluations, a number of problems can arise. In some cases, these problems are entirely within the purview of the actual implementation, such as the problem of process leakage described in Section 3.4.3. In other cases, the problems can be fairly general such as evaluations that report CRASHED sporadically or that are exceeding their requested cutoff time. Alternatively, there are features that are useful for any and all TargetAlgorithmEvaluator implementations, such as caching, or logging information about runs being processed. Consequently, there are issues and features that are general and apply regardless of which TargetAlgorithmEvaluator implementation or meta-algorithm is being used.

It would introduce an unnecessary burden to require that every implementation of the TargetAlgorithmEvaluator interface deal with these cases explicitly. At the same time, we do not want to push the complexity of dealing with these issues onto the meta-algorithm designer either. Ultimately, we want to deal with this at a layer between user-supplied TargetAlgorithmEvaluator implementations, which should be simple to implement, and the meta-algorithm, which should have a simple way of doing evaluations.

The solution to this problem comes in the form of the Decorator design pattern (Gamma et al. [1994]). After retrieving the requested TargetAlgorithmEvaluator implementation, it is modified by applying a number of decorators to provide additional functionality or properties. One such decorator, shown in Figure 3.24, automatically retries runs that are crashed a user defined number of times. As mentioned in Section 3.6.1, invoking the TargetAlgorithmEvaluatorBuilder.getTargetAlgorithmEvaluator() method retrieves the TargetAlgorithmEvaluator instance; By default this instance is decorated 14 times, and within the AEATK API, there are ~40 decorators available that serve different purposes, most of which can be configured in the code and/or by the user via the command line (see Section 3.7.1).
The decorators in AEATK correspond (loosely) to the following categories, a full reference is available in Appendix A:

**Debugging** These decorators aid in debugging applications. For instance, some allow users to control logging of every evaluation, and will check that the \texttt{TargetAlgorithmEvaluator} is properly shutdown when the program ends. Finally, others check various invariants, such as that the client is not asking for duplicate target algorithm evaluations in the same request or that the \texttt{TargetAlgorithmEvaluator} implementation is not misbehaving (e.g., a \texttt{TargetAlgorithmEvaluator} could due to a concurrency bug return the wrong \texttt{AlgorithmRunResult} objects).

**Functionality** These decorators implement some added functionality; for instance, one of the decorators implements the \texttt{getOutstandingEvaluations()} method (instead of requiring the base-type to do it) and another allows transforming the response values according to any analytic function. Yet another decorator simulates delays to allow for reproducibility and debugging (e.g., if a run reports that it took 5 seconds to execute, this decorator will wait, if necessary, to ensure that it took 5 seconds from request to response).

**Helpers** These decorators are viewed as providing generally helpful functionality. One decorator terminates runs that have taken 10 times their cutoff time to execute. Another decorator retries crashed runs, as described previously. Another decorator will notify observers of the wall time that runs have taken to execute, if the target algorithm evaluator is not aware of the actual runtime (e.g., if the wrapper does not use the modified version of \texttt{runsolver}). Another decorator

Figure 3.24: Sequence diagram showing how CRASHED runs can be dealt with
always calls the observer with the final result (this ensures that if key logic in a meta-algorithm is done in an observer, and the TargetAlgorithmEvaluator does not implement observation, it works and appears as if it was a cached run with observation). Another decorator produces a report of utilization overtime of the TargetAlgorithmEvaluator.

**Resource** These decorators generally control the flow of execution. Several decorators implement caching (in memory and file based). Another decorator limits the number of requests that can go past it at the same time (imagine wanting to do 1,000 runs simulating 16 cores with random response values; using this decorator and the one that simulates delays would achieve this effect). Another decorator allows runs to be submitted as “low priority” (will only run if nothing else needs to). Another decorator ensures that the evaluateRunAsync() method never blocks. Finally, another decorator submits requests to several decorated implementations that depend on a configurable policy (e.g., one might run an algorithm for 1 second locally before submitting it to the database).

**Safety** These decorators are concerned with ensuring safety and correctness. Decorators here may chose to treat crashes as aborts, or if the first run crashes, treat that as an abort (most of the time if the first run crashes, then every run crashes, and something is broken). Other decorators check that instances always report consistent results for problem instance satisfiability (either specified in a file, or just based on other runs). Another, will log warnings if the overhead detected in the target algorithm is high. Finally, another will warn if the TargetAlgorithmEvaluator does not seem to be making progress (the most common reason for this to occur is not scheduling workers when using the MySQLTargetAlgorithmEvaluator described in Section 3.5).

### 3.7 Easy Distribution

The final aspect that AEATK assists application developers with is easy distribution of meta-algorithmic procedures. McConnell [2004] notes that once a simple program has been completed, it takes twice as much work to release it for general consumption (citing Brooks [1995] & Shull et al. [2002]). Admittedly, AEATK (and, to a lesser extent, meta-algorithmic techniques in general) is primarily research focused; however, there is a strong desire for meta-algorithmic techniques to see wider adoption in both academia and industry. To do this, meta-algorithmic techniques should be reliable,
robust and easy to use. In previous sections, we have talked about reliability and robustness, and in this section we will discuss two aspects of using AEATK applications from the perspective of an end user: Supplying input to the meta-algorithm on the command-line in Section 3.7.1 and its output in Section 3.7.2.

3.7.1 Command-line Interface

In Sections 3.5 and 3.6.1, we alluded to being able to change the TargetAlgorithmEvaluator fairly easily, and while it is easy to envision how this could be done in code, it is unclear how a user could simply create a new TargetAlgorithmEvaluator implementation and then not have to change the meta-algorithm implementation. Additionally, in Section 3.6.3, we mentioned a host of options that can provide additional functionality that may be of use to TargetAlgorithmEvaluator implementers, meta-algorithmic designers, but also to end users of those meta-algorithms. Both selection of the TargetAlgorithmEvaluator and configuration of it and the decorators are controlled through a standard command line interface library included with AEATK called JCommander (http://www.jcommander.org).

JCommander allows developers to declare a standard Java object and then use annotations to have those fields exposed on the command line. JCommander also allows these objects to be composed (in JCommander terminology; they are parameter delegates). AEATK supplies a series of delegates that can allow for a large amount of objects and choices to be specified on the command line as needed. We also modified JCommander to provide additional support for reading options files (including default files located in the user home directory). Finally, it was also extended to provide more flexibility in displaying help to the user. For example, in JCommander options can either be hidden or not, but in our modified version different levels of help are available to the user of an AEATK application by using the option --help-level with values: BASIC, INTERMEDIATE, ADVANCED, or DEVELOPER.

In Section 3.5 (page 32), we noted that to changing an existing application to leverage parallel resources can be done with a single argument, and at this point, we can see how and when this is true. New TargetAlgorithmEvaluator implementations can specify options (third row of Figure 3.23). These options are then read by JCommander as well as the name from the TargetAlgorithmEvaluatorFactory.getName() method. On the command-line, the user can select this TargetAlgorithmEvaluator by using its name with the --tae parameter, which selects base-type of the TargetAlgorithmEvaluator returned by the
3.7.2 Output

While the previous section detailed how the user provides input to AEATK applications, AEATK also provides some support for handling output of an application. For direct user feedback, most applications rely on logging; AEATK uses SLF4J (http://www.slf4j.org/), a logging library in Java that acts as a facade and front end to a number of other logging libraries. Similar to how AEATK can dynamically change the TargetAlgorithmEvaluator an application is using, SLF4J allows the user to control which logging library they use. By default, AEATK also supplies logback (http://logback.qos.ch/) and some parameter delegates to control the logging options (so users can for instance change the log level of an application). It is important to note that the coupling of SLF4J does not introduce a serious restriction on the meta-algorithm, as SLF4J allows users to switch logging frameworks at runtime. For instance, HAL uses the Apache Commons Logging, and so when an AEATK meta-algorithm is running inside HAL, SLF4J will redirect log messages to the same framework instead of logback.

3.8 Conclusions and Limitations

In Section 3.1.2, we outlined the principal design goals that drove the development of AEATK. Algorithms should be easy for users to execute – the mechanism for this was discussed in Section 3.2 and 3.3. Meta-algorithmic designers can evaluate algorithms with a simple method call, and yet the API also provides much richer methods; however, users only need to deal with the complexity they need. In part because of this philosophy, users of meta-algorithms can also easily leverage parallel resources, as discussed in 3.5 and 3.7.1. Additionally, the plugin (see Section 3.6.1) and decorator structure (see Section 3.6.3) of AEATK allows meta-algorithms leveraging AEATK to be versatile and adaptable to users needs. Finally, AEATK provides some support for making redistributable applications described in Section 3.7, as a toolkit users can pick and chose the functionality they need (MySQL and JCommander are optional), and the coupling to the logging framework, SLF4J, is arguably not likely to be a serious impediment to future users.

The reader might wonder where all the complexity related to algorithm execution went; the answer to this question is that, to a large degree, it has been hidden from both the meta-algorithm designer and the TargetAlgorithmEvaluator imple-
menter. This is aided by design principles such as Inversion of Control, which the
callback and observer structure follow, and leads to more extensible code. Indeed,
the simplicity that both groups of AEATK users benefit from, is offset by the com-
plexity of implementing some decorators. For example, the decorator which retries
runs if they are CRASHED is surprisingly complex, because of the callback mecha-
nism. Additionally, reasoning about decorator order can be surprisingly complex. For
instance, the waitForOutstandingEvaluations() method may not behave as the
meta-algorithm expects, if another decorator transparently reschedules runs (e.g., be-
cause they crashed and a decorator is retrying them). Most meta-algorithm designers
and TargetAlgorithmEvaluator implementers are not exposed to this, however, as
the ordering is an internal aspect of AEATK, and this issue only arises when new
decorators need to be created that cannot be inserted near the base or at the top of
the decorator chain.

Much of AEATK grew organically driven by porting SMAC to Java, and also by
needs of users on other projects. As a general rule, no idea would be implemented, no
matter how good, unless there was an actual need for it (similar to the principle in
extreme programming). The four best ideas that have not been implemented at this
point are:

1. Allowing meta-algorithms to take multiple samples per point as discussed in
   Section 3.1.3.

2. A distributed TargetAlgorithmEvaluator implementation that does not rely
   on a MySQL server being available.

3. Allowing the configuration of a target algorithm to be specified using a Map
   instead of an instance of the ParameterConfiguration class, would decouple
   the execution framework from the PCS format.

4. Allowing decorators for the TargetAlgorithmEvaluator to be loaded from
   plugins.

   The first three of these ideas are not particularly onerous to implement, but the
   final one would require some API changes and careful consideration.

   In general, however, AEATK has been successful, and it has facilitated work that
gave rise to several publications, including in Thornton et al. [2013a] and Hutter

   One can achieve this in some cases by creating a new TargetAlgorithmEvaluator as in Section
   3.6.1, and then supplying an option that loads the actual base type manually, but this is coarse and
does not give you the ability to select where in the decorator structure it appears.
et al. [2013] where AEATK facilitated data collection. The reference implementations for detecting parameter importance via ablation analysis Fawcett and Hoos [2013], and functional ANOVA Hutter et al. [2013] were also written in AEATK. Finally, beyond forming the basis for the re-implementation of SMAC, efforts are under-way by others to use AEATK as the reference implementations in SatZilla by Chris Cameron and Alexandre Fréchette, ParamILS 2.0 by Aymeric Blot and SATFC by Alexandre Fréchette.
Chapter 4

SMAC in AEATK

4.1 Introduction

Sequential Model-based Algorithm Configuration (SMAC) (Hutter et al. [2011b]) is an algorithm which solves the algorithm configuration problem defined in Section 2.3. SMAC was originally implemented in Matlab; in this chapter, we describe the implementation of the SMAC algorithm in AEATK & Java (hereafter referred to as the Java implementation). The Java version is now the reference implementation for SMAC and has been used to configure state-of-the-art algorithms for a broad range of combinatorial problems (see e.g., Hoos et al. [2012], Lindauer et al. [2015], Seipp et al. [2015]), and machine learning tasks (see e.g., Domhan et al. [2014], Thornton et al. [2013a], Eggensperger et al. [2013]).

4.2 Algorithm Overview

As discussed in Section 2.3.2, SMAC traces its origins to the EGO algorithm (Jones et al. [1998]), which (after an initial design) alternated between building a model and evaluating the most promising point found from the model. Algorithm 4.1 presents the pseudo-code for the SMAC algorithm in more detail (it is adapted from Hutter et al. [2011b] but also includes details from Hutter et al. [2011a]). At the highest level, SMAC uses a single run of the default as an initial design, and then alternates between building models (random forests) and evaluating promising configurations selected from it. Promising configurations are selected from the model by performing ten local searches using a one-exchange neighbourhood\(^1\). The results of these local searches are

\(^1\)For continuous parameters, the set of neighbours for a parameter \( \theta \) with value \( x \) normalized to (0,1) is sampled 4 times from \( x' \sim \mathcal{N}(x, 0.2) \), with rejection if \( x' \notin [0,1] \).
mixed with 10000 random samples and sorted in order of best acquisition function value\(^2\). These 10010 configurations are there interleaved with randomly sampled configurations, and SMAC evaluates these configurations until enough time has passed and it decides to rebuild its model. Algorithm 4.1 shows high level pseudocode for SMAC.

### 4.3 AEATK Implementation

The SMAC project heavily relies on the AEATK library for target algorithm evaluation, input parsing and output. Existing parameter delegates (see Section 3.7.1) take care of most of the input validation and typically catch problems with inputs early. The principal functionality of SMAC is distributed amongst five classes:

- **AbstractAlgorithmFramework** contains the core functionality of the SMAC algorithm (as presented in Algorithm 4.1). By itself, this class implements the ROAR algorithm specified in Hutter et al. [2011b], where no model is learnt and configurations are selected randomly.

- **SequentialModelBasedAutomaticConfigurator** is a subtype of **AbstractAlgorithmFramework** and contains the logic for building the model and performing local search.

- **Validator** performs validation of the configurations found during the execution of SMAC, typically after SMAC has executed but also as a stand alone utility. This utility can be used for not just SMAC, but also ParamILS (Hutter et al. [2007b]) and GGA (Ansotegui et al. [2009]).

- **SMACExecutor** is the program entry-point and parses command-line arguments.

- **SMACBuilder** builds all of the necessary dependencies for running SMAC (e.g., retrieving the instances, setting the **TargetAlgorithmEvaluator**, etc.). Both AEATK and SMAC following a coding convention called constructor-based dependency injection\(^3\). While this in general leads to more portable code, it has the side

---

\(^2\)Typically exponential expected improvement as defined by Hutter et al. [2009a].

\(^3\)Dependency injection means that a class is given all dependencies it needs by its caller, and does not resolve the dependencies internally (e.g., the **AbstractAlgorithmFramework** is given the **TargetAlgorithmEvaluator** from outside). Constructor-based means that all dependencies are supplied in the constructor, instead of using **setter** methods to supply them. An advantage of this approach is that once objects are constructed they are always in a usable state and have all necessary dependencies.
effect of making the constructors quite large. The SMACBuilder class simplifies construction of the AbstractAlgorithmFramework and SequentialModelBasedAutomaticConfigurator objects.

Figure 4.1 shows a sequence diagram which represents how these classes interact. Most of the core functionality depicted in Algorithm 4.1 is contained within the run() and intensify() methods in the AbstractAlgorithmFramework class.
4.4 Differences Between MATLAB and Java Implementations

While both implementations follow the algorithm outlined in Algorithm 4.1, there are some minor differences between them:

1. Following the ideas outlined by Hoos [2012a], the Java version exposes almost every design parameter on the command line, although most users are shielded from this by AEATKs support for different levels of options as mentioned in Section 3.7.1.

2. The Java version supports independently seeding the different pseudo-random number generators of different components (e.g., seeds for building models, random configuration generation, etc.), which is particularly useful for localized debugging.

3. The MATLAB version has support for many types of models, such as Gaussian Processes, whereas the Java version is coupled very tightly to the Random Forest models.

4. The Java version supports pre-loading the random forest model with existing data before running. This can be beneficial if existing data is available, a feature that was used for instance by Feurer et al. [2015].

5. The Java version supports tracking where and when configurations were instantiated and how far they progressed, which enables one to answer queries like how many of the final incumbents were selected via expected improvement, and how many were selected as a result of random search.

6. The Java version is better documented, both at the code level but also for users. It includes a quick start guide, example scenarios and a comprehensive manual.

7. The MATLAB and Java versions have different command line interfaces, but utilize the same file formats (similar to ParamILS).\(^5\)

---

\(^4\)The MATLAB and Java version share the exact same random forest code, as the MATLAB version uses the implementation written in Java.

\(^5\)Technically, the feature file format is incompatible, since the Java version requires a header to name all instance features (whereas in MATLAB one can just supply a vector of doubles). This change was mandated after discovering instance sets with features sorted in different order.
8. If the MATLAB version re-executes a run with a longer cap-time it *forgets* about the previous execution entirely, and the preceding run does not count towards time limits. In Java, the preceding run is properly counted.

9. In contrast to the Matlab version, the Java version can be run easily, without the need for expensive Matlab licenses or time-consuming setup of the Matlab Runtime Environment. This has facilitated the wide-spread use of SMAC.

Although now included in AEATK, the Java version of SMAC includes utilities for detecting problems (e.g., missing instances, missing features, syntax errors in files, etc...) with an experiment before they are begun, and can do this across multiple experiments at the same time. Another included utility allows users to call their target algorithm exactly as SMAC would, to help debug issues that may not be obvious when users simulate calling their algorithm from the command line/shell.\(^6\)

### 4.5 Experimental Comparison

In this section we compare the performance of the MATLAB and Java implementations of SMAC. For reference we use a subset of the scenarios presented in Hutter et al. [2011b] for the SAPS (Hutter et al. [2002]), and SPEAR (Babić and Hutter [2007]) SAT solvers, that are contained with the AClib project (http://www.aclib.net/). Additionally, our runs for both the Matlab and the Java version differ in that they use the *adaptive capping mechanism* outlined in Hutter et al. [2011a], which is the standard experimental set-up for runtime optimization. Finally, we included the SAPS-QWH scenario introduced in Hutter et al. [2009b]. Table 4.1 outlines the parameters for each configuration scenario. For both MATLAB and Java implementations of SMAC, the *Tuner Time* is calculated as the sum of the CPU time of SMAC, plus the greater of 0.1 and the reported CPU time of each individual target algorithm run\(^7\).

#### 4.5.1 Computational Environment

This experiment was performed using the META cluster at the University of Freiburg, Germany. Table 4.2 lists the hardware and software resources of the machines in this cluster. Multiple runs of each scenario were performed (the actual numbers are in Section 4.5.2). For submission to the Sun Grid Engine clustering platform, runs of both

\(^6\)A common example is when users are relying on features of their shell, for example quotation marks or the “∼” character expecting it to be turned into their home directory.

\(^7\)Other configurators may calculate this differently.
MATLAB and Java SMAC were aggregated into groups of 16 that would be dispatched to an entire node. Then, each grouping of 16 runs were executed concurrently, which each individual run being assigned a distinct processor affinity (i.e., run \( n \) of SMAC and its target algorithm executions must all compete for a single core on the machine).

### 4.5.2 Results

Figures 4.2, 4.3, and 4.4 show the value of the objective function on the training set over time for both configurators. Qualitatively, the graphs look very similar, and while there are some noticeable differences, neither configurator tends to dominate the other based upon a visual inspection of the graphs.

Tables 4.3 and 4.4 outline the final median performance for both configurators. We use the Mann-Whitney U test with a Bonferroni correction of 15 to test for a distributional difference at a significance level of 0.01 and find only 4 scenarios where they diverge\(^8\). In the cases where there is a statistically significant difference we note that the difference in medians occurs at the 10s of milliseconds level for IBM-Spear-med, the millisecond level for SWV-Spear-med and SWV-Spear-q095 and at the sub millisecond level for IBM-Spear-q025. At these levels the mechanism used by these scenario to determine runtime are very noisy (see Chapter 9 of Bryant and O’Hallaron [2002]). Consequently, we believe that the MATLAB and Java implementations of SMAC are comparable.

### 4.5.3 Conclusions & Future Work

In this chapter, we outlined the re-implementation of the SMAC algorithm from Hutter et al. [2011b] in Java, which now is the reference implementation of SMAC. There are several interesting directions in SMAC which one might consider exploring. The first is to meta-configure SMAC (i.e., tune SMAC’s parameters, many of which were only set heuristically), as much of the parameter space is exposed on the command line, and afterwards to conduct an analysis of parameter importance via either the method of Fawcett and Hoos [2013] or Hutter et al. [2013], to improve SMAC’s performance. Another important question is how much of the improvement SMAC discovers in a configuration space is due to the choice of instance selection and its use of random configurations.

\(^8\)For the skeptical reader, if a significance level of 0.05 was desired and no Bonferroni correction is applied, then only the performance on two additional scenarios, SWGCp075-PAR1-saps and SWGCp095-PAR1-saps, would be deemed significantly different.

53
Figure 4.2: Median Performance and Interquartile range for SAPS Single Instance Scenarios.
Algorithm 4.1: Simplified SMAC Algorithm

R keeps track of all target algorithm runs performed so far and their performances (i.e., SMBO’s training data \(\{(\theta_1, x_1, o_1), \ldots, (\theta_n, x_n, o_n)\}\)), \(\mathcal{M}\) is SMBO’s model, \(\Theta_{\text{new}}\) is a list of promising configurations, and \(t_{\text{fit}}\) and \(t_{\text{select}}\) are the runtimes required to fit the model and select configurations, respectively.

\[\begin{align*}
\text{Input} & : \text{Target algorithm } A \text{ with parameter configuration space } \Theta; \text{ instance set } \Pi; \text{ cost metric } \hat{c} \\
\text{Output} & : \text{Optimized (incumbent) parameter configuration, } \theta_{\text{inc}} \\
1 & [R, \theta_{\text{inc}}] \leftarrow \text{Initialize}(\Theta, \Pi) \\
2 & \text{repeat} \\
3 & \quad \mathcal{M} \leftarrow \text{learnModel}(R) \\
4 & \quad \Theta_{\text{new}} \leftarrow \text{selectConfigurations}(\mathcal{M}) \\
5 & \quad t_{\text{intensify}} \leftarrow \text{duration of previous function call} \\
6 & \quad //\text{intensify()} \text{ is in-lined here} \\
7 & \quad i \leftarrow 0 \\
8 & \quad \text{foreach } \theta_{\text{new}} \in \Theta_{\text{new}} \text{ do} \\
9 & \quad \quad //\text{challengeIncumbent()} \text{ is in-lined here} \\
10 & \quad \quad R \leftarrow \text{ExecuteNewRunForIncumbent}(R, \theta_{\text{inc}}) \\
11 & \quad \quad //\text{Number of Runs to do per stage} \\
12 & \quad \quad N \leftarrow 1; \\
13 & \quad \text{while true do} \\
14 & \quad \quad S = \{(\pi_i, s_i)\}_{i=1}^{N} \leftarrow \text{NextPISPSForChallenger}(N, \theta_{\text{new}}, \theta_{\text{inc}}) \\
15 & \quad \quad \text{foreach } (\pi, s) \in S \text{ do} \\
16 & \quad \quad \quad (\pi, s, \kappa) \leftarrow \text{ComputeCapTimes}(\pi, s) \\
17 & \quad \quad \quad R \leftarrow \text{ExecuteRun}(R, \theta_{\text{new}}, \pi, s, \kappa) \\
18 & \quad \quad \quad T = \{(\pi_i, s_i)\}_{i=1}^{M} \leftarrow \text{GetPISPSRunForConfig}(\theta_{\text{inc}}) \\
19 & \quad \quad \quad V = \{(\pi_i, s_i)\}_{i=1}^{L} \leftarrow \text{GetPISPSRunForConfig}(\theta_{\text{new}}) \\
20 & \quad \quad \quad o_{\text{inc}} = \text{ComputeObjectiveOverPISPS}(\theta_{\text{inc}}, V) \\
21 & \quad \quad \quad o_{\text{new}} = \text{ComputeObjectiveOverPISPS}(\theta_{\text{new}}, V) \\
22 & \quad \quad \text{if } o_{\text{new}} > o_{\text{inc}} \text{ then break ;} \\
23 & \quad \quad \text{else if } T = V \text{ then } \theta_{\text{inc}} \leftarrow \theta_{\text{new}}; \text{ break ;} \\
24 & \quad \quad \text{else } N \leftarrow 2 \cdot N; \\
25 & \quad \quad \text{if time spent in this call to this procedure exceeds } t_{\text{intensify}} \text{ and } i \geq 2 \\
26 & \quad \quad \quad \text{then break ;} \\
27 & \quad \quad \quad i \leftarrow i + 1 \\
28 & \quad \text{endwhile} \\
29 & \quad \text{endforeach} \\
30 & \quad \text{until total time budget for configuration exhausted;} \\
31 & \text{return } \theta_{\text{inc}}; \\
\end{align*}\]
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Instances</th>
<th>Features</th>
<th>Cutoff (s)</th>
<th>Tuner Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-Spear-med</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>IBM-Spear-q025</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>IBM-Spear-q075</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>QCPmed-PAR1-saps</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>QCPq075-PAR1-saps</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>QCPq095-PAR1-saps</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>QWHmed-PAR1-saps</td>
<td>1</td>
<td>0</td>
<td>1.0</td>
<td>3600</td>
</tr>
<tr>
<td>SWGCPmed-PAR1-saps</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>SWGCPq075-PAR1-saps</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>SWGCPq095-PAR1-saps</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>SWV-Spear-med</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>SWV-Spear-q075</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>SWV-Spear-q095</td>
<td>1</td>
<td>0</td>
<td>5.0</td>
<td>1800</td>
</tr>
<tr>
<td>QCP-PAR10-saps</td>
<td>955</td>
<td>125</td>
<td>5.0</td>
<td>18000</td>
</tr>
<tr>
<td>QCP-spear-PAR10</td>
<td>976</td>
<td>125</td>
<td>5.0</td>
<td>18000</td>
</tr>
<tr>
<td>SWGCP-PAR10-saps</td>
<td>1000</td>
<td>125</td>
<td>5.0</td>
<td>18000</td>
</tr>
<tr>
<td>SWGCP-spear-PAR10</td>
<td>1000</td>
<td>125</td>
<td>5.0</td>
<td>18000</td>
</tr>
</tbody>
</table>

Table 4.1: Configuration scenarios for MATLAB and Java SMAC Implementations. Additionally, all of the multi-instance scenarios used the continuous parameter configuration spaces. For single instance scenarios involving the SPEAR solver, the configuration space was discretized.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Ubuntu Server 14.04.1 LTS x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2×Intel Xeon Processor E5-2650 v2 (8 cores / CPU)</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GiB</td>
</tr>
<tr>
<td>Kernel Version</td>
<td>3.13.0-24-generic</td>
</tr>
<tr>
<td>Java Version</td>
<td>1.7.0_45</td>
</tr>
</tbody>
</table>

Table 4.2: System configuration for all configuration scenarios
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Unit</th>
<th>Default</th>
<th>Java Median</th>
<th>#</th>
<th>MATLAB Median</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM-Spear-med</td>
<td>[-10^0 s]</td>
<td>2.300</td>
<td>1.593</td>
<td>160</td>
<td>1.513</td>
<td>160</td>
</tr>
<tr>
<td>IBM-Spear-q025</td>
<td>[-10^-1 s]</td>
<td>5.516</td>
<td>3.693</td>
<td>160</td>
<td>3.695</td>
<td>160</td>
</tr>
<tr>
<td>QCPmed-PAR1-saps</td>
<td>[-10^-2 s]</td>
<td>3.396</td>
<td>2.561</td>
<td>160</td>
<td>2.534</td>
<td>160</td>
</tr>
<tr>
<td>QCPq075-PAR1-saps</td>
<td>[-10^-1 s]</td>
<td>2.500</td>
<td>0.919</td>
<td>160</td>
<td>0.921</td>
<td>159</td>
</tr>
<tr>
<td>QCPq095-PAR1-saps</td>
<td>[-10^0 s]</td>
<td>3.934</td>
<td>0.463</td>
<td>160</td>
<td>0.481</td>
<td>160</td>
</tr>
<tr>
<td>QWHmed-PAR1-saps</td>
<td>[-10^-2 s]</td>
<td>4.212</td>
<td>2.738</td>
<td>32</td>
<td>2.838</td>
<td>31</td>
</tr>
<tr>
<td>SWGCPq075-PAR1-saps</td>
<td>[-10^0 s]</td>
<td>3.024</td>
<td>0.110</td>
<td>160</td>
<td>0.113</td>
<td>144</td>
</tr>
<tr>
<td>SWGCPq095-PAR1-saps</td>
<td>[-10^0 s]</td>
<td>2.687</td>
<td>0.123</td>
<td>160</td>
<td>0.127</td>
<td>160</td>
</tr>
<tr>
<td>SWV-Spear-med</td>
<td>[-10^-2 s]</td>
<td>6.631</td>
<td>3.710</td>
<td>160</td>
<td>3.724</td>
<td>160</td>
</tr>
<tr>
<td>SWV-Spear-q075</td>
<td>[-10^0 s]</td>
<td>5.000</td>
<td>0.506</td>
<td>160</td>
<td>0.507</td>
<td>160</td>
</tr>
<tr>
<td>SWV-Spear-q095</td>
<td>[-10^0 s]</td>
<td>5.000</td>
<td>0.506</td>
<td>160</td>
<td>0.507</td>
<td>160</td>
</tr>
</tbody>
</table>

Table 4.3: Median training performance of MATLAB and Java implementations of SMAC, along with the default performance for single-instance scenarios from Hutter et al. [2011b]. The number of independent runs for each is denoted in the # column. Some runs of MATLAB did not complete successfully. Bold-faced entries represent configurator performance that is significantly better under a Mann-Whitney U Test with significance level of 0.01 (a Bonferroni correction was applied so the individual p-values are less than $6.667 \times 10^{-4}$).

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Unit</th>
<th>Default</th>
<th>Java Median</th>
<th>#</th>
<th>MATLAB Median</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>QCP-PAR10-saps</td>
<td>[-10^0 s]</td>
<td>9.367</td>
<td>3.484</td>
<td>32</td>
<td>3.557</td>
<td>32</td>
</tr>
<tr>
<td>QCP-spear-PAR10</td>
<td>[-10^0 s]</td>
<td>1.897</td>
<td>0.653</td>
<td>32</td>
<td>0.616</td>
<td>31</td>
</tr>
<tr>
<td>SWGCP-PAR10-saps</td>
<td>[-10^0 s]</td>
<td>15.094</td>
<td>0.107</td>
<td>32</td>
<td>0.108</td>
<td>32</td>
</tr>
<tr>
<td>SWGCP-spear-PAR10</td>
<td>[-10^0 s]</td>
<td>6.987</td>
<td>6.027</td>
<td>32</td>
<td>6.018</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 4.4: Median performance of MATLAB and Java implementations of SMAC, along with the default performance for multi-instance scenarios from Hutter et al. [2011b]. The number of independent runs for each is denoted in the # column. Some runs of MATLAB did not complete successfully. Bold-faced entries represent configurator performance that is significantly better under a Mann-Whitney U Test with a significance level of 0.01 (a Bonferroni correction for 15 samples was applied so the individual p-values are less than $6.667 \times 10^{-4}$).
Figure 4.3: Median Performance and Interquartile range for SPEAR Single Instance Scenarios.
Figure 4.4: Median Performance and Interquartile range for Multi-Instance Scenarios
Chapter 5

Parallel Automatic Configuration

5.1 Introduction

There are many possible ways in which an automatic configurator can leverage parallel resources to achieve improved configuration outcomes. In this chapter, we first examine two current approaches and then explore a simple mechanism to increase parallel performance.

5.1.1 Parallel Independent Runs

One method that has been used prominently in practice to leverage parallel resources is the $k$-independent runs protocol outlined in Hutter et al. [2009b]. This has become a standard protocol for parallelizing algorithm configuration that has been used in a variety of papers (see e.g., Hutter et al. [2007a], Hutter et al. [2010a], Fawcett and Hoos [2013], Hutter et al. [2012], Styles and Hoos [2013]). In this protocol, $k$ independent runs with different seeds are run on parallel computing resources. Once all $k$ runs are completed, the configuration with the best performance on the training set is selected. This approach has several benefits, including that it is easy to schedule the $k$ runs independently in shared cluster environments, and that they need not execute concurrently. This approach’s power comes from exploiting the randomness inherent in the algorithm configurators. Intuitively, one can imagine rolling a set of dice and obtaining the minimum value; the expected minimum value becomes smaller if you roll the roll more dice. This is precisely why this protocol works, and this protocol
yields the largest benefits for high-variance algorithm configurators (like ParamILS), but in general it is applicable to any configurator that is sufficiently randomized.

### 5.1.2 Parallel Automatic Configurator

In this approach, all computational resources are centralized and coordinated by a master thread or process. This master process schedules multiple algorithm runs in parallel and manages their results. Some configurators that leverage this are GGA (Ansotegui et al. [2009]) and D-SMAC (Hutter et al. [2012]). The advantage of this approach is that there is less (if any) redundant computation, and one is better able to leverage parallel resources (e.g., if ten algorithm runs are required for a challenger to beat the incumbent, they can be scheduled in parallel in this approach but not in the parallel independent runs method outlined in Section 5.1.1). One drawback of this approach is that it has a single point of failure. Another drawback is that these runs are typically difficult to schedule on shared cluster systems, as they must reserve a larger portion of the available resources. Additionally, in particular the runtime of algorithm evaluations for runtime optimization tend to exhibit large variation when the objective is runtime minimization, which can require significant complexity to effectively utilize the cores. D-SMAC, for instance, primarily assumes that all algorithm evaluations take the same amount of time.

### 5.1.3 Parallel Dependent Runs

In this protocol and its instantiation with SMAC (called pSMAC), the automatic configurator runs are executed in parallel as in Section 5.1.1. However, over time these parallel runs share data. The data being shared is the data used to build the model, namely the parameter configurations and instances being run and the corresponding measured value of the objective function for each of them. The utility of sharing run data between otherwise independent runs depends greatly on the configurator selected. In the case of ParamILS, it would be of limited utility, because it would only allow for ParamILS to skip runs that had already been completed by another run. Intuitively, one would expect this to happen very rarely, because each ParamILS run would typically explore different parts of the search space, not to mention that when optimizing randomized algorithms, the evaluations themselves would likely have different pseudo-random number generator seeds. However, in SMBO methods, such as SMAC, it is very easy to exploit this run data, by adding it to the data used to build the model. In principle, such an approach could also be leveraged by GGA, by
selectively breeding with configurations from other runs.

Conceptually, this approach has several advantages. The first is that each run has access to an amount of data closer to a fully parallel automatic configurator, albeit with perhaps less quality and some redundancy. In the case of SMAC, a significant portion of the runs being performed are also random evaluations. Another advantage this kind of protocol enjoys is redundancy, in that a failure of a single run or machine does not necessarily ruin the experiment. Indeed, in some of the best runs of pSMAC reported in Section 5.3, some runs ran out of memory, and yet the configuration experiment as a whole still succeeded. These runs can also be easier to schedule on a shared cluster environment because individual configurator runs can appear and disappear at different times.

Of course, there are also some obvious disadvantages to this approach. One is that it is very possible that the dependent runs converge to a single configuration (one can imagine that, e.g., parallel runs of EGO (Jones et al. [1998]) would likely converge to the same point and not maintain enough diversity). Another disadvantage compared to the parallel automatic configurator is that the evaluation of individual configurations is still limited to a single processor.

## 5.2 pSMAC

*pSMAC* is an extension of SMAC that follows the parallel dependent runs paradigm for parallelization. Each run of pSMAC writes its own measured observations of the target algorithm to disk and reads the other runs’ observations. To avoid stress on a shared file system, other runs are only read periodically. The shared runs are written to the model and optionally can be used as a cache of runs. The number of duplicate algorithm executions (and consequently the benefits of using them as a cache) is typically expected to be very low for a number of reasons:

1. Different runs of pSMAC will likely use different pseudo-random number generator seeds, which will be treated differently.

2. Continuous parameters in the configuration space are unlikely to ever be selected in a way that exactly matches the previous value, as the random forest model is piecewise constant.

3. Parameters that have no significant effect on the model are unlikely to be set to the same value.
Consequently, the mechanism through which we would expect pSMAC to benefit is through the ability to build models with more data than would otherwise be available.

5.3 Experiments

5.3.1 Experiment I - Performance Over Time
In this first experiment, we examine pSMAC's performance over time in comparison with SMAC the same scenarios in Section 4.5; in fact, the data for SMAC and experimental setup are the exact same (and so are not repeated here). The numbers in this experiment will differ from the previous scenario, because we are comparing the best of $k$ independent samples against the best of $k$ dependent samples of pSMAC (denoted pSMACx$k$).

5.3.2 Results
Tables 5.1 and 5.2 show the performance of pSMACx4 and pSMACx8 respectively. In general, there are only a few cases in which pSMAC beats SMAC at a statistically significant level. Interestingly, this occurs more frequently for pSMACx8 than pSMACx4. If we validate the performance over time, as shown in Figures 5.1, 5.2, 5.3 for pSMACx4 and Figures 5.4, 5.5, 5.6 we can see some additional patterns. In many scenarios (e.g., QCP-PAR10-SAPS, SWGC-P-q075-PAR1-SAPS, SWGC-P-q095-PAR1-SAPS), there is a significant period of time during which pSMAC performs significantly better, but because of the time budget, eventually the $k$ independent runs of SMAC catch up. In other scenarios, (e.g., SWV-SPEAR-MED), pSMAC offers no apparent benefit over SMAC, but it also does not hurt. In some scenarios pSMAC does seem to perform
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Unit</th>
<th>Default</th>
<th>Best of SMACx4</th>
<th>pSMACx4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Median #</td>
<td>Median #</td>
</tr>
<tr>
<td>IBM-Spear-med</td>
<td>([-10^0 \text{ s}])</td>
<td>2.300</td>
<td>1.593 160</td>
<td>1.455 40</td>
</tr>
<tr>
<td>IBM-Spear-q025</td>
<td>([-10^{-1} \text{ s}])</td>
<td>5.516</td>
<td><strong>3.693</strong> 160</td>
<td>3.704 40</td>
</tr>
<tr>
<td>QCPmed-PAR1-saps</td>
<td>([-10^{-2} \text{ s}])</td>
<td>3.396</td>
<td>2.561 160</td>
<td>2.522 40</td>
</tr>
<tr>
<td>QCPq075-PAR1-saps</td>
<td>([-10^{-1} \text{ s}])</td>
<td>2.500</td>
<td>0.919 160</td>
<td>0.918 40</td>
</tr>
<tr>
<td>QCPq095-PAR1-saps</td>
<td>([10^0 \text{ s}])</td>
<td>3.934</td>
<td>0.463 160</td>
<td><strong>0.437</strong> 40</td>
</tr>
<tr>
<td>SWGCPq075-PAR1-saps</td>
<td>([10^0 \text{ s}])</td>
<td>3.024</td>
<td>0.110 160</td>
<td>0.109 40</td>
</tr>
<tr>
<td>SWGCPq095-PAR1-saps</td>
<td>([10^0 \text{ s}])</td>
<td>2.687</td>
<td>0.123 160</td>
<td>0.121 40</td>
</tr>
<tr>
<td>SWV-Spear-med</td>
<td>([-10^{-1} \text{ s}])</td>
<td>6.631</td>
<td><strong>3.710</strong> 160</td>
<td>3.713 40</td>
</tr>
<tr>
<td>SWV-Spear-q075</td>
<td>([10^0 \text{ s}])</td>
<td>2.644</td>
<td>0.363 160</td>
<td><strong>0.362</strong> 40</td>
</tr>
<tr>
<td>SWV-Spear-q095</td>
<td>([10^0 \text{ s}])</td>
<td>5.000</td>
<td>0.506 160</td>
<td><strong>0.504</strong> 40</td>
</tr>
<tr>
<td>QCP-PAR10-saps</td>
<td>([10^0 \text{ s}])</td>
<td>9.367</td>
<td>3.484 32</td>
<td>3.470 8</td>
</tr>
<tr>
<td>QCP-spear-PAR10</td>
<td>([10^0 \text{ s}])</td>
<td>1.897</td>
<td>0.653 32</td>
<td>0.606 8</td>
</tr>
<tr>
<td>QWHmed-PAR1-saps</td>
<td>([-10^{-2} \text{ s}])</td>
<td>4.212</td>
<td>2.738 32</td>
<td>2.768 8</td>
</tr>
<tr>
<td>SWGCP-PAR10-saps</td>
<td>([10^0 \text{ s}])</td>
<td>15.094</td>
<td>0.107 32</td>
<td>0.104 8</td>
</tr>
<tr>
<td>SWGCP-spear-PAR10</td>
<td>([10^0 \text{ s}])</td>
<td>6.987</td>
<td>6.027 32</td>
<td>5.595 8</td>
</tr>
</tbody>
</table>

Table 5.1: Median training performance of SMAC and pSMAC implementations. For **Best of SMACx4**, we performed bootstrap samplings of 4 runs from the # column, computed the minimum, and then the medium of 1000 samples. For the **pSMACx4** run, we took grouped runs into batches of 4 for sharing runs, and took # independent samples from this. Bold-faced entries represent configurator performance that is significantly better under a Mann-Whitney U Test with significance level of 0.01 (because of the repeated testing we applied a Bonferroni correction and adjusted the individual p-values used for significance of the test to less than \(6.67 \times 10^{-4}\)).

worse for a significant portion of overall running time, but these are the multi-instance scenarios for pSMACx8 which have only 4 runs, although **SWGCP-SPEAR-PAR10** for pSMACx4 also performs worse for a period of time on 8 runs.

### 5.4 Experiment II - CSSC Performance

The Configurable SAT Solver Challenge 2014 (Hutter et al. [2014]) is a competition designed to encourage SAT solvers to parameterize their algorithms so that meta-algorithmic techniques can be applied to improve performance on an instance type of
interest. In this experiment, we compare the original CSSC 2014 data with a run of pSMAC for each configuration scenario run on the same hardware.

5.4.1 Experimental Setup

In the original competition, a number of instance sets were created comprising SAT problems that were grouped in four categories:

- Industrial: (BMC, CircuitFuzz, IBM)
- Crafted (GI, LABS, Queens)
Figure 5.2: Median Performance and Interquartile range for SPEAR Single Instance Scenarios using 4 cores

- Random SAT+UNSAT (3cnf, K3, unif-k5)
- Random SAT (3sat1k, 5sat500, and 7sat90)

Each competing solver was run on the instance sets to determine the default performance and then was configured using the configurators GGA (Ansotegui et al. [2009]), ParamILS (Hutter et al. [2007b]), and SMAC (Hutter et al. [2011b]). For ParamILS, we performed four independent algorithm configuration runs on a discretized version of the parameter space. SMAC was used with four independent runs of the continuous configuration space, and four independent runs for a discretized version
Figure 5.3: Median Performance and Interquartile range for Multi-Instance Scenarios using 4 cores

of the configuration space. GGA was run once with 4 cores each on the discretized version of the space and the continuous version. In the CSSC, the best solver was determined as the one that solved the most instances, with ties broken by the solver with the least runtime. Each algorithm configurator run was given the minimum of 172,800 CPU seconds or 172,800 wall-clock seconds to run. Each run of the target algorithm was given a limit of 300 CPU seconds. After the configuration process has been completed we perform validation on the same problem instances and seeds that were used in the CSSC competition.

Using the original competition data from the CSSC competition and running on the same hardware (see Section 4.2), we compared the original configuration runs with two runs of pSMAC (one on the discretized and the other on the continuous version of the space). Hutter et al. [2014] reports the number of timeouts for each solver and their PAR1 score (i.e., the sample mean performance treating timeouts as the maximum value), the individual configurators were optimizing the PAR10 score (i.e., the sample mean performance treating timeouts as 10 times the maximum value). Since we want to summarize performance in a single value, we also use PAR10 to compare all configurators.
### Table 5.2: Median training performance of SMAC and pSMAC implementations.

For **Best of SMACx8**, we performed bootstrap sampling of 8 runs from the `#` column, computed the minimum, and then the medium of 1000 samples. For the **pSMACx8** run, we took grouped runs into batches of 8 for sharing runs, and took `#` independent samples from this. Bold-faced entries represent configurator performance that is significantly better under a Mann-Whitney U Test with significance level of 0.01 (because of the repeated testing we applied a Bonferroni correction and adjusted the individual p-values used for significance of the test to less than $6.667 \times 10^{-4}$).

#### 5.4.2 Results

Altogether there are 72 scenarios, on only 66 of which any configurator makes any progress. In summary, GGA is the best configurator in 4 scenarios, ParamILS in 15, SMAC in 24 and pSMAC 23.

In Figure 5.7, we plot the difference in performance between a configurator’s best run on the training set versus the best run of all other configurators. Therefore, for example, if the PAR10 scores of pSMAC, SMAC, ParamILS and GGA were (1, 5, 10, 20) respectively, we would report (+4, -4, -9, -19) for the 4 configurators respectively. The PAR10 score of an incomplete or missing run is defined to be 3000. In a number of scenarios, the difference in performance is small, but in others it is clear that some configurators perform significantly better than others.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Unit</th>
<th>Default</th>
<th>Best of SMACx8</th>
<th>pSMACx8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Median</td>
<td>#</td>
</tr>
<tr>
<td>IBM-Spear-med</td>
<td>$\cdot10^0$ s</td>
<td>2.300</td>
<td>1.593</td>
<td>160</td>
</tr>
<tr>
<td>IBM-Spear-q025</td>
<td>$\cdot10^{-1}$ s</td>
<td>5.516</td>
<td><strong>3.693</strong></td>
<td>160</td>
</tr>
<tr>
<td>QCPmed-PAR1-saps</td>
<td>$\cdot10^{-2}$ s</td>
<td>3.396</td>
<td>2.561</td>
<td>160</td>
</tr>
<tr>
<td>QCPq075-PAR1-saps</td>
<td>$\cdot10^{-1}$ s</td>
<td>2.500</td>
<td>0.919</td>
<td>160</td>
</tr>
<tr>
<td>QCPq095-PAR1-saps</td>
<td>$\cdot10^0$ s</td>
<td>3.934</td>
<td>0.463</td>
<td>160</td>
</tr>
<tr>
<td>SWGCPq075-PAR1-saps</td>
<td>$\cdot10^0$ s</td>
<td>3.024</td>
<td>0.110</td>
<td>160</td>
</tr>
<tr>
<td>SWGCPq095-PAR1-saps</td>
<td>$\cdot10^0$ s</td>
<td>2.687</td>
<td>0.123</td>
<td>160</td>
</tr>
<tr>
<td>SWV-Spear-med</td>
<td>$\cdot10^{-1}$ s</td>
<td>6.631</td>
<td><strong>3.710</strong></td>
<td>160</td>
</tr>
<tr>
<td>SWV-Spear-q075</td>
<td>$\cdot10^0$ s</td>
<td>2.644</td>
<td>0.363</td>
<td>160</td>
</tr>
<tr>
<td>SWV-Spear-q095</td>
<td>$\cdot10^0$ s</td>
<td>5.000</td>
<td>0.506</td>
<td>160</td>
</tr>
<tr>
<td>QCP-PAR10-saps</td>
<td>$\cdot10^0$ s</td>
<td>9.367</td>
<td>3.484</td>
<td>32</td>
</tr>
<tr>
<td>QCP-spear-PAR10</td>
<td>$\cdot10^0$ s</td>
<td>1.897</td>
<td>0.653</td>
<td>32</td>
</tr>
<tr>
<td>QWHmed-PAR1-saps</td>
<td>$\cdot10^{-2}$ s</td>
<td>4.212</td>
<td>2.738</td>
<td>32</td>
</tr>
<tr>
<td>SWGCP-PAR10-saps</td>
<td>$\cdot10^0$ s</td>
<td>15.094</td>
<td>0.107</td>
<td>32</td>
</tr>
<tr>
<td>SWGCP-spear-PAR10</td>
<td>$\cdot10^0$ s</td>
<td>6.987</td>
<td>6.027</td>
<td>32</td>
</tr>
</tbody>
</table>
of cases, GGA had problems executing on a scenario. Thus, to prevent the graph from being overwhelmed by GGA’s performance it has been truncated to only show a loss of -150. Finally, we only plot the 66 scenarios where at least one configurator found an improving configuration.

From Figure 5.7 we can observe the following. First, while there is seemingly a large amount of variance among configurators, pSMAC does seem to allow for the exploration of previously unseen regions of the configuration space. Second, it performed well in similar scenarios as SMAC: when it performed best, the next best configurator after pSMAC was often SMAC. Areas where pSMAC did poorly tended to
be dominated by SMAC, and the performance loss of both SMAC variants was never as severe as that of other configurators. Only for two of 66 scenarios, did pSMAC lose by a PAR10 of more than 50.

In Figure 5.8, we show the variation between individual runs of a configurator. In this graph, we plot, in increasing order for each scenario, how much variance there is between the individual runs’ PAR10 score. In the case of ParamILS, it is the sample standard deviation across 4 runs, and in the case of (p)SMAC it is the sample standard deviation across 8 runs (4 on the normal configuration space and 4 on the discretized configuration space).

Figure 5.5: Median Performance and Interquartile range for SPEAR Single Instance Scenarios using 8 cores
Figure 5.6: Median Performance and Interquartile range for Multi-Instance Scenarios using 8 cores

Figure 5.7: Configurator Performance in CSSC 2014 Competition Data
Figure 5.8: Configurator Variance. In the plot the final point where SMAC’s variance sky-rockets is due to some missing data for SMAC.

The figure suggests that pSMAC has lower variance in its configurator runs than both SMAC and ParamILS. In conjunction with what we observed from Figure 5.7, this suggests that pSMAC is both a very competitive configurator and a relatively stable configurator.

### 5.5 Future Extensions

A fundamental limitation of pSMAC, as opposed to a parallel automatic configurator, is that individual pSMAC runs can only explore the configuration space a single evaluation at a time. In the later stages of a challenge, when we have obtained many measurements of the incumbents performance, it is possible that a promising configuration might need more than a hundred evaluations, each of which must be done sequentially.

It would not be difficult in practice to fix this behaviour, as it would only require two changes. The first is that it would require the SMAC algorithm (shown in Algorithm 4.1) to request runs be evaluated in parallel and to use dynamic adaptive capping (as
The second is that we would replace the local execution mechanism with the distributed mechanism (described in Section 3.5), but at the same time, launch a worker locally in parallel. Some synchronization would be needed between the local worker and pSMAC to ensure that the worker is not evaluating a run while pSMAC is building a model. Consequently, both could be launched within the same process, and thread synchronization could be used to coordinate between them. When pSMAC submits a run to the database, the worker would start processing jobs from the database. When pSMAC is notified that the run it requested is completed, it would wait for the local worker to finish its evaluation, and then control would return to the pSMAC algorithm. Figure 5.9 illustrates this approach.
Bibliography


configuration. In Proc. of LION-6, pages 55–70.


Hutter, F. and Ramage, S. (2013). Manual for SMAC version v2.06.01-master. Department of Computer Science, UBC.


combined selection and hyperparameter optimization of classification algorithms.

In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 847–855.

Appendix A

AEATK Decorators

In this appendix, we outline all of the decorators supplied with AEATK. For each decorator we outline the class name and provide a description. One word of caution is that the names of the decorators have not been maintained nor has a consistent naming scheme been applied, for consistency in the thesis we omit the words TargetAlgorithmEvaluator and Decorator from the name, as the context is implicit. Most of the decorators listed below are available to any AEATK based application on the command-line, and many are enabled by default, a few rare ones are only available to meta-algorithm designers via the API for certain special use cases. The distinction between the type of decorator is at times blurry, never the less these groupings are the groupings that exist in the code (at the Java package level). Finally most decorators were created as a result of a real problem, and were added to the official release in case they proved useful later, nevertheless some of these decorators are admittedly very esoteric or bizarre.

A.1 Debug Decorators

These decorators are useful for debugging meta-algorithms or TargetAlgorithmEvaluator implementations.

CheckForDuplicateRunConfig This decorator checks to ensure that the meta-algorithm does not supply duplicate AlgorithmRunConfiguration objects within a single request, if so an exception is thrown. The motivation for this decorator is that as per the conceptual model outlined in Section 3.1.3, there is no reason to ask for the same point to be evaluated twice within a single request. Other decorators and TargetAlgorithmEvaluator may use the
AlgorithmRunConfiguration as a key in a Map and may not handle the case
where a request has duplicates. Instead of throwing an exception, this decorator
could silently fix the request but the original impetus was a bug in a meta-
algorithm and there seems to be no good reason for a meta-algorithm to ever
make such a request.

**EqualTester** This decorator wraps two TargetAlgorithmEvaluator implementa-
tions and ensures that they both answer the same value.

**LeakingMemory** This decorator leaks a configurable amount of memory for every
request by allocating an array.

**LogEveryObservation** This decorator logs a message with the results of every
notification of an observer.

**LogEvery** This decorator logs a message for every AlgorithmRunConfiguration
submitted and AlgorithmRunResult returned.

**RunHashCodeVerifying** This decorator keeps track of a hash for all runs completed
thus far, it is useful if one wants to verify that the runs being executed matches
a predefined sequence.

**SanityChecking** This decorator ensures that the AlgorithmRunResult objects being
returned and observed correspond to the AlgorithmRunConfiguration objects
being submitted.

**UncleanShutdownDetecting** The TargetAlgorithmEvaluator is a resource like
a file that must be closed when completed by calling the notifyShutdown() method\(^1\). Closing a TargetAlgorithmEvaluator allows it to do necessary
shutdown tasks before the program exits, such as writing everything to the
disk/database, or clean up in other decorators. This decorator uses a shutdown
hook, to detect when Java is exiting without the TargetAlgorithmEvaluator
being told as this is commonly a bug and has some unfortunate side effects
(e.g., if any thread pools are still running, after the main() method returns, the
program will hang and never exit).

\(^1\)When the code was ported to Java 7, TargetAlgorithmEvaluator was changed to extend
AutoCloseable and the close() method was created as an alias for notifyShutdown()
A.2 Functionality Decorators

These decorators provide some functionality to the TargetAlgorithmEvaluator.

**PrePostCommand** This decorator schedules shell commands either before the meta-algorithm starts, or when it completes (technically when it calls notifyShutdown()).

**Portfolio** This decorator allows users to treat a series of AlgorithmExecutionConfiguration objects as one as far as the meta-algorithm is concerned. Where this is useful for instance is in trying to configure a portfolio of configurations for an algorithm, if you want your objective being optimized to be the minimum of a new candidate configuration, and the objective of the current configurations in a portfolio, this decorator allows you to do that.

**OutstandingEvaluations** This decorator supplies the implementation of the getNumberOfOutstandingBatches(), getNumberOfOutstandingRuns(), and waitForOutstandingEvaluations(). Many decorators internally alter the number of runs being executed at any time, in a way that should be transparent to the calling code. This decorator typically applied near top of the decorator chain allows these methods to behave properly even if internally the behaviour is different. For example, if a decorator is retrying all CRASHED runs, then there is a race condition between when the run is completed and when it is resubmitted, where the client could potentially erroneously see zero runs outstanding.

**OutstandingEvaluationWithAccessor** This decorator is a subtype of the above that provides additional methods that additionally provides methods to access the set of outstanding runs. It is used primarily for debugging. A subtype is used because this decorator must keep track of the actual runs, where as the previous one can simply use counters.

**SimulatedDelay** This decorator slows down the returning of AlgorithmRunResult to be no faster than the reported runtime. This is primarily used for debugging when one wants to use a test function as an algorithm but the test function itself evaluates instantly.

**TerminateAllRunsOnFileDelete** This decorator silently invokes the kill() method on all outstanding runs if a file on disk is deleted. It is through
this mechanism that Auto-WEKA can terminate SMAC near instantaneously, as otherwise it would have to wait for SMAC to finish executing the target algorithm.

**Transform** This decorator allows the user to transform the returned `AlgorithmRunResult` according to some user-defined function. For instance if you wanted to treat instances that were SAT differently than UNSAT as far as optimization is concerned.

### A.3 Helper Decorators

These decorators provide some helpful functionality.

**CallObserverBeforeCompletion** This decorator notifies the observer of the final result before returning. This essentially simplifies implementing the `TargetAlgorithmEvaluator` interface, by requiring no semantics for observation, yet allowing the meta-algorithm to have the useful semantic of always being able to observe the final result. Without this decorator certain work-flows can be annoying, for instance imagine a user interface which wants to display live results of runs. Without this decorator users would have to use two code paths to update the UI, the first being the observer, the second is when the runs are complete. This decorator allows the client code to be simplified by guaranteeing that the observer always sees the final result.

**KillCaptimeExceedingRuns** This decorator invokes the `kill()` method automatically if the observed runtime exceeds some ratio, for instance 10 times the requested cutoff time. This is useful because occasionally misbehaving wrappers can otherwise cause a meta-algorithm to become stuck.

**OutstandingRunLogging** This decorator tracks the time all runs are submitted to the `TargetAlgorithmEvaluator` and the time they are returned, as well as the measured runtime and walltime. From this data, one can plot the approximate utilization of the meta-algorithm over time. The data is computed by tracking the time when a run was first determined to be completed (i.e., either when the request completes, or we first observe it completed) and from that subtracting the reported runtime and walltime respectively to determine when it started (i.e., this approximation puts all of the wall-clock overhead at the start of the run). For example, in the experiment in Section 3.5.2, we can examine how well
the MySQLTargetAlgorithmEvaluator was utilized with a batch size of 32 and a cutoff of 4 second which is shown in Figure A.1. While the plot is noisy one can notice for instance that over time the batches are completed in less time as the workers adjusted, the first batch takes almost 12 seconds, well the last takes less than 10. The utilization is computed in buckets and is approximate, the spike above 16 is due to the polling frequency (e.g., if a worker does two runs between polls then both runs will appear to take place at the same time).

**RetryCrashedRuns** This decorator resubmits runs that report CRASHED a configurable number of times.
**StrictlyIncreasingRuntimes** This decorator\(^2\) ensures that the meta-algorithm only observes an increasing value for the runtime of the algorithm. This is necessary to guard against some race conditions where a run is restarted (e.g., a worker dies and it is reassigned), while simultaneously the meta-algorithm requests the run be terminated and then unexpectedly the meta-algorithm receives a run with a much lower runtime than it’s invariants expect (e.g., in our improved VBS in Section 3.1.4 would want KILLED runs to have a higher runtime than the best completed run).

**NotifyTerminationCondition** This decorator notifies an object called a TerminationCondition about runs being completed. These objects are used in meta-algorithms to track how much CPU time has been used, and terminate the meta-algorithm when a limit has been reached.

**UseDynamicCappingExclusively** This decorator transforms all AlgorithmRunConfiguration objects to have the same cutoff time as the AlgorithmExecutionConfiguration object, and then supplies an observer that terminates the run at the requested cutoff time. This is used to improve cache hit rates as two runs that differ only in their cutoff time can be treated as the same run.

**WalltimeAsRuntime** This decorator causes the meta-algorithm to see a scaled down version of the wall-time as the runtime of the run when being observed. This is useful if the target algorithm does not support observation of runtime (e.g., if runsolver isn’t being used as discussed in Section 3.4), and only kicks in after 5 seconds by default.

### A.4 Resource Decorators

These decorators help manage computation resources within the TargetAlgorithmEvaluator.

**Bounded** This decorator limits the number of AlgorithmRunConfiguration objects that are allowed to be submitted to the decorated TargetAlgorithmEvaluator, and blocks the caller of evaluateRunAsync() until all runs have been submitted. This decorator is useful in a number of situations, the first is to

\(^2\)Which probably should be called StrictlyNonDecreasingRuntimes.
allow TargetAlgorithmEvaluator implementations to easily limit the number of target algorithm evaluations in progress at any time. For instance, the MySQLTargetAlgorithmEvaluator’s evaluateRunAsync() method never blocks, because runs can always be written to the database, so if a meta-algorithm wants to limit its progress based on the number of available computational resources then this decorator provides the required functionality. By this decorators’ nature, it inherently supports observation and termination of target algorithm evaluations, because it sees completed runs come back in pieces, and can chose not to submit runs that the user invoked kill() on. So, even if a TargetAlgorithmEvaluator does not support these features, using this decorator can be a pretty good approximation performance wise if many runs are being submitted.

Caching  This decorator uses an in-memory cache for caching duplicate runs submitted in different requests.

FileCache  This decorator reads and writes all runs to a directory and uses those runs as a cache. Primarily it is used for debugging as it allows developers of meta-algorithms to skip redoing the same results in subsequent runs of their meta-algorithm.

Forking  This decorator, based on a configurable policy, may chose to submit the AlgorithmRunConfiguration to a different TargetAlgorithmEvaluator. For instance, to speed up processing of short target algorithms with the MySQLTargetAlgorithmEvaluator, the this decorator may simultaneously submit the runs to a CommandLineTargetAlgorithmEvaluator but only running each for 1 second.

NonBlocking  This decorator allows every call to evaluateRunAsync() return immediately. It complements the Bounded decorator above. This decorator is used when a component of your meta-algorithm must always be able to submit it’s runs in a timely fashion, for instance, to avoid a deadlock. This functionality is implemented by putting all submitted runs into a queue, and having another thread actually invoke evaluateRunAsync() on the decorated TargetAlgorithmEvaluator.

Preempting  This decorator exposes a method that provides another low-priority TargetAlgorithmEvaluator, but AlgorithmRunConfiguration objects submitted to it will be terminated and restarted if other runs are submitted and block.
This low-priority support is very heuristic as the `TargetAlgorithmEvaluator` API does not natively expose this kind of information. It is based on observation, if the `TargetAlgorithmEvaluator` reports that low priority runs have runtimes greater than zero, and yet the normal runs are zero, then we kill low priority runs to make room.

**RunHistoryCaching** This decorator uses another AEATK object called a `RunHistory` object as a cache of runs. In pSMAC (see Section 5.2), the `RunHistory` object is populated with run data from other runs, and this decorator allows those runs to be added to the cache. It differs from the `FileCache` decorator, in that the `FileCache` decorator is responsible for managing the cache, where in this one the `RunHistory` object is.

### A.5 Safety Decorators

These decorators are exist to ensure safety and correctness.

**AbortOnCrash** This decorator treats `CRASHED` runs as `ABORT` runs and is configurable by the user.

**AbortOnFirstRunCrash** This decorator checks if the very first run is a `CRASHED` run and if so changes it to an `ABORT`. The motivation for this decorator is two-fold, first meta-algorithms like SMAC behave poorly if the first run is a `CRASHED` because it is uninformative. The second is that most likely if the first run is a `CRASHED`, every run is likely to be a `CRASHED`.

**CrashedSolutionQualityTransforming** This decorator makes sure that the reported solution quality of `CRASHED` runs is greater than some value. It guards against wrappers which default the solution quality to 0, which the meta-algorithm will treat as very good.

**ExitOnFailure** This decorator forcibly exits the JVM via a combination of `exit()` and `halt()` if an exception is encountered evaluating the `AlgorithmRunRequest`. The best explanation is the use-case it was designed for, which was a distributed meta-algorithm using the `MySQLTargetAlgorithmEvaluator` where the worker `TargetAlgorithmEvaluator` was not the `CommandLineTargetAlgorithmEvaluator` but a custom one that utilized JNI to talk to a C target algorithm directly. This target algorithm occasionally
had non-deterministic problems with memory corruption and to make the meta-algorithm more reliable it was decided that the worker process should just exit abruptly, and be restarted.

**JVMShutdownBlocker** This decorator *detaches* submitted and observed `AlgorithmRunConfiguration` objects from their caller when the JVM is shutting down. It also silently drops requests that are submitted after the shutdown has started. The primary purpose of this decorator is to avoid generating spurious or erroneous results to the meta-algorithm that are due to shutdown. For instance if the meta-algorithm was terminated by the user via `CTRL+C`, the entire process tree will start shutting down. This often means that wrappers will report `CRASHED`. Depending on how long it takes the JVM to shutdown, the meta-algorithm may see this and start logging a number of warnings or errors that are misleading. Consequently when the JVM detects shutdown, this decorator basically stops notifying the caller of updates, and stops submitting new runs.

**SATConsistency** This decorator does an in-memory check that for each problem instance the satisfiability (i.e., whether the run is SAT or UNSAT) is consistent, as per the conceptual model in Section 3.1.3.

**SynchronousObserver** This decorator simply invokes the user-supplied observer in a synchronized method. This prevents the observer from being invoked simultaneously, and ensures that there are no memory visibility issues if it is called by multiple threads, and guards against the observer being notified by two threads concurrently. This protection isn’t perfect however, as if the same observer is supplied with multiple sets of runs the user must still provide additional thread safety guarantees.

**TimingChecker** This decorator logs warnings if the target algorithm wall-time is substantially higher than the measured runtime, suggesting that there is substantial unaccounted for overhead that the user may be unaware of.

**VerifySAT** This decorator uses the instance specific information on the `ProblemInstance` object to verify whether the target algorithm correctly detected satisfiability of the instance.

**WarnOnNoWallOrRuntime** This decorator logs a warning if none of the runs submitted have a runtime or walltime greater than 0. The most common
use case where this occurs is when the workers are not started with the MySQLTargetAlgorithmEvaluator.