Automating Meta-algorithmic Analysis and Design

by

Christopher Warren Nell

B. Sc., University of Guelph, 2005

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

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

October 2011

© Christopher Warren Nell, 2011

Abstract

Sophisticated empirical methods based upon automated experimental analysis techniques drive the development of high-performance solvers for an increasing range of problems from industry and academia. However, tools implementing these methods are often difficult to develop and to use. This thesis describes work towards addressing this issue. First, we develop a formal description of *meta-algorithmic problems*, and use it as the basis for a framework supporting the development and application of a broad class of automated algorithm analysis and design techniques. Second, we describe the High-performance Algorithm Laboratory (HAL), an extensible software implementation of this framework that provides developers and users of automated empirical techniques with support for distributed execution, remote monitoring, data management, and statistical analysis of results. We use HAL to construct a suite of interoperable tools that implement a variety of automated empirical techniques, and demonstrate our approach by conducting a sequence of increasingly complex analysis and design tasks on state-of-the-art solvers for Boolean satisfiability and mixed-integer programming problems.

Preface

HAL, the main contribution of this thesis, represents the development and eventual realization of an idea proposed by Holger Hoos [33]. As the principal investigator of the HAL project, I performed the majority of design, development, analysis, and expository work presented herein; in particular:

- I designed and developed the core HAL framework, including all APIs and implementations for all experiment modelling, data management, and execution management components used in this thesis; discussed in Chapter 4 and demonstrated in Chapters 5 and 6.
- I designed and implemented all HAL analysis procedures discussed in Chapter 3 and used in Chapters 5 and 6. I also developed the *ZILLA and HYDRA-* plugins used in Chapter 6, and contributed to the implementation of the GGA and PARAMILS plugins used in Chapter 5.
- I conducted and analysed all computational experiments presented in Chapters 5 and 6.
- I am the primary author of all documentation of the HAL framework, including this thesis and its appendices, the work originally presented at LION-5 [63] on which Chapters 1–5 of this thesis are based, and all supporting documentation associated with the HAL software distribution available online at hal.cs.ubc.ca.

As HAL is intended as a foundation for ongoing, collaborative research efforts, I have also worked with several excellent co-authors and collaborators throughout my research. This joint work has contributed to many aspects of HAL:

- As a core contributor to the HAL project and co-author of our LION-5 paper, Chris Fawcett led design and implementation of the HAL user interface, made significant contributions to the GGA and PARAMILS plugins for algorithm configuration used in the case study, and assisted in conducting preliminary experiments. Since then, he has continued to improve the HAL user interface, and has taken an enhanced role in the development of data management components for HAL. Chris is also an active end-user of HAL (see, e.g., [21, 79]), and has provided valuable insight throughout the framework's design and development.

- Chapter 4 includes discussion of a TORQUE execution manager that was implemented by James Styles based on the Grid Engine execution manager I implemented. James continues to be involved in the design and development of execution management components for HAL.
- Chapters 5 and 6 make use of the ROAR plugin for algorithm configuration in HAL that was designed and implemented by Frank Hutter [44], in addition to the PARAMILS algorithm configuration plugin that is based on his earlier work [39, 40]. Frank continues to lead a group of researchers developing new meta-algorithmic procedures with HAL.
- Chapter 6 describes the design and implementation of the ∗ZILLA plugin for portfolio-based algorithm selection and the HYDRA-∗ plugin for per-instance configuration. Both of these plugins are based on previous implementations by Lin Xu *et al.* [84], and I consulted Lin regularly when adapting them for the HAL framework and analysing their performance.
- All work presented in this thesis was performed under the supervision of Holger Hoos and Kevin Leyton-Brown.

In the remainder of this thesis, I adopt the first person plural in recognition of these collaborators.

Finally, HAL and its plugins have been implemented using a variety of open-source libraries and tools, most significantly including components from Oracle (oracle.com), Hyperic (hyperic.com), The MathWorks (mathworks.com), Google (google.com), the Eclipse Foundation (eclipse.org), and the Apache Foundation (apache.org). Specific components and their authorship and licensing details are listed more exhaustively in the HAL software distribution.

Table of Contents

Abstract . . . ii

Preface . . . iii

Table of Contents . . . v

List of Tables . . . ix

List of Figures . . . x

Acknowledgments . . . xi

1 Introduction . . . 1
- 1.1 Contributions . . . 3
- 1.2 Thesis Organization . . . 4

2 Background and Related Work . . . 5
- 2.1 Algorithm Analysis . . . 6
  - 2.1.1 Empirical Analysis Techniques . . . 6
  - 2.1.2 Analysis Tools . . . 8
- 2.2 Algorithm Design . . . 9
  - 2.2.1 Direct Design Techniques . . . 9
  - 2.2.2 Meta-algorithmic Design Techniques . . . 11

3 Meta-algorithmic Problems . . . 14
- 3.1 Fundamentals . . . 14
- 3.2 Meta-algorithmic Problems . . . 18
  - 3.2.1 Meta-algorithmic Analysis Problems . . . 18
    - 3.2.1.1 Single-Algorithm Analysis . . . 19
    - 3.2.1.2 Pairwise Comparison . . . 21

3.2.1.3 *k*-way Comparison . . . 22
3.2.2 Meta-algorithmic Design Problems . . . 24
3.2.2.1 Algorithm Configuration . . . 24
3.2.2.2 Per-instance Portfolio-Based Selection . . . 27
3.2.2.3 Per-instance Configuration . . . 29

4 The High-Performance Algorithm Laboratory . . . 31
4.1 Design Considerations . . . 31
4.1.1 Components for Experiment Modelling and Meta-algorithmic Logic . . . 33
4.1.2 Execution and Data Management Infrastructure . . . 34
4.2 The HAL 1.1 Software Environment . . . 36
4.2.1 Experiment Modelling Subsystem . . . 37
4.2.2 Execution and Data Management Subsystem . . . 38
4.2.3 User Interface Subsystem . . . 40

5 Algorithm Analysis & Design with HAL . . . 41
5.1 Experimental Setup . . . 41
5.1.1 Analysis Procedures . . . 41
5.1.1.1 Single-Algorithm Analysis Procedure: SCD-Based Analysis . . . 42
5.1.1.2 Pairwise Comparison Procedure: Comprehensive Pairwise Comparison . . . 42
5.1.1.3 *k*-way Comparison Procedure: Comprehensive *k*-way Comparison. 42
5.1.2 Algorithm Configuration Procedures . . . 43
5.1.2.1 Algorithm Configuration Procedure: PARAMILS . . . 43
5.1.2.2 Algorithm Configuration Procedure: GGA . . . 43
5.1.2.3 Algorithm Configuration Procedure: ROAR . . . 44
5.2 Case Study: Choosing a MIP Solver . . . 44
5.2.1 Single-Algorithm Analysis . . . 44
5.2.2 Pairwise Comparison . . . 45
5.3 Case Study: Adapting a SAT Solver . . . 46
5.3.1 Single-Algorithm Analysis . . . 46
5.3.2 Pairwise Comparison . . . 47
5.3.3 Algorithm Configuration . . . 48

6 Developing Meta-algorithms with HAL . . . 51
6.1 Per-instance Portfolio-Based Selection with ∗ZILLA . . . 51
6.1.1 The ∗ZILLA Selector Algorithm . . . 53

6.1.2 The ∗ZILLA Design Procedure 55
6.2 Per-instance Configuration with HYDRA-∗ 58
6.2.1 The HYDRA-∗ Design Procedure 58
6.2.2 Portfolio Candidate Filtering 60
6.3 Performance Validation 61
6.3.1 Experimental Setup 63
6.3.2 Results 65

7 Conclusions & Future Work 70

Bibliography 72

A HAL 1.1 User Reference 79
A.1 Installation 79
A.2 Running Experiments 80
A.2.1 Instance Sets 80
A.2.2 Algorithms 81
A.3 Analysis Procedures 82
A.3.1 SCD-Based Analysis 83
A.3.2 Comprehensive Pairwise Comparison 83
A.3.3 Comprehensive *k*-way Comparison 83
A.4 Design Procedures 84
A.4.1 PARAMILS 84
A.4.2 GGA 84
A.4.3 ROAR 85
A.4.4 ∗ZILLA 85
A.4.5 HYDRA-∗ 85

B HAL 1.1 Developer Reference 98
B.1 Tutorial: Developing Meta-algorithms with HAL 98
B.1.1 Implementation Basics 99
B.1.1.1 The *MetaAlgorithmImplementation* constructor 101
B.1.1.2 *MetaAlgorithmImplementation* methods 103
B.1.1.3 The *MetaAlgorithmRun* constructor 105
B.1.1.4 *MetaAlgorithmRun* methods 106
B.1.2 Testing 108
B.1.2.1 Target *Algorithms* 108

B.1.2.2 Target *ProblemInstances* 110
B.1.2.3 *Environments* 111
B.1.2.4 Performance Metric 111
B.1.2.5 Unit test 111
B.1.3 Additional Outputs and Improvements 113
B.1.3.1 Improving Parallelism with *AlgorithmRunVisitors* 113
B.1.3.2 *Plots* and *Statistics* 115
B.1.3.3 Testing 117
B.1.4 Plugin Distribution 117
B.2 Execution Management 118
B.2.1 HAL 1.1 Execution Pipeline 119
B.2.1.1 Algorithm Transformations 121
B.2.1.2 Limitations 121
B.2.1.3 *ExecutionManager* Implementations 122
B.2.2 *LocalExecutionManagers* 122
B.2.3 *SSHExecutionManagers* 123
B.2.4 *SGE*- and *TorqueClusterExecutionManagers* 125
B.3 Data Management 126
B.3.1 Data Manager Interfaces 127
B.3.1.1 *ReadOnlyDataManagers* 128
B.3.2 Experimental Data Management 128
B.3.2.1 *DecoratedExecutionManagers* 129
B.3.2.2 Run Filters 129
B.3.3 SQL Data Managers 130
B.3.3.1 Database Schema 131
B.3.3.2 Query Handlers and Beans 131
B.3.3.3 *SQLRunFilters* 131

List of Tables

Table 5.1 Summary of case study results 44

Table 6.1 Performance of ∗ZILLA and HYDRA-∗ designs 66

Table A.1 SCD-Based Analysis inputs/outputs 87
Table A.2 Comprehensive Pairwise Comparison inputs/outputs 88
Table A.3 Comprehensive *k*-way Comparison inputs/outputs 90
Table A.4 ParamILS inputs/outputs 92
Table A.5 GGA inputs/outputs 93
Table A.6 ROAR inputs/outputs 94
Table A.7 ∗ZILLA inputs/outputs 95
Table A.8 HYDRA-∗ inputs/outputs 97

Table B.1 Input/output semantics 102
Table B.2 Performance metrics 112

List of Figures

Figure 4.1 Architectural overview of HAL 37

Figure 5.1 Single-algorithm analyses of CPLEX and SPEAR 45
Figure 5.2 Pairwise comparison of CPLEX and Gurobi on MILP benchmark set 46
Figure 5.3 Pairwise comparisons of SPEAR designs on SWV test set 47
Figure 5.4 Comparison of PARAMILS, GGA, and ROAR designs on SWV test set 50

Figure 6.1 Pseudocode: ∗ZILLA selectors 54
Figure 6.2 Pseudocode: ∗ZILLA design procedure 56
Figure 6.3 Pseudocode: HYDRA-∗ design procedure 59
Figure 6.4 Pseudocode: Independent-validation candidate filter 62
Figure 6.5 Pseudocode: Reused-run candidate filter 62
Figure 6.6 Pseudocode: Reuse-maximizing candidate filter 63
Figure 6.7 Performance evolution of HYDRA designs 67
Figure 6.8 Effect of overhead costs on ∗ZILLA selector algorithm performance 68
Figure 6.9 Performance correlation between HYDRA-∗ designs using ROAR vs. PARAMILS 69

Figure B.1 SCDs of test algorithms vs. seed 118
Figure B.2 Class diagram: *AlgorithmImplementations* and *AlgorithmRuns* 134
Figure B.3 Class diagram: *ParameterSpaces* and *ParameterSettings* 135
Figure B.4 Class diagram: *Domains* 136
Figure B.5 Class diagram: *Environments* 137
Figure B.6 Class diagram: *Algorithms* 138
Figure B.7 Class diagram: *ProblemInstances* 138
Figure B.8 SQL schema 139

Acknowledgments

I would first like to express my sincere thanks to Holger Hoos and Kevin Leyton-Brown, my supervisors. Our many meetings, discussing issues ranging from abstract meta-algorithmic concepts to specific user interface details, have helped me develop both research and project management skills. I have no doubt that your feedback has made me a much better scientist than I was when I arrived at UBC.

I also thank Chris Fawcett, co-author of our LION-5 paper, for the countless hours of work he put into HAL, both before publication and in proactive use and testing afterwards. I would similarly like to thank Frank Hutter for implementing ROAR and testing HAL, Lin Xu for assistance in implementing and validating ∗ZILLA and HYDRA-∗, and all of the members of EARG for the various ways you have (directly and indirectly) contributed to my work.

I would like to acknowledge NSERC for awarding me a CGS M fellowship, which has allowed me to devote more time and energy to this research than would otherwise have been possible.

Finally, to my best friend, my partner, my wife, my Mai – without your unwavering support, this work would have been impossible.

Chapter 1

Introduction

Empirical techniques play a crucial role in the design, study, and application of high-performance algorithms for computationally challenging problems. Indeed, state-of-the-art solvers for prominent combinatorial problems, such as propositional satisfiability (SAT) and mixed integer programming (MIP), rely heavily on heuristic mechanisms that have been developed and calibrated based on extensive computational experimentation (see, e.g., [38, 33, 41]). Performance assessments of such solvers are also based on empirical techniques, as are comparative analyses of competing solvers (see, e.g., [15, 13, 35]).

Empirical methods tend to be used in an ad-hoc fashion, relying upon informal experimentation rather than established best practices. For example, high-performance heuristic solvers are often constructed iteratively, with a designer alternately modifying an existing algorithm and empirically evaluating the algorithm on a set of benchmark problem instances. Designing an algorithm in this way requires considerable expertise, as the number of possible modifications at each step is typically large, and the cost of implementation and evaluation is significant. In order to control development costs, even experts consider only a tiny subset of all possible designs, and often severely restrict the scope of intermediate empirical analyses. These restrictions can lead to unnecessarily complicated algorithms that do not make optimal use of their heuristic components, and whose performance generalizes poorly beyond the limited set of benchmark instances used during development.

Leveraging recent increases in the availability of cheap computational resources, a variety of techniques have been introduced to overcome these limitations in manual algorithm analysis and design. Several such techniques focus on the automated application of empirical methods to existing algorithms, and have led to substantial improvements in the state of the art for solving many challenging computational problems (see, e.g., [81, 11, 78]). Since these techniques operate upon algorithms, we refer to them as *meta-algorithmic* techniques. Familiar meta-algorithmic analysis

techniques include the characterization of a single algorithm's performance using a solution cost distribution, and the comparison of several algorithms using statistical tests (see, e.g., [35: Ch. 4]). Prominent meta-algorithmic design techniques include algorithm configuration (e.g., F-Race [7, 4], PARAMILS [39, 40], GGA [2]), per-instance portfolio-based selection (e.g., SATZILLA [64, 81], 3S [59]), and per-instance configuration (e.g., HYDRA [83, 84], ISAC [52], CluPaTra [58]). Unfortunately, despite the potential demonstrated in this growing body of literature, the empirical methods used in practice often remain rather elementary. We believe that this is due to the fact that many practitioners do not have sufficient knowledge of more sophisticated meta-algorithmic techniques, and that automated implementations of these techniques are often difficult to use if publicly available at all. These limitations impose a significant barrier to the further development and wider adoption of automated meta-algorithmic analysis and design techniques.

Many of the conceptual and practical obstacles facing developers and end users of automated meta-algorithmic analysis and design techniques arise from challenges associated with their underlying empirical methods. By their nature, empirical methods rely on access to performance data, and they often make assumptions (for example, distributional assumptions) about this data that may or may not not hold in a given application scenario. Even once an appropriate method is selected, coordinating the potentially thousands or even millions of individual algorithm runs required for data collection is a significant undertaking; in our experience it is not rare for practitioners using meta-algorithmic techniques to spend more time managing and troubleshooting computational experiments than performing any other aspect of their work. These observations highlight two pressing needs of the empirical algorithmics community:

1. the need for a unified conceptual framework to facilitate the design, comparison, and selection of meta-algorithmic analysis and design techniques; and

2. the need for an intuitive, robust software platform to facilitate the implementation and operation of automated tools based on these techniques.

In this work, we aim to satisfy both of these needs. We first formalize a set of fundamental concepts that are useful to model and critically assess meta-algorithmic techniques, and use them to describe, relate, and compare a variety of established meta-algorithmic design and analysis techniques from the literature. We then develop a software framework using a design informed by these key concepts that supports the implementation and application of arbitrary automated meta-algorithmic analysis and design tools. We use this framework to re-implement, generalize, and empirically evaluate the same established techniques, thereby providing both a unified framework for the comparison, selection, and operation of meta-algorithmic design and analysis techniques, and also a proven platform for their continued development.

1.1 Contributions

The main contributions of this thesis arise from our development of HAL, the High-performance Algorithm Laboratory – a computational environment for empirical algorithmics. HAL was conceived to support the computer-aided design and empirical analysis of high-performance algorithms by means of a wide range of ready-to-use, state-of-the-art meta-algorithmic analysis and design procedures. HAL was also designed to facilitate the development, dissemination, and ultimately wide adoption of novel analysis and design techniques. In describing the principles underlying our design for HAL, we address the need for a conceptual framework for researching meta-algorithmics techniques; HAL itself addresses the need for a software platform with which to develop automated implementations of these techniques.

During the early stages of designing HAL, we realized that appropriately formalized notions of meta-algorithmic procedures, and of the underlying problems solved by these procedures, were necessary to meet our practical goals for the system. This formalization, presented in Chapter 3 and applied to the design of HAL in Chapter 4, promotes ease of use by inducing a set of common fundamental components with which to describe, and ultimately implement and operate, all meta-algorithmic analysis and design techniques. One benefit of this principled design is that it facilitates the selection and application of multiple (or alternative) analysis or design procedures. For example, configuration procedures like PARAMILS and GGA solve the same underlying problem, and with HAL it is easy to conduct analogous experiments using both of them (see Chapter 5). Similarly, this design simplifies the combination of various procedures (such as algorithm configuration and per-instance portfolio-based selection to perform per-instance configuration; see Chapter 6), as well as their sequential application (such as design followed by comparative performance analysis; see Chapters 5 and 6). Finally, by adopting a design based on fundamental meta-algorithmic concepts, we were able to implement HAL in an intuitive and extensible way.

HAL offers several practical features important for practitioners working in empirical algorithmics. First, to support large computational experiments, HAL uses a database to automatically archive and manage data related to algorithms, benchmark instances, and experimental results. Second, while HAL can be used on a standalone computer, it also supports distributed computation on computer clusters. Third, it promotes reproducibility by allowing researchers to export experiment designs, including all instances, solvers, and settings; another user can easily import the experiment design into a new HAL installation and replicate the original experiment. All of these features were exploited when conducting the large-scale experiments of Chapters 5 and 6.

HAL was also designed to facilitate the development and critical assessment of automated meta-algorithmic procedures. To this end, HAL is realized as an extensible, open environment that offers robust support for recurring tasks such as launching, monitoring, and analyzing individual algorithm

runs. This built-in support simplifies implementation and testing of meta-algorithmic procedures, as demonstrated in Chapter 6, and has enabled us to provide a wide variety of ready-to-use meta-algorithmic procedures. As discussed in Chapter 3, the current version of HAL includes automated analysis procedures for characterizing single algorithms, for comparing two algorithms, and for comparing multiple algorithms; it includes automated design procedures for performing algorithm configuration, per-instance portfolio-based selection, and per-instance configuration. In short, HAL allows developers to focus more on building useful and powerful meta-algorithmic procedures and less on the infrastructure required to support them. We hope that this will help to bring about methodological progress in empirical algorithmics, and specifically in the development of novel meta-algorithmic procedures, incorporating contributions from a broad community of researchers and practitioners.

1.2 Thesis Organization

In Chapter 2, we discuss the state of the art in empirical-performance-oriented algorithm analysis and design, including both manual and automated techniques. In Chapter 3, we develop a formal description of the meta-algorithmic concepts that underly the design of HAL, and use this formalism to describe and compare analysis procedures for single-algorithm analysis, pairwise comparison, and *k*-way comparison, and design procedures for algorithm configuration, per-instance portfolio-based selection, and per-instance configuration. In Chapter 4, we identify the technical functionality (related to algorithm execution, data management, statistical analysis, and usability) that is required to meet our design goals, and explain how our software implementation of the HAL framework provides an extensible environment for empirical algorithmics research that satisfies these requirements. In Chapter 5 we conduct a sequence of algorithm analysis and design tasks on state-of-the-art solvers for Boolean satisfiability and mixed-integer programming problems by using HAL's automated meta-algorithmic procedures for analysis and algorithm configuration. In Chapter 6 we describe the development and testing of additional design procedures for per-instance portfolio-based selection and per-instance configuration. Finally, we conclude with a summary of the capabilities of the HAL framework in Chapter 7, as well as a discussion of ongoing HAL developments and promising avenues for future research.

Chapter 2

Background and Related Work

Empirical algorithmics – the study of computer algorithms in terms of their practical performance on real-world problems – is a dynamic area of computer science. Perhaps because they are fundamentally conceptual artifacts, algorithms have traditionally been studied theoretically, using the tools of discrete mathematics and logic. However, as both computer systems and the algorithms they execute have increased in complexity and sophistication, the inability of theoretical analysis alone to explain and precisely predict algorithm behaviour has become increasingly apparent. For example, there are many problems for which theory does not predict that efficient algorithms exist, but which nonetheless are regularly solved by modern algorithms. A diverse sample of such problems and their algorithms is seen in the various DIMACS Implementation Challenges [19] (including network flow and matching problems [47], shortest path problems [16], and NP-hard problems like maximum clique, graph colouring, Boolean satisfiability, and the travelling salesman problem [49, 48]). By adopting statistical techniques of the kind long used in other sciences, the empirical algorithmics community aims to augment our theoretical understanding and to drive continued progress in algorithmics research.

There are two major facets of empirical algorithmics. The first deals with the empirical analysis of computer algorithms, and is closely related to the other empirical sciences in that it is primarily concerned with the characterization of complex systems through sound application of statistical techniques. The second is concerned with algorithm design and implementation, and represents the combination of these statistical techniques with ideas from computer science and software engineering. In this chapter, we discuss these two facets in turn, exploring both established methodologies and automated applications thereof. We identify the need for a framework that unifies empirical analysis and algorithm design, and proceed to develop and demonstrate our implementation of such a framework in the remainder of this thesis.

2.1 Algorithm Analysis

Just as in other sciences, the primary role of empirical methods in algorithmics research is to characterize observable phenomena – in this case, the behaviour of algorithms when executed by computers. There has been significant methodological progress in this analytic aspect of the field since the seminal paper by Crowder *et al.* [15] proposed guidelines for reporting computational experiments, such that many of its techniques are mature enough to be considered best practices and included in textbooks (see, e.g., [13, 35]; in particular, the presentation of Hoos and Stützle informs much of the following subsection). Despite this, these methods are not so universally adopted as to preclude discussion here, especially as they sometimes differ from more familiar techniques due to the peculiarities of computer algorithms as contrasted with the physical, biological, or social systems studied in other empirical sciences.

2.1.1 Empirical Analysis Techniques

The most basic task in empirical algorithmics is to measure the performance of a single deterministic algorithm solving a single problem instance. In principle this can be accomplished by observing a single run of the algorithm to completion, but the resultant characterization depends entirely on the performance metric used to quantify this observation. Using simple *wall-clock runtime* is problematic, as the task-switched multitasking nature of modern operating systems can result in arbitrarily long delays during execution. *CPU time* is often measured instead, but this too has limitations in repeatability (for example, due to memory paging and CPU cache effects), comparability (due to its inherent implementation dependence), and descriptive power. The use of representative operation counts (i.e., *run length*) has been advocated to avoid these issues (see, e.g., [1]), but they in turn have their own limitations (such as when assessing parallel algorithms with significant communication bandwidth requirements). Finally, alternative measures of *solution quality* are often of interest; for example, when assessing an optimization algorithm under a fixed computational resource budget, or when measuring the error rate of a classification algorithm.

In more complex situations, informative empirical analyses typically require many algorithm runs. When assessing randomized algorithms, otherwise identical runs with different random seeds can be measured to construct an empirical estimation of the *runtime distribution* (RTD) that describes the effect of randomization on the algorithm's runtime performance when solving the instance. If the algorithm is an optimization algorithm, then multiple runs of some fixed duration can be measured to estimate the *solution quality distribution* (SQD) that describes the effect of randomization on solution quality; or more ambitiously, for a variety of fixed durations[1] to estimate the

[1] In practice, it is common for optimization algorithms to report *solution quality over time* (SQT) during the course of execution, obviating the need to run for multiple fixed durations.

bivariate RTD that describes the effect of randomness on runtime and solution quality simultaneously. When assessing performance on multiple instances, a *solution cost distribution* (SCD) over the instances can be estimated by estimating (bivariate) RTDs for the instances, and then computing some appropriate statistic (e.g., the median) of each. Finally, when comparing multiple algorithms, independent RTDs and SCDs can be estimated for each algorithm under consideration and then compared. Most empirical algorithm analysis techniques involve the application of standard statistical methods to one or more of these empirically-estimated performance-characterizing distributions.

An important characteristic of these distributions is that they are rarely Gaussian[2], but instead are often irregular and not infrequently heavy-tailed. This means that it can be important to adopt robust summary statistics instead of, or at least in addition to, more familiar alternatives; for example, it can be helpful to measure central tendency by the median instead of the mean, and dispersion by the interquartile range instead of the standard deviation. The frequent irregularity of these distributions also motivates direct visualization, for example by inspecting plots of the respective empirical *cumulative distribution functions* (CDFs), as a means of detecting interesting behavioural characteristics like multimodality or pathological performance issues like stagnation.

Similar considerations factor into empirical best practices for the comparison of two or more algorithms. Non-parametric statistical hypothesis tests are often more appropriate than their more more familiar parametric counterparts, which typically make distributional assumptions that are likely to be violated in empirical work. For example, using the Wilcoxon signed-rank test to quantify the significance of observed performance differences between two algorithms does not require the normality assumptions of a paired Student's t-test; similarly, using the Spearman coefficient for quantifying pairwise performance correlations does not require the normality assumptions of the related Pearson coefficient. Again due to the typically non-Gaussian nature of the underlying performance distributions, comparative visualizations such as scatter plots and overlaid single-algorithm distributional plots are valuable for discovering phenomena not captured by such tests.

All of these considerations must be taken into account when performing large-scale empirical analyses. First, the performance metrics, summary statistics, statistical tests, and visualizations employed in any analysis must be appropriately chosen and their results carefully interpreted. Second, the performance-characterizing distributions that these techniques operate upon must be accurately estimated by conducting many individual algorithm runs, raising a variety of issues related to efficient and robust use of computational resources. In order to effectively conduct large-scale empirical analyses, these challenges are typically met through automation.

[2]In fact, RTDs are never Gaussian, because negative runtimes are impossible and CPU operations are discrete. However, and more fundamentally, empirically-estimated performance-characterizing distributions tend not to be well-approximated by Gaussian distributions.

2.1.2 Analysis Tools

As computational experiments have become larger and more complex, the tools used to conduct them have become increasingly sophisticated. These tools typically fall into one of two categories: they are either flexible, general-purpose *numerical computing packages* that provide programmatic libraries of analysis functionality, or they are specialized *experimental frameworks* that support algorithm performance data acquisition and analysis, often through interactive graphical user interfaces. In our work, we developed a framework that, among other contributions, combines the advantages of these two types of analysis tools.

Numerical computing packages such as R [68], MATLAB [60], and SciPy [51] provide libraries of general-purpose statistical and mathematical routines that can be used to automate experimental analyses. Because they make relatively few assumptions about the data being analyzed and are highly customizable, these packages are regularly employed by a large number of practitioners from a variety of fields. This has at least three benefits; first, these packages tend to implement a wide array of potentially useful routines; second, these routines tend to be implemented correctly and efficiently; and third, it is possible to leverage these routines to build sophisticated tools. However, this generality also means such systems do not provide complete, ready-to-use algorithm analysis procedures, and moreover that they do not provide any functionality to address the significant practical challenges associated with collecting large quantities of algorithm performance data. Instead, users must write (or adapt) custom data collection and analysis programs every time a new experiment is conducted. This exposes practitioners to the ongoing risk of introducing experimental design or implementation errors, and severely impedes third-party reproducibility of resultant computational experiments.

There have been attempts to address these issues with general-purpose numerical computing packages by designing experimental frameworks for empirical algorithmics in particular. Many such frameworks are quite application-specific, supporting automated evaluation of algorithms that either implement a particular interface or use specific software components. Examples of application-specific frameworks include the JAMES II system for modelling and simulation of multi-agent systems [20], the UBCSAT environment for local-search SAT solvers [76], and the HyFlex framework for hyper-heuristic algorithms [10] (see also Section 2.2). More implementation-agnostic frameworks include the PAVER system [62], which performs automated performance analysis of arbitrary optimization algorithms through a web-based interface, but requires that raw performance data be collected by some separate, problem-dependent tool. On the other hand, ExpLab [32] provides tools for reproducible experimental data collection, but only very basic analysis tools. The SatEx system [71] used to conduct the SAT Competition (see, e.g., [72]) automates distributed performance data collection and archiving, and also provides a web interface to analyse results (including sum-

mary statistics and diagnostic plots); unfortunately, SatEx is neither publicly available nor easily applicable to problems other than SAT. EXACT [31] supports the specification, execution, and analysis of computer experiments for arbitrary problem domains, although its authors note it provides only limited support for distributed computation and statistical analysis, and has no interactive user interface. Inspired by such tools, EDACC [5] is an experiment management framework that similarly supports distributed algorithm execution, centralized data storage, and interactive web-based analysis tools, but in addition is publicly available and, in recent versions, is problem domain independent; of these frameworks, it is perhaps most closely related to our own work. Unlike the framework we present in this thesis, however, neither EDACC nor any other experimental framework of which we are aware has been designed to facilitate third-party development and application of new automated empirical techniques.

2.2 Algorithm Design

Given a limited budget of human and computational resources, the algorithm developer is tasked with identifying and implementing an efficient algorithm to solve a problem of interest from a vast space of possible designs. An algorithm design technique is any strategy that assists developers to identify and explore structured regions of design space that are likely to contain high-performance designs. Devising effective algorithm design techniques is one of the central goals of computer science; in this section, we describe and relate two classes of design techniques from the literature: *direct* techniques (including most traditional design techniques), and *meta-algorithmic* techniques. We identify the potential of meta-algorithmic techniques to advance the state of the art in high-performance algorithm design for a wide variety of problem domains, and motivate the need for a unified empirical algorithmics framework in order for this potential to be reached. The main contribution of this thesis lies in the development of such a framework.

2.2.1 Direct Design Techniques

When the design space identified by an algorithm design technique is structured in terms of the method by which individual problem instances are solved, we refer to that technique as a *direct* design technique. Most techniques covered by introductory algorithmics texts (see, e.g., [14, 54]), including those used to develop *greedy*, *divide and conquer*, and *dynamic programming* algorithms, outline systematic methods for efficiently constructing solutions to problems that exhibit certain properties; as such, they are canonical examples of direct techniques. The challenge for an algorithm developer using these techniques is to identify which of them (if any) is appropriate for a problem at hand, and then to formulate the problem in a way that is amenable to the chosen technique. Done correctly, this usually leads to a principled algorithm that admits theoretical performance analysis.

For many practical problems, especially those known to be solvable in polynomial time, skilled use of these techniques can yield complete, exact algorithms that are asymptotically efficient; well-known examples include Prim's greedy algorithm for finding minimum spanning trees in weighted graphs [67] and Dijkstra's related dynamic programming algorithm for computing single-source shortest paths in non-negatively weighted graphs [18].

For more computationally challenging tasks (such as solving large instances of polynomial-time solvable problems, or instances of problems not thought to be polynomial-time solvable), even asymptotically optimal complete algorithms can prove too computationally expensive for practical use. In some cases, acceptable approximation algorithms can be designed using similar methods, but in many others this is not possible and alternative design techniques must be employed. If the task at hand can be cast as a combinatorial optimization problem, heuristic search techniques can be used instead. The field of *metaheuristics*[3] is concerned with general-purpose heuristic techniques, and notably includes forms of stochastic local search that have been used to design state-of-the-art algorithms for a number of domains; for detailed coverage of metaheuristics and their successes, see e.g., [65, 26, 35]. Many metaheuristic techniques are largely domain independent; problem-specific details are usually encapsulated in a few easily abstracted components. Several software frameworks exploit this conceptual separability by supplying implementations of metaheuristic control logic; to use these frameworks, the algorithm developer is only required to configure this logic and to implement the necessary problem-specific components (see, e.g., [17, 10]).

Unlike the algorithms that result from systematic design techniques, heuristic algorithms are often difficult to analyse theoretically and rarely come with attractive asymptotic performance guarantees. Since theory is unable to effectively guide design space exploration, the associated design process is usually much more dependent on implementation and empirical evaluation tasks (using, e.g., the techniques discussed in Section 2.1). Some recent computer-aided direct design techniques explicitly couple these design, implementation, and evaluation tasks by automating exploration of the structured design spaces they provide. For example, *program synthesis* techniques generate both an executable implementation and an associated correctness proof from a formal specification (e.g., via transformations of the specification, or application of a predefined program schema); synthesis tools have been used for variety of applications (see, e.g., [74, 80]), although their focus has tended more towards correctness than empirical performance. Other work proposes treating algorithm design itself as a combinatorial optimization problem and applying metaheuristic techniques (see, e.g., [12]); *genetic programming* is a notable example of this approach, in which a population of candidate algorithms is evolved from a developer-specified set of sub-algorithmic components (e.g., input values, constants, functions) in order to optimize an evaluation function that considers both

[3] A related field, hyper-heuristics, is concerned with heuristics for designing metaheuristic techniques (see, e.g., [9]). In this discussion we consider hyper-heuristics to be a class of metaheuristics and do not discuss them separately.

correctness and performance (see, e.g., [66, 22]). These and other computer-aided design techniques (including several of the meta-algorithmic techniques of Section 2.2.2) are discussed further in [33].

2.2.2 Meta-algorithmic Design Techniques

When the design space identified by an algorithm design technique involves the the use of existing algorithms for the problem at hand, we refer to that technique as a *meta-algorithmic* design technique. Meta-algorithmic design techniques are not direct design techniques because they do not directly address the method by which individual problem instances are solved; instead, they help algorithm developers optimize overall empirical performance by building upon existing algorithms. Meta-algorithmic design techniques are typically automated, making design decisions based on empirical analyses of subsidiary algorithm performance, and their inherent problem independence means that (unlike automated direct techniques) they can, at least in principle, be used to improve the state of the art in any domain for which existing algorithms are available. In this way, meta-algorithmic design techniques are complementary to more traditional direct design techniques.

Algorithm configuration is a particularly well-studied meta-algorithmic design technique that has contributed to the state of the art for several problem domains. Given a parameterized algorithm for some problem of interest, algorithm configuration techniques explore the design space defined by its parameters to identify high-performance designs optimized for a particular set of problem instances. A variety of domain-independent automated algorithm configuration tools have been described in the literature. For example, F-Race [7] implements a racing procedure that uses statistical tests to identify high-performance designs from the space of possible parameter configurations; it has had a variety of successful applications, including to the travelling salesman, vehicle routing, scheduling, and course timetabling problems (see, e.g., [4, 8]). Other tools use local search to explore the design space: PARAMILS [39, 40] uses an iterated local search, whereas GGA [2] uses a genetic algorithm. Both of these have been applied (either directly or as part of a per-instance configuration procedure; see below) to state-of-the-art algorithms for boolean satisfiability (SAT) [39, 38, 2, 83] and mixed-integer programming (MIP) [41, 84, 52] problems, in some cases leading to order-of-magnitude performance improvements. In addition, PARAMILS has been successfully applied to state-of-the-art algorithms for timetabling [11] and planning [21] problems, and has also been used to repeatedly advance the state of the art in local search for SAT by configuring novel algorithms designed expressly for automatic configuration [53, 77, 78]. Finally, recent work in algorithm configuration has focused on learning performance-predictive models to guide the design space search; SMBO [44] is an example of this approach that has also been demonstrated on SAT and MIP problems. The procedures underlying all of these automated approaches are described further in Chapter 3.2.2.1.

Another class of meta-algorithmics design techniques is based on the idea that performance can be improved by using a set of algorithms rather than a single algorithm, provided that the per-instance performance of individual algorithms in the set is sufficiently uncorrelated; such a set is called an *algorithm portfolio* in analogy with investment portfolios from finance. One well-known portfolio technique is based on solving the *algorithm selection problem* [69]. Per-instance portfolio-based algorithm selection techniques explore the design space of algorithms that select and execute one algorithm[4] from an algorithm portfolio on a per-instance basis (see, e.g., [56, 30]). Solvers designed using a portfolio-based selection approach have dominated recent SAT Competitions (SATZILLA [81]; 3S [59, 46]). Despite these high-profile successes, we are not aware of a publicly-available, domain-independent tool for automatically constructing per-instance portfolio-based selection algorithms other than the generalization of SATZILLA that we present in Chapter 6 of this thesis.

Closely related to per-instance portfolio-based algorithm selection techniques are *sequential* and *parallel portfolio* techniques. Parallel portfolio techniques concern the space of algorithms that run several (or even all) portfolio components, either sequentially or in parallel, until the problem instance is solved. Recent work has considered both static portfolios (see, e.g., [36, 27]) and dynamic, or per-instance, portfolios (see, e.g., [23, 75]). Of these, static portfolio strategies have had notable practical success, winning recent planning competitions (PbP [25], a round-robin sequential approach) and performing surprisingly well in the most recent SAT Competition (`ppfolio` in [46]; both sequential and parallel variants won multiple medals). Dynamic portfolios have yet to make a significant practical impact in solver design; we suspect this is at least partially due to the unavailability of associated automated design tools.

Algorithm configuration and portfolio-based techniques have complementary strengths: configuration is especially well-suited to instance sets that are homogeneous (in the sense that all problem instances are of comparable difficulty for the designs under consideration), whereas portfolio techniques are ideal for more heterogeneous domains. Several recent lines of work have combined these approaches to implement *per-instance configuration* techniques. HYDRA [83, 84] iteratively uses an algorithm configuration procedure to obtain new designs for addition to a per-instance portfolio-based selector, in each iteration optimizing performance on the full training instance set according to a metric that reflects net contribution to the existing portfolio. In existing implementations of HYDRA, choices for algorithm configuration and portfolio-based selector design procedures were fixed to PARAMILS and SATZILLA respectively. ISAC [52] (using GGA for configuration) and CluPaTra [58] (using PARAMILS) take a different approach: the training set is clustered into relatively homogeneous partitions, each of which is then used to obtain a design via algorithm configura-

[4] In the interest of robustness, practical implementations of portfolio-based selection techniques like SATZILLA may sometimes execute multiple algorithms; see Chapter 6 for details.

tion and to perform subsequent selection between these designs. All three approaches have shown clear advantages over algorithm configuration alone. However, as was the case for portfolio-based selection, the generalization of HYDRA we develop in Chapter 6 is the only automated domain-independent per-instance configuration tool of which we are aware that is presently available for public use.

Due to their many high-profile successes, we expect interest in meta-algorithmic design techniques to continue to grow, and that resultant applications will continue to advance the state of the art in a diverse range of problem domains. More ambitiously, we predict that meta-algorithmic design techniques will ultimately become an indispensable part of every algorithm developer's repertoire. At the same time, we currently observe an unfortunate paucity of publicly-available, domain-independent, easy-to-use tools for the automated application of many of these techniques. Since such automated tools must ultimately rely on subsidiary empirical analyses, their development and use is subject to the same challenges that arise when performing any large-scale empirical analysis – in particular, conceptual challenges related to the selection and use of appropriate statistical techniques, and logistical challenges related to the management of large numbers of algorithm runs. Based on our experience, we believe these challenges are a significant factor in the limited availability of automated meta-algorithmic design tools. However, we also observe that any system that supports the automation of one empirical technique should, in principle, be able to support many others. From these observations, we arrive at the major goal of this work: to build HAL, a unified framework for the development and application of automated empirical analysis and meta-algorithmic design techniques.

Chapter 3

Meta-algorithmic Problems

The ideas of meta-algorithmic analysis and design problems and procedures are central to the work presented in this thesis. However, they are concepts which have not to our knowledge been explicitly studied in a general sense in the literature. In this chapter we formally introduce and develop these ideas, and use them to provide a conceptual framework that we use to frame the algorithm analysis and design techniques of Chapter 2. We ultimately use the concepts developed here to inform the design of both the HAL framework and of the various meta-algorithmic procedures it supports.

Before discussing meta-algorithmic problems and procedures in detail, we draw a parallel with the idea of design patterns from software engineering (see, e.g., [24]). Design patterns identify commonly-encountered software engineering problems, and describe specific proven solutions to them. Similarly, meta-algorithmic problems reflect challenges that arise in algorithm development, and meta-algorithmic procedures constitute specific solutions to these challenges. However, just as choosing between related design patterns for a specific application relies on understanding the benefits and drawbacks of each, so too does selecting and applying an appropriate meta-algorithmic procedure. By presenting meta-algorithmic work from the literature in terms of a conceptually unified framework (Section 3.2), and ultimately by providing a software environment in which the associated procedures can be directly compared, we hope to contribute to such an understanding.

3.1 Fundamentals

We begin by defining a computational *problem* as a specification of a relationship between a set of possible inputs I_p and a corresponding set of possible of outputs O_p. An *instance* of problem p is any element $\mathbf{x} \in I_p$, and each instance $\mathbf{x}$ is associated with one or more *solutions* $\mathbf{y} \in O_p$. Thus, we can represent a problem p by a function $p : I_p \to 2^{O_p}$ (where 2^X is the power set of X), or equivalently, as the set of all instance-solution pairs $L_p = \{(\mathbf{x},\mathbf{y}) \mid \mathbf{x} \in I_p \wedge \mathbf{y} \in p(\mathbf{x})\} \subseteq I_p \times O_p$; in

problem
instance
solution

this thesis we adopt the former. Without loss of generality, we assume that both $\mathbf{x}$ and $\mathbf{y}$ are vectors (whose types are determined by the associated problem); in the following we specify such a vector by enclosing its elements in angle brackets.

In order to define a specific problem p, we formally describe a general instance of that problem, as well as its corresponding solution(s). We leave I_p implicitly defined as the set of all possible instances, and O_p as the set of all possible solutions. For example, SAT— the Boolean satisfiability problem — can be defined as:

Instance: $\langle V, \psi \rangle$, where:

V is a finite set of variables

ψ is a Boolean formula containing variables from V

Solution: $\langle s \rangle$, where:

$$s = \begin{cases} \text{true} & \text{if there exists some } K : V \to \{\text{true}, \text{false}\} \text{ such that } \psi = \text{true under } K \\ \text{false} & \text{otherwise} \end{cases}$$

Thus, $\mathbf{x} = \langle \{a, b\}, (\neg a \vee b) \wedge (a \vee b) \rangle$ is a SAT instance with a unique solution: $\text{SAT}(\mathbf{x}) = \{\langle \text{true} \rangle\}$.

Three useful concepts for discussing problems are instance compatibility, solution compatibility, and reducibility. Intuitively, a problem p is instance-compatible with another problem q if any instance of q can be translated into an instance of p, solution-compatible with q if any solution of q can be translated into a solution of p, and reducible to q if the ability to solve q provides the ability to solve p. More formally, given two sets $C \subseteq X$ and $D \subseteq X$, C is reducible to D (written $C \leq D$) via *reduction function* f if there exists some $f : X \to X$ such that $x \in C \Leftrightarrow f(x) \in D$; the computational complexity of the reduction is simply the complexity of f. Using this definition, we say p is *instance-compatible* with q if $I_q \leq I_p$ and *solution-compatible* with q if $O_q \leq O_p$.

compatible

If p and q are both decision problems (i.e., $O_p = O_q = \{\text{true, false}\}$), p is *mapping reducible* to q (written $p \leq_m q$) if $I_p \leq I_q$; in this case, instance-compatibility is equivalent to mapping reducibility.[1] In more general cases, another form of reducibility is useful. Defining an *oracle* for problem q as a function $\Omega_q : I_q \to O_q$ that satisfies $\forall \mathbf{x} \in I_q$, $\Omega_q(\mathbf{x}) \in q(\mathbf{x})$ and that can be queried at no cost, we say that p is *Turing reducible* to q (written $p \leq_T q$) if there exists some deterministic procedure that computes $T_p^q : I_p \to O_p$ by querying Ω_q. Turing reducibility is a generalization of mapping reducibility; any reduction $p \leq_m q$ can be implemented by a procedure that converts the input p-instance into a q-instance via the reduction function and solves it with a single oracle query (i.e., $T_p^q : \mathbf{x} \mapsto \Omega_q \circ f(\mathbf{x})$). We thus extend the concept of mapping reducibility to arbitrary solution-compatible problems by allowing a final solution conversion step using a second reduction function

mapping reducible

oracle

Turing reducible

[1] We note that most definitions of mapping (or many-to-one) reducibility for decision problems are stated in terms of formal *languages*, not instance sets (see, e.g., [73, 14]). Our definition is compatible with these under the assumption that problem instances are represented by words in an appropriate formal language.

g (i.e., $p \leq_m q$ if there exists some $T_p^q : \mathbf{x} \mapsto g \circ \Omega_q \circ f(\mathbf{x})$ that solves p). In the language of mapping reducibility, we say $p \leq_m q$ if $I_p \leq I_q$ via f, $O_q \leq O_p$ via g, and $(g(\mathbf{y}) \in p(x)) \Leftrightarrow (\mathbf{y} \in q \circ f(\mathbf{x}))$.

For example, consider the model-finding variant of SAT, SAT_{MF}:

Instance: $\langle V, \psi \rangle$, where:

V is a finite set of variables

ψ is a Boolean formula containing variables from V

Solution: $\langle M \rangle$, where:

$$M = \begin{cases} \text{any } M_\psi : V \to \{\text{true}, \text{false}\} \text{ such that } \psi = \text{true under } M_\psi & \text{if such an } M_\psi \text{ exists} \\ M_0 \ \text{(some indicator that no such } M_\psi \text{ exists)} & \text{otherwise} \end{cases}$$

While $\text{SAT}_{\text{MF}} \not\leq_m \text{SAT}$,[2] we can easily see that $\text{SAT} \leq_m \text{SAT}_{\text{MF}}$; for example, via:

$$f : \mathbf{x} \mapsto \mathbf{x}$$

$$g : \mathbf{y} \mapsto \begin{cases} \langle \text{false} \rangle & \text{if } \mathbf{y} = \langle M_0 \rangle \\ \langle \text{true} \rangle & \text{if } \mathbf{y} = \langle M \rangle \text{ and } M \in O_{\text{SAT}_{\text{MF}}} \\ \langle G_0 \rangle & \text{(some invalid input indicator) otherwise}^3 \end{cases}$$

Returning to the previous example instance $\mathbf{x}$, there are two model-finding solutions; i.e., $\text{SAT}_{\text{MF}}(\mathbf{x}) = \{\mathbf{y}_1, \mathbf{y}_2\}$, where $\mathbf{y}_1 = \langle (a = \text{true},\ b = \text{true}) \rangle$ and $\mathbf{y}_2 = \langle (a = \text{false},\ b = \text{true}) \rangle$.

It is often useful to consider restricted classes of instances of a particular problem. For example, SAT instances derived from industrial problems are quite different from randomly-generated instances, and as such they are frequently studied separately. An *instance distribution* for problem p describes such a restricted class of instances, and is formally defined by a probability mass function[4] $\mathscr{D} : I_p \to [0,1]$, where $\sum_{\mathbf{x} \in I_p} \mathscr{D}(\mathbf{x}) = 1$. However, in practice it is somewhat rare to work with instance distributions directly; instead, one typically encounters representative benchmark instance sets. An *instance set* is any set $S \subseteq I_p$, where it is assumed that $S_{\mathscr{D}}$ was constructed by sampling from I_p according to some underlying distribution $\mathscr{D}$ of interest.

instance set

An *algorithm* is a well-defined computational procedure that implements some function $a : I_a \to O_a$, where we again assume without loss of generality that inputs $\mathbf{u} \in I_a$ and outputs $\mathbf{v} \in O_a$ are vectors. Executable algorithm implementations typically require inputs $\mathbf{u}$ to include a problem instance in some encoding format, and may require additional elements such as a random seed or a CPU time budget; they typically produce outputs $\mathbf{v} = a(\mathbf{u})$ that include a solution to the input

algorithm

[2] One can show more generally that $\text{SAT}_{\text{MF}} \leq_T \text{SAT}$, although we do not provide the oracle Turing machine here.

[3] The third case is a technicality; it is required because $g : X \to X$ must be defined over some $X \supseteq O_{SAT} \cup O_{SAT_{MF}}$. In order for $(g(\mathbf{y}) \in p(x)) \Leftrightarrow (\mathbf{y} \in q \circ f(\mathbf{x}))$ to be true, g cannot map elements $\mathbf{y} \notin O_{SAT_{MF}}$ to O_{SAT}; for example, $g(\langle true \rangle)$ must be defined, but can be neither ⟨true⟩ nor ⟨false⟩.

[4] In the case of continuous I_p, an instance distribution is instead defined by a probability *density* function, which satisfies $\int_{\mathbf{x} \in I_p} \mathscr{D}(\mathbf{x})\, d\mathbf{x} = 1$

instance in some format, as well as other elements including measures of progress during execution. Intuitively, then, we say that an algorithm solves a problem if any instance of that problem can be encoded into its inputs, and a solution to that instance can be decoded from its outputs upon execution. More formally, algorithm a solves problem p (equivalently, a is a p-solver) if $p \leq_m p_a$, where $p_a : I_a \rightarrow \mathscr{P}(O_a)$ is defined by $p_a : \mathbf{u} \mapsto \{a(\mathbf{u})\}$, and where the asymptotic complexity of a is (weakly) greater than that of the reduction (i.e., it is meaningful to say that a, and not a reduction function, solves p).

Importantly, this definition allows algorithms to require inputs other than those directly associated with the problems they solve. We distinguish three types of algorithm input: the problem instance to be solved, algorithm-specific *parameters* that control the logic employed to solve the instance, and any other *settings* that might be required for operation (e.g., a random seed or a CPU time budget). We refer to algorithms that have parameters as *parameterized*, and to the rest as *parameterless*. Any parameterized algorithm can be made parameterless by fixing values for all of its parameters. Thus, a parameterized algorithm defines a structured space of parameterless algorithms; we henceforth use a to denote a single parameterless algorithm, and $\mathscr{A}$ to denote a space of parameterless algorithms.

parameter

setting

When an algorithm a is executed on input $\mathbf{u}$ by a computer system C, the resulting *algorithm run* $\mathbf{r} = \langle a,\ \mathbf{u},\ C \rangle$ has a variety of observable properties. A *performance evaluator* $\delta : R \rightarrow \mathbb{R}$ quantifies the observable properties of a single run; here, the domain R indicates the set of all possible algorithm runs. An *aggregator* $\mu : \mathbb{R}^k \rightarrow \mathbb{R}$ summarizes a collection of quantifications; in particular, those produced by a performance evaluator. Common performance evaluators include CPU time $\delta^\tau(\mathbf{r} = \langle a, \mathbf{u}, C \rangle) \equiv$ (CPU time taken to execute a on $\mathbf{u}$ using C) and solution quality $\delta^p(\mathbf{r} = \langle a, \mathbf{u}, C \rangle) \equiv$ (value of element of $a(\mathbf{u})$ corresponding to solution quality under problem p); common aggregators include statistics like the mean, median, min, and max functions. Finally, a *performance metric* combines a performance evaluator with one or more aggregators. Many performance metrics used in the literature use a single aggregator for all purposes; for example, *median CPU time* combines δ^τ and the median, *average solution quality* combines δ^p and the mean, and *penalized average runtime (PAR-k)* combines $\delta^\tau_{k,\kappa}(\mathbf{r}) \equiv (k \cdot \kappa \text{ if } \delta^\tau(\mathbf{r}) \geq \kappa \text{ else } \delta^\tau(\mathbf{r}))$ and the mean (where k is a penalty factor for runs longer than some maximum CPU time κ; see, e.g., [40]). While these pre-defined performance metrics are often convenient, formally it is necessary to explicitly specify evaluators and aggregators separately in applications where multiple "levels" of aggregation are required (for example, to indicate aggregation of multiple runs of a randomized algorithm for each of a number of instances using the median, and again across all instances using the mean).

algorithm run

evaluator

aggregator

performance metric

PAR-k

Some problems are of interest primarily in the context of designing algorithms to solve more broadly relevant problems; we refer to such problems as *features*. Any problem q that is instance-

feature

compatible with a given problem p can be considered a *p-instance feature*; we refer to the q-solution to instance $\mathbf{x}$ as the *value* of feature q on $\mathbf{x}$, and to a q-solver as a *feature extractor* for q. In practice, *low-cost* features q that admit extractors of lower asymptotic complexity than any known algorithm for the associated problem p are the most useful; in this work, all features are understood to be low-cost. Low-cost features are critical, for example, to algorithms that rely on predictive models; in such applications, the semantics of individual features are irrelevant, so long as access to their values improves prediction quality. Useful low-cost SAT features have included measures of the number of clauses, of the number of variables, and of the progress of established SAT solvers when run for a fixed CPU time or run length [81].

feature extractor

3.2 Meta-algorithmic Problems

We can now introduce the meta-algorithmic concepts underlying the remainder of the work presented in this thesis. A *meta-algorithmic problem* is a problem whose instances contain one or more algorithms; we refer to these algorithms as *target algorithms*, and to the problems target algorithms solve as *target problems*. A *meta-algorithmic analysis problem* is a meta-algorithmic problem whose solutions include a statement about target algorithm performance; a *meta-algorithmic design problem* is a meta-algorithmic problem whose solutions include at least one algorithm. Both meta-algorithmic analysis and design problem instances typically include a performance metric and a target problem instance set, and for some problems include target problem feature extractors in addition to other target algorithm(s). Finally, a *meta-algorithm*, or *meta-algorithmic procedure*, is an algorithm that solves some meta-algorithmic problem. We refer to a meta-algorithm that solves an analysis problem as a *meta-algorithmic analysis procedure*, and one that solves a design problem as a *meta-algorithmic design procedure*.

target algorithm

meta-algorithm

In the remainder of this chapter, we present a taxonomy of meta-algorithmic analysis and design problems for which automated procedures exist, using a format inspired by similar taxonomies of software engineering design patterns (see, e.g., [24]). For each problem discussed, we introduce the **context** in which it commonly arises, and provide a formal definition in terms of a general **instance** and corresponding **solution**. We then describe and compare automated **procedures** for solving the problem, and finally identify **related problems** that are applicable in similar contexts.

3.2.1 Meta-algorithmic Analysis Problems

We initially defined an analysis problem as a meta-algorithmic problem whose solutions include a statement about target algorithm performance. Recalling that algorithm performance is defined in terms of a performance metric (consisting of an evaluator and one or more aggregators) and at least

one problem instance, and assuming that any performance statement can be broken into quantitative components (represented by one or more scalars) and qualitative components (represented by one or more visualizations), we introduce the generic meta-algorithmic analysis problem P_{A_0}:

Instance: $\langle A, S_{\mathscr{D}}, \delta, M \rangle$, where:

A is a set of parameterless target algorithms

$S_{\mathscr{D}}$ is a set of target problem instances drawn from some distribution $\mathscr{D}$

δ is a performance evaluator

M is a set of aggregators

Solution: $\langle Q, V \rangle$, where:

Q is a set of quantifications of the performance of A on $\mathscr{D}$ in terms of δ and M

V is a set of visualizations of the performance of A on $\mathscr{D}$ in terms of δ and M

A meta-algorithmic analysis problem is then any problem P_A such that P_{A_0} is both instance- and solution-compatible with P_A. Analysis problems are ubiquitous, albeit not always solved by fully automated procedures. Here, we identify the meta-algorithmic analysis problems and procedures corresponding to the methodological best practices outlined in Chapter 2.1.

3.2.1.1 Single-Algorithm Analysis

Context: Consider the task of evaluating a single algorithm on a benchmark instance distribution (represented by an instance set or, in the degenerate case, a single instance); or equivalently, of attempting to characterize an instance distribution in terms of its difficulty for a specific target algorithm. This fundamental task arises in nearly every application of empirical algorithmics, and corresponds to the *single-algorithm analysis problem*:

Instance: $\langle a, S_{\mathscr{D}}, \delta, \mu^1, \boldsymbol{\mu}^2 \rangle$, where:

a is a parameterless target algorithm

$S_{\mathscr{D}}$ is a set of target problem instances drawn from some distribution $\mathscr{D}$

δ is a performance evaluator

μ^1 is an aggregator for multiple runs on single instances $\mathbf{x} \in S$

$\boldsymbol{\mu}^2$ is a vector of aggregators for runs across multiple instances $S' \subseteq S$

Solution: $\langle \mathbf{q}, \mathbf{v} \rangle$, where:

$\mathbf{q}$ is a vector of quantifications of the performance of a on $\mathscr{D}$ in terms of δ, μ^1, and $\boldsymbol{\mu}^2$

$\mathbf{v}$ is a vector of visualizations of the performance of a on $\mathscr{D}$ in terms of δ, μ^1, and $\boldsymbol{\mu}^2$

Procedures: As single-algorithm analysis is a fundamental empirical task, a variety of distinct

manual procedures have been used to solve it in the literature. However, several best practices have emerged (as discussed in Chapter 2.1).

Where a is a deterministic algorithm, best practices suggest computing an empirical SCD over the instance set by performing one run per instance in $S_{\mathscr{D}}$ and evaluating each run using δ. The solution is then comprised of $\mathbf{q}$, containing values of the aggregators in $\boldsymbol{\mu}^2$ for this empirical SCD; and $\mathbf{v}$, containing a visualization (e.g., an empirical CDF) of the SCD.

Where a is a randomized algorithm, best practices dictate collecting empirical RTDs by performing multiple runs per instance in $S_{\mathscr{D}}$ and evaluating each run according to δ, and then aggregating across runs for each instance according to μ^1 to yield an empirical SCD. The solution is composed of statistics (from the aggregators in $\boldsymbol{\mu}^2$) and visualizations of this SCD as before, as well as those of the individual RTDs.

The *SCD-based Analysis* plugin for HAL implements both of these cases; see Chapter 5 or Appendix A for more details. Most experimental frameworks (including those discussed in Section 2.1.2) also provide automated single-algorithm analysis procedures, and often facilitate batch application of these procedures to multiple algorithms. More advanced single-algorithm analysis procedures might also report *confidence intervals* for some or all of the computed statistics (such a procedure is currently under development for HAL), or correlate instance feature values with observed performance (see also the related *scaling analysis problem* below).

Related Problems: When the goal is not to characterize the performance of a single algorithm in isolation, but to understand performance differences between several algorithms, procedures solving the pairwise comparison problem (Section 3.2.1.2) or k-way comparison problem (Section 3.2.1.3) can be more principled, more computationally efficient, and more conclusive than conducting multiple single-algorithm analyzes and manually comparing the results.

When the goal is to analyze not a single parameterless algorithm a, but a single parameterized algorithm $\mathscr{A}$, single-algorithm analysis is insufficient. If the size of $\mathscr{A}$ is relatively small, a k-way comparison between the parameterless algorithms it contains may be sufficient; otherwise, alternative analysis techniques are required. One option is to use per-instance algorithm configuration (Section 3.2.2.3) to reduce the number of parameterless algorithms to be analyzed to a sufficiently small number that k-way comparison becomes computationally feasible. However, this yields an incomplete analysis at best; we expect that recent progress in the model-based algorithm configuration literature (see, e.g., [37, 42]) will ultimately lead to procedures that attempt to more directly solve the *parameter response analysis problem*: the task of quantifying the effect of changes in parameter values on empirical performance.

When the goal is not to characterize the performance of a on $\mathscr{D}$, but on some related distribution $\mathscr{D}'$, the problem at hand is a *generalized scaling analysis problem*. The task of extrapolating algorithm performance on relatively small instances to characterize performance on larger instances is the canonical example of a scaling analysis problem, and might be solved by a meta-algorithm that learns a predictive model for algorithm performance based on instance features. Potentially useful work has been conducted in the context of per-instance portfolio-based selection (Section 3.2.2.2), but we are not aware of any automated procedures that apply this work to explicitly solve any scaling analysis problem.

3.2.1.2 Pairwise Comparison

Context: Consider the task of comparing two algorithms on an instance distribution. This task arises most commonly when assessing the effect of a design modification on algorithm performance, but also arises more generally with faced with exactly two competing algorithms for a problem of interest. In such cases, the goal is to characterize the performance differences between these two algorithms, rather than to simply provide an independent performance analysis of each. This task corresponds to the *pairwise comparison problem*:

Instance: $\langle a_1, a_2, S_{\mathscr{D}}, \delta, \mu^1, \boldsymbol{\mu}^2 \rangle$, where:

a_1 and a_2 are parameterless target algorithms

$S_{\mathscr{D}}$ is a set of target problem instances drawn from some distribution $\mathscr{D}$

δ is a performance evaluator

μ^1 is an aggregator for multiple runs on single instances $\mathbf{x} \in S_{\mathscr{D}}$

$\boldsymbol{\mu}^2$ is a vector of aggregators for runs across multiple instances $S' \subseteq S_{\mathscr{D}}$

Solution: $\langle \mathbf{q}, \mathbf{v} \rangle$, where:

$\mathbf{q}$ is a vector of statistics comparing a_1 to a_2 on $\mathscr{D}$ in terms of δ, μ^1, and $\boldsymbol{\mu}^2$

$\mathbf{v}$ is a vector of visualizations comparing a_1 to a_2 on $\mathscr{D}$ in terms of δ, μ^1, and $\boldsymbol{\mu}^2$

Procedures: As with single-algorithm analysis, a variety of manual procedures for comparing algorithms have been used in the literature, and again several best practices have been established.

Many pairwise comparison procedures require performing single-algorithm analyzes to characterize the two target algorithms, and then conducting statistical tests to determine the significance of any performance differences revealed by these analyzes. Usually, one question answered by these tests is whether either of the algorithms consistently outperforms the other; for this purpose, the non-parametric Wilcoxon signed-rank test is often an appropriate choice. Other questions may include determining whether the algorithms exhibit identical

performance (via, e.g., the Kolmogorov-Smirnov test), or to what degree their inter-instance performance is correlated (via, e.g., Spearman's correlation coefficient).

Additional visualizations are also appropriate when comparing two algorithms. While distributional plots for each of the algorithms can be overlaid to facilitate comparison, one useful visualization is often a scatter plot with which the algorithms' performance on each instance can be directly compared. In particular, these plots allow the user to quickly determine which instances are easier for which algorithm, and can indicate the existence of potentially interesting clusters of instances exhibiting similar performance characteristics.

The *Comprehensive Pairwise Comparison* plugin for HAL builds on SCD-based Analysis in order to perform pairwise performance comparison; see Chapter 5 or Appendix A for more details.

Related Problems: Single algorithm analysis (Section 3.2.1.1) is applicable when only one target algorithm is of interest, and k-way comparison (Section 3.2.1.3) is a generalization of pairwise comparison appropriate when several target algorithms are to be compared.

When the goal is not to characterize the performance of two algorithms, but rather to efficiently identify the best-performing algorithm, meta-algorithmic design problems such as algorithm configuration or per-instance configuration are more relevant; see Section 3.2.2 for more details.

3.2.1.3 k-way Comparison

Context: Consider the task of comparing a number of competing algorithms on a set of benchmark instances. This task arises most prominently when comparing a number of candidate solvers for a problem of interest (or example, when conducting a solver performance competition), but can also arise when evaluating a number of design changes during the algorithm design process. Since the goal is to characterize the performance differences between these algorithms rather than to summarize the performance of each, this corresponds to a generalization of pairwise comparison called the *k-way comparison problem*:

Instance: $\langle \mathbf{a}, S_{\mathscr{D}}, \delta, \mu^1, \boldsymbol{\mu}^2 \rangle$, where:

$\mathbf{a}$ is a k-dimensional vector of parameterless target algorithms

$S_{\mathscr{D}}$ is a set of target problem instances drawn from some distribution $\mathscr{D}$

δ is a performance evaluator

μ^1 is an aggregator for multiple runs on single instances $\mathbf{x} \in S_{\mathscr{D}}$

$\boldsymbol{\mu}^2$ is a vector of aggregators for runs across multiple instances $S' \subseteq S_{\mathscr{D}}$

Solution: $\langle \mathbf{q}, \mathbf{v} \rangle$, where:

$\mathbf{q}$ is a vector of statistics comparing a_1 to a_2 on $\mathscr{D}$ in terms of δ, μ^1, and $\boldsymbol{\mu}^2$

$\mathbf{v}$ is a vector of visualizations comparing a_1 to a_2 on $\mathscr{D}$ in terms of δ, μ^1, and $\boldsymbol{\mu}^2$

Procedures: Procedures solving the k-way comparison problem are often direct generalizations of procedures for pairwise performance comparison. However, since the number of potential pairwise analyzes increases quadratically with k, it is common to first determine whether any of the k algorithms outperform the others in a statistically significant way (via, e.g., the Friedman test). If so, pairwise *post-hoc* analyzes can be conducted to characterize other algorithms with respect to the best. The *Comprehensive k-way Comparison* plugin for HAL generalizes the Comprehensive Pairwise Performance plugin in this way; see Chapter 5 or Appendix A for more details.

Designing effective visualizations for k-way comparisons can be challenging; for example, overlaid single-algorithm distributional plots (such as individual SCDs) become increasingly difficult to interpret as k grows. Box-and-whisker plots can be useful for comparing the performance of many algorithms, visually indicating outlier instances in addition to quartile summary statistics. However, unlike distributional plots, they do not reveal other interesting characteristics, such as multimodality.

When $|S| \times k$ is large, it can become infeasible to perform an exhaustive k-way comparison; in particular, it can become infeasible to collect runtime data for each algorithm on each instance under a realistic computational resource budget. While methods to maximize performance estimate quality subject to computational resource constraints have been described in the algorithm configuration literature (see, e.g., [42]), we are not aware of any automated procedures that apply such methods to solving the k-way comparison problem directly.

Perhaps because of its relationship to solver competitions, k-way comparison is arguably a commonly-automated analysis problem. However, many systems used for this task are only *partially* automated, typically performing single-algorithm analyzes for each of the k algorithms, but stopping short of automatically conducting comparative analysis. Instead, they allow the user to manually conduct statistical tests and generate comparative visualizations based on the single algorithm analyzes; this allows for greater flexibility when performing exploratory analysis at the expense of increased susceptibility to experimental design errors and limited utility in multi-phase automated experiments. For this reason, we do not regard such systems as constituting automated k-way comparison procedures, but rather see them as experimental frameworks that facilitate the use of automated single-algorithm analysis procedures.

Related Problems: Single-algorithm analysis (Section 3.2.1.1) is applicable when $k = 1$, and pairwise comparison (Section 3.2.1.2) when $k = 2$. When k is very large, k-way comparison can become prohibitively expensive. If these k algorithms can be meaningfully combined into a single parameterized algorithm $\mathscr{A}$, parameter response analysis may be applicable; unfortunately, as discussed in the context of single-algorithm analysis, automated procedures for performing parameter response analysis are not yet available.

When the goal is not to characterize the performance of a number of algorithms, but rather to efficiently identify a *single* high-performing algorithm, meta-algorithmic design problems such as algorithm configuration or per-instance configuration are more relevant; see Section 3.2.2 for more details.

3.2.2 Meta-algorithmic Design Problems

We introduced a design problem as any meta-algorithmic problem whose solutions include at least one algorithm. We now formally define the generic meta-algorithmic design problem, P_{D_0}:

Instance: $\langle A, S_{\mathscr{D}}, \delta, M \rangle$, where:

A is a set of parameterless target algorithms

$S_{\mathscr{D}}$ is a set of target problem instances drawn from some distribution $\mathscr{D}$

δ is a performance evaluator

M is a set of aggregators

Solution: $\langle a' \rangle$, where:

a' is a parameterless algorithm that optimizes performance on $\mathscr{D}$ according to δ and M

In other words, a meta-algorithmic design problem is any problem P_D such that P_{D_0} is both instance- and solution-compatible with P_D. Here, we discuss the practical strengths and limitations of the prominent automated meta-algorithmic design procedures referenced in Chapter 2.2.2, and formally define the problems that they solve.

3.2.2.1 Algorithm Configuration

Context: When designing an algorithm for a problem of interest, the designer is often faced with decisions for which the optimal choice is unclear. Such decisions commonly include numerical values for implementation-specific constants, and categorical choices such as heuristics to use at various stages of algorithm execution. In order to avoid poor performance, the designer can defer these decisions by exposing parameters that allow them to be made or changed after

the implementation is otherwise complete. The *algorithm configuration problem* is the problem of setting such parameters in order to optimize performance on an instance distribution:

Instance: $\langle \mathscr{A}, S_{\mathscr{D}}, \delta, \mu^1, \mu^2 \rangle$, where:

$\mathscr{A}$ is a parameterized target algorithm

$S_{\mathscr{D}}$ is a set of target problem instances drawn from some distribution $\mathscr{D}$[5]

δ is a performance evaluator

μ^1 is an aggregator for multiple runs on single instances $\mathbf{x} \in S_{\mathscr{D}}$

μ^2 is an aggregator for runs across multiple instances $S' \subseteq S_{\mathscr{D}}$

Solution: $\langle a' \rangle$, where:

a' is a parameterless algorithm for the target problem corresponding to an instantiation of $\mathscr{A}$'s parameters to values that optimize aggregate performance on $\mathscr{D}$ according to δ, μ^1, and μ^2.

Procedures: Many automated algorithm configuration procedures have been presented in the literature. For the most part these procedures can be classified as either *model free* (wherein algorithm configuration is treated as a combinatorial optimization problem and solved with heuristic techniques) or *model based* (wherein a predictive model of algorithm performance as a function of instance features is learned and exploited to optimize parameter settings). Of these, model-free techniques have enjoyed significant early success, and model-based procedures are yielding increasingly promising results.

F-Race [7] is a model-free algorithm configuration procedure that is quite closely related to *k*-way comparison. In F-Race, competing parameter configurations are first analyzed on a single instance from $S_{\mathscr{D}}$, and a Friedman test is used to eliminate any statistically significantly outperformed designs. The remaining configurations are analyzed on an additional instance from $S_{\mathscr{D}}$, and the process is repeated until a single parameter configuration remains. Although the initial implementation of F-Race was largely limited to configuring algorithms with relatively few, numerical-valued parameters, subsequent improvements have substantially addressed both of these limitations (see, e.g., [4, 8]), although it remains unclear how these most recent versions of F-Race handle very large configuration spaces of the kind that have been optimized by the other configuration procedures described here.

ParamILS [39, 40] is a model-free configuration procedure based on an iterated local search over parameter space. Its most commonly-used variant, FocusedILS, aggressively limits computational resource usage by capping algorithm runs based on the performance of the best incumbent design at any particular point during its sequential search process. It also ex-

[5]While model based algorithm configuration procedures can benefit from the availability of feature values for these instances, they are not strictly required by the general algorithm configuration problem.

ploits knowledge of parameter conditionality (i.e., parameters that only take effect if other parameters take a certain values) to improve performance when many conditional parameters corresponding to parameterized heuristics are present; currently-available implementations of ParamILS support conditionality in the form of a single logical conjunction (i.e., of the form $p_0 \mid p_1 \in D_1 \wedge \ldots \wedge p_k \in D_k$, for parameters p_i and domains D_i). ParamILS supports large, discrete parameter spaces (for example, we use it to configure a space containing $4.75 \cdot 10^{45}$ distinct configurations in Chapter 6); continuous-valued parameters must be discretized before it can be used. The ParamILS plugin for HAL implements the FocusedILS variant of ParamILS; see Chapter 5 or Appendix A for more details.

GGA [2] is a model-free configuration procedure based on a genetic algorithm. Like ParamILS, it is not limited to small parameter spaces; unlike ParamILS, it is able to configure both discrete- and continuous valued parameters directly. The implementation of GGA provided to us by its authors is able to exploit a limited form of parameter conditionality (in particular, of the form $p_0 \mid p_1 \in D_1$, for parameters p_i and domains D_i) and is inherently parallelized, requiring at least 8 simultaneous processes to operate to potential (personal communication, January 2010). GGA has been adapted into a plugin for HAL (see Chapter 5 for more details); unfortunately, due to licensing restrictions, no implementations of GGA are publicly available (including the HAL plugin).

SMBO [44] is a sequential model-based optimization framework that builds upon earlier work applying black-box function optimization techniques to the algorithm configuration problem (see, e.g., [50, 6, 43]). It has been used to implement two algorithm configuration procedures to date. ROAR is effectively a model-free implementation (it uses a constant, uniform model) that exploits a run-capping intensification strategy similar to that used by FocusedILS, and supports large numbers of real-valued and discrete parameters and complex parameter conditionality structures (in particular, disjunctions of conjunctions of the form supported by ParamILS). SMAC, which uses a response surface model to predict the performance of different parameter configurations, similarly supports both real-valued and categorical parameters, and in current development versions also supports aggressive run capping and conditional parameters. SMAC benefits from the availability of feature values for instances in $S_{\mathscr{D}}$, but it can also build its models based only on parameter values. The authors of SMBO have developed a HAL plugin for ROAR (see Chapter 5 or Appendix A), and are presently nearing completion of a corresponding plugin for SMAC.

Related Problems: Algorithm configuration is closely related to k-way comparison (and even more closely related to parameter response analysis; see Section 3.2.1.3) in that both problems involve analysing the performance of multiple algorithm designs. A key distinction is that the

focus of k-way comparison is on providing reliable comparative performance characterizations of all target algorithms, whereas the focus of algorithm configuration is on identifying a single, top-performing design. In practice this means that while k-way comparison procedures must distribute computational resources approximately equally between all target algorithms and problem instances, algorithm configuration procedures are able to more aggressively focus on "promising" algorithm designs and "discriminative" problem instances.

Algorithm configuration is also related to per-instance configuration (Section 3.2.2.3). Algorithm configuration works best on relatively homogeneous instance distributions; per-instance configuration improves performance for heterogeneous distributions. However, per-instance algorithm configuration *requires* instance feature data, whereas algorithm configuration does not. Current per-instance configuration procedures are also significantly (i.e., an order of magnitude) more computationally expensive than non-per-instance configuration procedures.

3.2.2.2 Per-instance Portfolio-Based Selection

Context: Consider the situation where multiple algorithms are available for a problem of interest, and none of them dominates any of the others for an instance distribution under consideration (in other words, each algorithm is state-of-the-art on some instances of interest). While a user could elect to use only the algorithm that performs best overall, this is clearly not an optimal strategy. The *per-instance portfolio-based selection problem* is the task of designing an algorithm that selects the best of these for any given problem instance; framed as a meta-algorithmic design problem, it is defined as:

Instance: $\langle A, S_{\mathscr{D}}, F, X, \delta, \mu \rangle$, where:

A is a set of parameterless target algorithms

$S_{\mathscr{D}}$ is a set of target problem instances drawn from some distribution $\mathscr{D}$

F is a set of target problem instance features

X is a set of feature extractors collectively able to extract values for all $f \in F$

δ is a performance evaluator

μ is an aggregator for multiple runs on single instances $\mathbf{x} \in S_{\mathscr{D}}$

Solution: $\langle a' \rangle$, where:

a' is a parameterless algorithm for the target problem that accepts any input instance $\mathbf{x}$ drawn from $\mathscr{D}$ and executes one $a \in A$ that optimizes performance on $\mathbf{x}$ according to δ and μ

Procedures: SATZILLA [64, 81] is a prominent application of algorithm design using per-instance portfolio-based selection. The originally-published version of SATZILLA learns a regression-based performance prediction model for each of the portfolio components (target algorithms

$a \in A$) based on values for features in F. When run on a new instance, the SATZILLA selector extracts feature values for that instance, predicts runtimes for each of the portfolio components, and executes the one with the lowest predicted runtime. Subsequent developments (in application to both SAT and MIP, as MIPZILLA [84]) improved performance through use of alternative classification-based selection models. In Chapter 6 we generalize and completely automate the procedure used to design SATZILLA, making it applicable to any problem domain for which training instances and feature extractors are available.

3S [59] is another successful application of algorithm selection techniques to designing SAT solvers. When building 3S, its designers first evaluated a number of portfolio components on a set of training instances for which features were available. When 3S is run on a new instance, it extracts instance features and identifies the k training instances most similar to the new instance in terms of these features. It then selects the portfolio component with the best performance aggregated over those k instances. While an automated design tool capable of building 3S-like selection designs is not publicly available, the procedure itself is simple and in principle applicable to arbitrary problem domains.

Related Problems: Closely related to per-instance portfolio-based selection is the *parallel portfolio scheduling* problem, which does not involve instance features and which requires solutions where a' executes multiple algorithms from $\mathbf{a}$ in parallel and returns the first solution found, allocating computational resources to optimize aggregate performance on $\mathscr{D}$ according to δ, μ, and some μ^2 for aggregation over multiple instances. For stochastic target algorithms with highly variable performance, parallel portfolio designs can outperform per-instance portfolio-based selection design: through parallelization of target algorithm runs, they increase the chance an algorithm run will "get lucky" and perform well. On the other hand, they risk being outperformed by a per-instance portfolio-based selector that chooses (only) the best algorithm. Unfortunately, no general-purpose automated procedures for the parallel portfolio scheduling problem are currently available.

The *per-instance parallel portfolio scheduling problem* goes a step further, combining portfolio-based selection and parallel portfolio scheduling to yield an a' that, for each input instance $\mathbf{x}$ (assumed drawn from $\mathscr{D}$), executes multiple algorithms from A in parallel and returns the first solution found, allocating computational resources to optimize the performance on $\mathbf{x}$ according to δ and μ. Per-instance parallel portfolio scheduling is strictly more powerful than either portfolio based selection or parallel portfolio scheduling in that either can be recovered as a special case; however, just as no automated parallel portfolio scheduling procedures are available, neither are any for its per-instance variant.

Finally, since it is straightforward to convert a set of parameterless algorithms into a param-

eterized algorithm with a single categorical parameter, per-instance portfolio-based selection can be seen as a special case of per-instance configuration (Section 3.2.2.3). In fact, all current per-instance configuration procedures ultimately return per-instance portfolio-based selection designs.

3.2.2.3 Per-instance Configuration

Context: Consider the case of a heterogeneous problem domain for which a single, highly parameterized solver is available. Algorithm configuration may not yield a strong design due to the heterogeneity of the domain, and since only one algorithm is available, it is not possible to use portfolio approaches directly. The *per-instance configuration problem* is a generalization of per-instance portfolio-based selection that is concerned with determining the optimal parameter configuration for an algorithm on a per-instance basis:

Instance: $\langle \mathscr{A}, S_{\mathscr{D}}, F, X, \delta, \mu \rangle$, where:

$\mathscr{A}$ is a parameterized target algorithm

$S_{\mathscr{D}}$ is a set of target problem instances drawn from some distribution $\mathscr{D}$

F is a set of target problem instance features

X is a set of feature extractors collectively able to extract values for all $f \in F$

δ is a performance evaluator

μ is an aggregator for multiple runs on single instances $\mathbf{x} \in S_{\mathscr{D}}$

Solution: $\langle a' \rangle$, where:

a' is a parameterless algorithm for the target problem that accepts any input instance $\mathbf{x}$ drawn from $\mathscr{D}$, and executes one $a^{\star}$ corresponding to an instantiation of $\mathscr{A}$'s parameters to values that optimize performance on $\mathbf{x}$ according to δ and μ

Procedures: All current per-instance configuration procedures operate by combining algorithm configuration procedures with per-instance portfolio-based selection procedures. Thus, they all eventually choose from a restricted set of configurations, so may ultimately be unable to produce the true optimal configuration when run on a new instance. However, especially for heterogeneous domains containing locally-homogeneous instance clusters, they can and do perform quite well in practice.

HYDRA [83, 84] is a per-instance configuration procedure that, in its initial implementation, iteratively used PARAMILS to design new parameterless algorithms for addition to a SATZILLA selection portfolio. In each iteration, HYDRA uses PARAMILS to find a design a^{+} corresponding to an instantiation of the parameters of $\mathscr{A}$ that optimizes mean-aggregated

performance on $S_{\mathscr{D}}$ according to a dynamic evaluator that lower-bounds the CPU time of a hypothetical portfolio using target algorithms from the previous iteration's portfolio plus a^+. In this way, the HYDRA procedure is able to achieve robust performance without any *a priori* knowledge about the instance distribution other than a suspicion of heterogeneity. In Chapter 6 we generalize the HYDRA procedure, parameterizing the choice of configuration and portfolio-based selection procedure, removing domain dependence, supporting additional performance metrics, and ultimately producing the first completely automated general-purpose per-instance algorithm configuration procedure.

ISAC [52] takes a different approach, first clustering training instances $S_{\mathscr{D}}$ into k partitions $S_1, \ldots, S_k$ based on domain-specific instance features, and then configuring $\mathscr{A}$ to optimize performance on each of these partitions and obtain a vector of parameterless algorithms for portfolio-based selection. When faced with a new instance, ISAC assigns the instance to an existing cluster based on extracted features, and then selects the associated configuration. CluPaTra [58] is similar to ISAC in that it operates by clustering problem instances, but unlike ISAC it operates on a novel class of features based on algorithm search trajectories that make it somewhat more domain-independent. Disadvantages of these approaches compared to HYDRA stem from the explicit clustering of instances according to feature values, and in particular include the assumed existence of a distance metric that accurately relates feature values to algorithm performance; this cannot be easily provided in a domain-independent way, and indeed, it is easy to construct examples with "noise" features that undermine such a metric. Finally, neither ISAC nor CluPaTra have publicly-available implementations, preventing third-party use.

Related Problems: Since it is straightforward to convert a set of parameterless algorithms into a parameterized algorithm with a single categorical parameter, per-instance configuration can be seen as a generalization of per-instance portfolio-based selection (Section 3.2.2.2). However, specialized per-instance portfolio-based selection procedures can be expected to outperform more general per-instance configuration procedures in cases where both are applicable — for example, selection procedures can feasibly evaluate each of the portfolio candidates, whereas per-instance configuration procedures cannot. As such, if there are only a limited number of parameterless algorithms of interest, per-instance portfolio-based algorithm selection remains the most appropriate meta-algorithmic design problem to solve.

If the instance distribution is relatively homogeneous in terms of performance as a function of parameter configuration, algorithm configuration (Section 3.2.2.1) is likely to yield comparable results to per-instance configuration at a fraction of the computational cost.

Chapter 4

The High-Performance Algorithm Laboratory

In the preceding chapters we have argued that a unified framework for meta-algorithmic analysis and design procedures is needed, and have presented a formal conceptualization of meta-algorithmic concepts that models the components and functionality necessary in such a framework, as well as the procedures it might support. In this chapter we more explicitly discuss the specific goals of our framework, the High-performance Algorithm Laboratory (HAL), and describe the high-level design choices that enable us to meet these goals. We then provide an overview of our current software implementation of the HAL framework in terms of its major architectural components. More detailed low-level documentation of the framework can be found in Appendix B.

4.1 Design Considerations

When designing HAL, we identified five broad functional requirements against which to judge our success. The first of these is that the design of HAL must be consistent and intuitive, and therefore easy to use. End users should be able to conduct useful meta-algorithmic experiments with a reasonable understanding of the procedures to be used, but almost no understanding of the system's implementation details. Moreover, the process for conducting an experiment should primarily depend on the meta-algorithmic problem being solved, not the idiosyncrasies of the specific procedure used to solve it. Just as importantly, meta-algorithmic developers should be able to implement new automated procedures for HAL using techniques similar to those they would otherwise have adopted, and should be able to exploit HAL's infrastructural features as transparently as possible. In short, any cognitive overhead associated with using HAL must be outweighed by the practical benefits provided by the framework.

Second, it is important that HAL promote best practices for reproducible research. This means that HAL should come with carefully designed procedures for solving the most common meta-algorithmic problems. It also means that these procedures should leverage robust, well-tested implementations of critical statistical procedures and visualizations, for example including statistical tests and machine learning models. Finally, it means that HAL should make it easy for experimenters and interested third parties to reproduce and thereby verify the results of all experiments conducted using the framework.

Third, it is important that HAL be broadly applicable while remaining specialized to meta-algorithmic work. In particular, this means that HAL and its meta-algorithmic procedures must be target problem independent, but that this independence does not impose undue restrictions on which meta-algorithmic problems can be supported, nor require that common meta-algorithmic concepts be reimplemented at each use. For example, meta-algorithmic problems such as per-instance portfolio-based selection should be supported in HAL by procedures that can assume the existence of instance features, but do not assume any associated domain-specific semantics. This requirement should ultimately help HAL reach the widest possible audience.

Fourth, it is important that HAL support large-scale empirical work, and that it make large experiments (almost) as easy to conduct as small ones. In particular, the system must provide infrastructural features for efficiently executing experiments in distributed computational environments as well as on single machines, and for managing the data associated with hundreds of large experiments. As argued in earlier chapters, the "user pain" associated with managing large computational experiments is a major factor in the desire for a unified empirical algorithmics framework, so it is critical that this pain be addressed by HAL.

Fifth, it is important that HAL be extensible. This means that third-party developers should be able to add new meta-algorithmic procedures to HAL, and that these procedures should be able to take advantage of all of HAL's infrastructural features. Moreover, it means that infrastructural features should themselves be extensible, and any new infrastructure support should (wherever possible) be automatically leveraged by existing meta-algorithmic procedures. This requirement anticipates application in operational environments that were not explicitly considered during initial framework design, and is critical for ensuring that HAL remains relevant as new empirical methods and supporting computational technologies are developed.

In an effort to meet all five of these design requirements, we have adopted a modular design that cleanly separates components for experiment modelling and meta-algorithmic logic from execution and data management infrastructure. Design considerations for each of these aspects of the system are discussed in turn in the following subsections.

4.1.1 Components for Experiment Modelling and Meta-algorithmic Logic

We designed HAL to align with the conceptual formalization of Chapter 3, thereby providing a unified environment for the empirical analysis and design of high-performance algorithms via general meta-algorithmic techniques. In particular, this alignment strongly influenced the way that experiments are represented in terms of standardized meta-algorithmic components. Under the HAL application programming interface (API), a user models an experiment by declaring a meta-algorithm, an associated meta-problem instance, and any necessary settings, and later executes it by passing this model to the execution infrastructure.

This approach to experiment modelling leads directly to several desirable characteristics of HAL that contribute to meeting four of our design goals. First and most importantly, it means that HAL is able to use consistent and intuitive components to model any meta-algorithmic application scenario that our conceptual framework can model; and that it facilitates the use (and, indeed, implementation) of different meta-algorithmic procedures that solve the same meta-algorithmic problem. Second, it makes it easy to support packaging and distribution of complete experiments (including algorithms, instances, and other experiment settings) for independent verification, for example to accompany a publication: since all experiments are represented by standardized components, they can be explicitly serialized and deserialized using standard object serialization techniques. Third, it enables HAL to automatically archive experimental data in an application-agnostic way (by simply archiving the serialized components), and to serve as a central repository for components like target algorithms and instance distributions. Finally, it ensures that HAL can work with arbitrary target problems and algorithms, and arbitrary meta-algorithmic design and analysis techniques, because all of the framework's functionality is implemented in terms of fully-generic components. This also simplifies the construction of complex experiments consisting of sequences of distinct design and analysis phases, because the algorithmic outputs of any design procedure are guaranteed to be compatible (in a low-level implementation sense) with procedures for solving applicable analysis problems.

To meet our extensibility goals and to support a wide range of meta-algorithmic design and analysis procedures, HAL allows developers to contribute self-contained plugin modules relating to specific meta-algorithmic problems and their associated procedures. A plugin might provide a new procedure for a relatively well-studied problem, such as algorithm configuration. Alternately, it might address new problems, such as scaling analysis or parameter response analysis, and provide procedures for solving them drawing on concepts such as solution cost and quality distributions or response surface models. In the long run, the value of HAL to end users will largely derive from the availability of a library of plugins corresponding to cutting-edge meta-algorithmic procedures. Thus, HAL is an open platform, and we encourage members of the community to contribute new

procedures.

To facilitate this collaborative approach, HAL is designed to ensure that the features offered to end users are mirrored by benefits to developers. Perhaps most importantly, the separation of experiment design from runtime infrastructure means that the execution and data management features of HAL are automatically provided to all meta-algorithmic procedures that implement the HAL API; these infrastructural features are discussed in the next subsection. Since the HAL API standardizes all fundamental meta-algorithmic components required when building a meta-algorithm (for example, target algorithms, parameter settings/spaces, problem instances, and features), it streamlines the development process for new meta-algorithmic procedures. Adoption of this API also simplifies the design of hybrid or higher-order procedures. For example, all existing per-instance configuration procedures work by solving algorithm configuration and per-instance portfolio-based selection subproblems (see, e.g., [83, 52, 58]); as demonstrated in Chapter 6, implementation using HAL allows the underlying configuration and selection sub-procedures to be easily replaced or interchanged.

Finally, as our group continues to use the HAL API, we have begun to implement additional functionality useful for developing increasingly sophisticated design and analysis procedures. While this remains an active line of development, we ultimately expect to assemble libraries that include standardized components for empirical analysis (e.g., hypothesis testing and data visualization; see Appendix B for an explicit example), machine learning (e.g., feature extraction and regression/classification), and design space exploration (e.g., local search and continuous optimization), adapted specifically for the instance and runtime data common in algorithm analysis and design scenarios and using proven techniques from the literature. We hope that the availability of libraries such as these will contribute to the correctness of the meta-algorithmic procedures available in HAL, as well as further streamline the task of implementing them.

4.1.2 Execution and Data Management Infrastructure

While the formalization of meta-algorithmic concepts presented in Chapter 3 led fairly directly to a principled design for the experiment modelling components of HAL, additional requirements analysis was required to design the execution and data management infrastructure. In particular, achieving a balance between extensibility and ease-of-use was a significant challenge.

Software frameworks typically adopt an inversion of control in which developers implement components specified by an API that are then invoked by control logic built into the framework itself (see, e.g., [70]). This control inversion promotes highly decoupled software systems; the developer needs to know only a few details of the framework's implementation in order to exploit the functionality it provides. This development paradigm can be contrasted with the more familiar style of procedural programming using a software library, in which the developer assembles custom

control logic using carefully-selected library components; in this case, the developer must fully understand how to combine library components in order to achieve the desired behaviour.

In order to support independent extensibility of both meta-algorithmic procedures and execution and data management infrastructure components, it is critical that meta-algorithms in HAL make very few assumptions about the infrastructure they operate upon. This means that it is not feasible to have meta-algorithm developers instantiate execution and data management components directly. As such, in early prototypes of HAL we employed a standard control inversion strategy, separating instantiation of execution and data management components from the meta-algorithm's implementation, and having the framework query standardized meta-algorithm API methods whenever resources were available to execute new target algorithm runs.

However, we soon found that yielding control to the framework was too limiting when implementing automated procedures for multiple meta-algorithmic problems. For example, the logic for a highly parallelizable procedure for single-algorithm analysis is quite different from that of a serial design procedure such as PARAMILS, which in turn is different from a hybrid procedure such as HYDRA (see Chapter 3 for descriptions of these procedures); our attempts to implement all three while constrained to a framework-determined control flow led to complex designs that we found highly unintuitive. Instead, we ultimately settled on an intermediate design that combines the loose dependency coupling of a traditional framework with support for intuitive and flexible procedural meta-algorithmic logic.

Meta-algorithms in HAL execute algorithms using a special *runner* object provided by the framework at runtime. From the perspective of the meta-algorithm developer, the runner is quite simple: it provides a method that takes a run specification as input, and produces a handle to the corresponding algorithm run as output. Thus, meta-algorithms can simply use this runner whenever a target algorithm (feature extractor, etc.) needs to be executed, and otherwise they are coded much as they would be in the absence of HAL.

In fact, the runner object is rather more complex than one might expect from its simple interface. Depending on the framework's configuration, the runner can fulfil run requests in a variety of execution environments, for example on the local machine or on a compute cluster. These environments can impose different constraints on execution; for example, a single-core machine might be restricted to one target algorithm run at a time, while a compute cluster of multi-core nodes might be optimized for batches of parallel runs. As a result of these varying constraints, although the runner always returns a handle to a run immediately, the run itself only starts when the execution environment has sufficient free resources to begin computation. In this way, the runner enforces a weak coupling between the procedural logic of the meta-algorithm and the inverted control of the execution environment.

Since algorithm runs are not necessarily started when the runner returns, and since they in general take time to complete once started, a mechanism for status update notification is required to coordinate large numbers of parallel target runs. In HAL, custom *visitor* code can be registered to execute whenever some *event* occurs; events include status updates (for example, from unstarted to started, or started to finished) and output production (for example, output of a new candidate design). This event-driven design allows highly parallel meta-algorithms to be implemented without requiring the developer to explicitly write complex multithreaded code. It also provides the mechanism by which HAL's data management components automatically archive experimental results.

Data management infrastructure in HAL is also abstracted via the runner object. Whenever a run request is made, the runner queries the appropriate data manager to determine whether a suitable run has already been completed; if so, a handle to that run is returned rather than executing a new run. Otherwise, the run is executed as before, but data management visitors are registered to update the data manager with the run's current status and outputs whenever the corresponding events occur. In this way, all runs performed by a meta-algorithm (and indeed, the meta-algorithm run itself) are automatically archived without any explicit configuration required from the meta-algorithm developer, and regardless of the execution environment being used.

4.2 The HAL 1.1 Software Environment

The remainder of this chapter introduces HAL 1.1, our implementation of HAL's core functionality. We implemented HAL 1.1 in Java, because the language is platform independent and widely used, its object orientation is appropriate for our modular design goals, it offers relatively high performance, and it boasts APIs for interfacing with a wide range of established third-party tools. Much of HAL's functionality is implemented through the use of such third-party APIs; where available, HAL 1.1 interfaces with R for statistical computing (otherwise, internal statistical routines are used), and with Gnuplot for data visualization. It similarly uses MySQL for data management where available (otherwise, an embedded SQLite database is used), and either Grid Engine or TORQUE for cluster computing. HAL 1.1 has been developed under Linux and Mac OS X, and supports most POSIX-compliant operating systems; the web-based UI can provide client access from any platform.

HAL 1.1 provides robust infrastructure for experiment modelling and execution management, and has been used to implement automated procedures for solving each of the meta-algorithmic problems defined in Chapter 3; these procedures are further described in Chapters 5 and 6, and in Appendix A. In the following subsections, we provide an overview of HAL 1.1's core architecture in terms of the three major subsystems anticipated in Section 4.1 and illustrated in Figure 4.1: the experiment modelling subsystem, the execution and data management subsystem, and the user

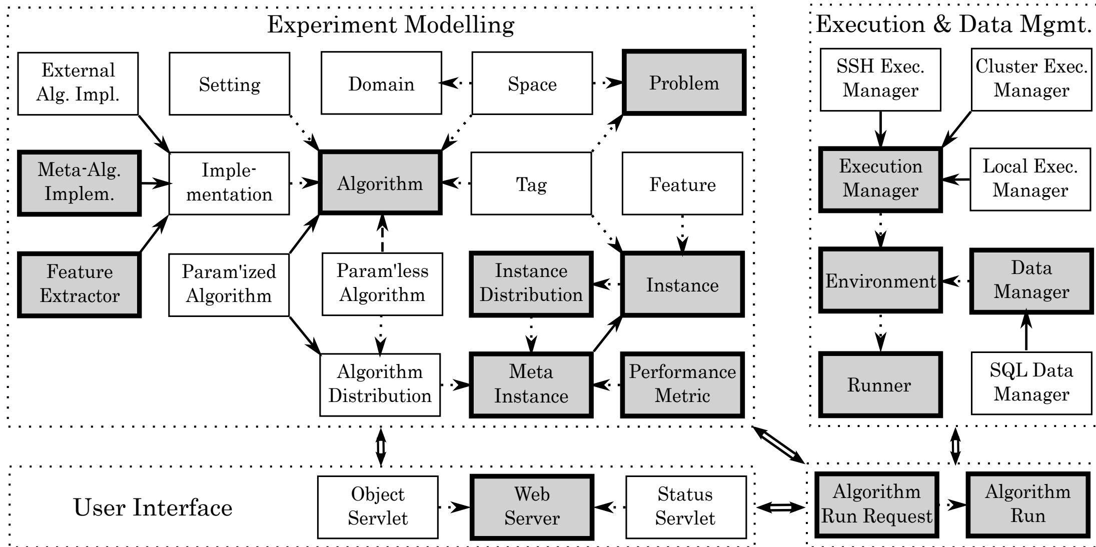

Figure 4.1: Schematic architectural overview of HAL. Dashed arrows indicate composition; solid arrows, inheritance. Key components are shaded. Note the distinct subsystems, with interactions between them (double arrows) typically moderated by *AlgorithmRuns/Requests*. For clarity, some components and relationships are not indicated; more complete class diagrams are available in Appendix B.

interface subsystem.

4.2.1 Experiment Modelling Subsystem

As discussed in Section 4.1.1, the components of the *experiment modelling subsystem* largely correspond to the fundamental algorithmic concepts defined in Chapter 4. This subsystem includes most of the classes exposed to meta-algorithm developers using the HAL API, including those that are extensible via plugins.

We will consider the running example of a user conducting an experiment with HAL 1.1, which allows us to describe the classes in each subsystem. The user's first step is to select a meta-algorithmic *Problem* to solve. (We indicate Java classes by capitalizing and italicizing their names.) Once a problem is selected, the user must select or import an *InstanceDistribution* containing target *ProblemInstances* of interest. HAL 1.1 currently supports finite instance lists, but has been designed to allow other kinds of instance distributions, such as instance generators. A *ProblemInstance* provides access to problem-specific instance data, as well as to *Feature* values and arbitrary user-defined tag *Strings*. In particular, an *ExternalProblemInstance* for a target problem includes a reference to the underlying instance file; a *MetaProblemInstance* contains target *Algorithms*, a target *InstanceDistribution*, a *PerformanceMetric*, and where applicable (e.g., for per-instance portfolio-

based selection), *Features* and corresponding *FeatureExtractors*.

The next step in experiment specification is to choose one or more target algorithms. In HAL 1.1, the *Algorithm* class encodes a description of the inputs and outputs of a particular *AlgorithmImplementation*. Two *Algorithm* subclasses exist: a *ParameterizedAlgorithm* includes configurable parameters in its input space, whereas a *ParameterlessAlgorithm* does not. Input parameters and output values are described by named *Domains* contained in *ParameterSpaces*; HAL 1.1 supports a variety of *Domains* including Boolean-, integer-, and real-valued *NumericalDomains*, and arbitrary *CategoricalDomains*. For a user-provided target algorithm, an *ExternalAlgorithmImplementation* specifies how the underlying executable is invoked and how its outputs should be interpreted; for meta-algorithms, a *MetaAlgorithmImplementation* implements all relevant meta-algorithmic logic.

Some meta-algorithmic procedures require *Features* and *FeatureExtractors*. A *Feature* simply defines a name and a *Domain* of possible values; a *FeatureExtractor* is nothing more than an *AlgorithmImplementation* that has outputs corresponding to named *Features*. Any *Algorithm* can be used to create a *WrappedFeatureExtractor*, provided its output *ParameterSpace* defines *Domains* that are compatible with those of the associated *Features*.

The final component needed to model a meta-algorithmic experiment is a performance metric. A *PerformanceMetric* in HAL 1.1 is capable of performing two basic actions: first, it can evaluate an *AlgorithmRun* with an *Evaluator* to produce a single real value; second, it can aggregate a collection of such values (for example, over problem instances, or over separate runs of a randomized algorithm) with an *Aggregator* into a single final score. HAL 1.1 includes implementations for commonly-used performance metrics including median, average, penalized average runtime (PAR), and average solution quality; it is straightforward to add others as required.

4.2.2 Execution and Data Management Subsystem

The *execution and data management subsystem* implements functionality for conducting experiments specified by the user (including all meta-algorithm and target algorithm runs); in HAL 1.1, it supports execution on the local system (i.e., the system executing the parent meta-algorithm; see below), on a remote system, or on a compute cluster. It also implements functionality for cataloguing individual resources (such as target algorithms or instance distributions) and for archiving and retrieving the results of runs from a database.

Before execution, an *Algorithm* must be associated with a compatible *ProblemInstance* as well as with *ParameterSettings* mapping any other input variables to specific values. More precisely, an *AlgorithmRunRequest* is composed of an *AlgorithmImplementation* indicating the algorithm to run, *ParameterSettings* and a *ProblemInstance* specifying all inputs, and a *ParameterSpace* specifying the outputs to be recorded. This *AlgorithmRunRequest* is ultimately fulfilled by the execution and

data management subsystem to yield an *AlgorithmRun*.

Once our user has completely specified an experiment, he must define the environment in which execution is to occur. An *Environment* in HAL 1.1 is defined by *ExecutionManagers*, which are responsible for starting and monitoring computation, and a *DataManager*, which is responsible for performing archival functions; access to an *Environment* is provided to meta-algorithms via a *Runner*. When an *AlgorithmRunRequest* is passed to a *Runner*, the associated *Environment* queries the *DataManager* to see if a satisfying *AlgorithmRun* has already been completed. If so, it is fetched and returned; if not, the request is passed to an *ExecutionManager* for computation. In either case, results are returned as an *AlgorithmRun* object which allows monitoring of the run's elapsed CPU time, status, and output value trajectories in real time during execution and after completion. It also provides functionality for early termination of runs and uses this to enforce any requested CPU time or run length limits.

HAL 1.1 includes four *ExecutionManager* implementations. The *LocalExecutionManager* performs runs using the same machine that runs the current instance of HAL 1.1, and the *SSHExecutionManager* performs runs on remote machines using a secure shell connection. Two *ClusterExecutionManager* implementations, *SGEClusterExecutionManager* and *TorqueClusterExecutionManager*, distribute algorithm runs across nodes of compute clusters managed by Grid Engine and TORQUE respectively. An *Environment* can be configured to use different *ExecutionManagers* in different situations. For example, for analysis of an algorithm on target problems that require a long time to solve, the user might specify an *Environment* in which the parent meta-algorithm is executed on a specific remote host, and target algorithm runs are distributed on a cluster. Alternatively, when target algorithm runs are relatively short, the user might specify an *Environment* in which all execution happens locally.

HAL 1.1 includes two *DataManager* implementations. By default, a subclass employing an embedded SQLite database is used. However, due to limitations of SQLite in concurrent applications, a MySQL-backed implementation is also provided and is recommended for all but the most simple tasks. These *SQLDataManagers* use a common SQL schema based on the same set of fundamental meta-algorithmic concepts to store not only experimental results, but also information sufficient to reconstruct all HAL objects used in the context of a computational experiment. We note that external problem instances and algorithms are not directly stored in the database, but instead at recorded locations on the file system, along with integrity-verifying checksums. This eliminates the need to copy potentially large data files for every run, but presently assumes that all compute nodes have access to a shared file system, or at least that all file dependencies are present at the same location in each machine's local file system.

4.2.3 User Interface Subsystem

The *user interface subsystem* provides a remotely-accessible web interface to HAL 1.1, via an integrated *WebServer*. Many classes have associated *ObjectServlets* in the *WebServer*, which provide interface elements for their instantiation and modification. The *ObjectServlets* corresponding to *Problems* are used to design and execute experiments; the servlet for a given *Problem* automatically makes available all applicable meta-algorithmic procedures. Additional *ObjectServlets* allow the user to specify and examine objects such as *Algorithms*, *InstanceDistributions*, *ParameterSettings*, and *Environments*. A *StatusServlet* allows the user to monitor the progress and outputs of experiments both during and after execution, by inspecting the associated *AlgorithmRun* objects. Finally, the interface allows the user to browse and maintain all objects previously defined in HAL, as well as to export these objects for subsequent import into a separate HAL database (e.g., for independent validation of results).

Chapter 5

Algorithm Analysis & Design with HAL

We now demonstrate HAL in action. Specifically, we walk through two case studies that illustrate application scenarios that might arise for a typical user. In this way, we also present several of the meta-algorithmic procedures that are currently available for HAL. The results of these case studies are summarized in Table 5.1 and in the following figures (generated by the respective procedures in HAL). Exported experiment designs are available online [63] to facilitate independent validation of our findings.

5.1 Experimental Setup

All case studies were conducted using a late beta version of HAL 1.0 and its plugins. Algorithm runs were executed on a Grid Engine cluster of 55 identical dual-processor Intel Xeon 3.2GHz nodes with 2MB cache and 4GB RAM running openSUSE Linux 11.1. Reported runtimes indicate total CPU time used, as measured by HAL. Data management was performed using a MySQL database running on a dedicated server.

The case studies of this chapter involves two analysis plugins for HAL, which provide procedures for the single-algorithm analysis and pairwise comparison problems. They also involve three design plugins for HAL, each of which provides a procedure for the algorithm configuration problem. We briefly describe each of these procedures below; Appendix A contains more detailed descriptions of these and other meta-algorithmic procedures available for HAL.

5.1.1 Analysis Procedures

Meta-algorithmic analysis procedures for HAL require as input a meta-problem instance containing some number of parameterless target algorithms that depends on the problem being solved, as well

as a set of benchmark instances and a performance metric. They also accept optional procedure-specific settings (notably including a random seed, a maximum number of runs per target instance, a maximum CPU time per target run, a maximum number of total target runs, and a maximum overall CPU time budget).

5.1.1.1 Single-Algorithm Analysis Procedure: SCD-Based Analysis

This comprehensive approach to single-algorithm analysis operates on a single parameterless target algorithm. It collects runtime data for this target algorithm on the provided instance distribution (in parallel, if supported by the execution environment used) until all required runs have been completed or the CPU time budget has been exhausted. Runs are evaluated using the provided performance metric, and the median performance on each instance is computed and used to construct an empirical SCD (see Chapter 2). Summary statistics are then computed for this SCD, and a plot of its empirical CDF is produced. Finally, the aggregation defined by the supplied metric is computed over the SCD in order to yield a single performance score to the algorithm in question.

5.1.1.2 Pairwise Comparison Procedure: Comprehensive Pairwise Comparison

The Comprehensive Pairwise Comparison procedure takes essentially the same input as the SCD-based Analysis procedure, except that it requires two parameterless target algorithms instead of one. In addition to performing an SCD-based analysis of each of the two given algorithms and overlaying the resultant SCD plots, the procedure generates a scatter plot illustrating paired performance across the benchmark set, and performs Wilcoxon signed-rank and Spearman rank correlation tests. The Wilcoxon signed-rank test determines whether the median of the paired performance differences between the two algorithms across the instance set is significantly different from zero; if so, the procedure identifies the better-performing algorithm. The Spearman rank correlation test determines whether a significant monotonic performance correlation exists between the two algorithms.

5.1.1.3 *k*-way Comparison Procedure: Comprehensive *k*-way Comparison.

Note: the k-way Comparison Procedure was not available when the experiments of this chapter were conducted. However, for completeness it is described here. For an example of its use, see Chapter 6.

This procedure has similar inputs to the Pairwise Comparison Procedure, which it generalizes from two to k parameterless target algorithms. The procedure performs SCD-Based Analysis for each of the k provided algorithms. It then performs a Friedman test to determine whether any of the k algorithms consistently performs better or worse than the others; if so, it performs *post-hoc* Wilcoxon

signed-rank tests between the best-performing algorithm and the remaining algorithms, and identifies all algorithms that do not exhibit statistically significantly worse performance. It also produces an overlay of the individual algorithm SCDs.

5.1.2 Algorithm Configuration Procedures

Algorithm configuration procedures for HAL require as input a single parameterized algorithm, a training problem instance set, and a performance metric; they also accept optional procedure-specific settings including per-target-run and overall CPU time budgets, and a random seed. They output a single parameterless algorithm corresponding to the best configuration found, an estimate of the quality of that design according to the user-provided performance metric and based on the runs performed during configuration, and various other procedure-specific diagnostic outputs; outputs are updated as the run progresses.

5.1.2.1 Algorithm Configuration Procedure: PARAMILS

HAL supports the FOCUSEDILS variant of the local-search-based PARAMILS configurator [40]. The plugin augments the original Ruby implementation of PARAMILS with an adapter class that integrates with the HAL framework. Unlike the other configurator plugins, PARAMILS requires sets of discrete values for all target algorithm parameters; therefore, the adaptor class uniformly[1] discretizes any continuous parameter domains (the granularity of this discretization is one of the optional inputs to the procedure).

5.1.2.2 Algorithm Configuration Procedure: GGA

HAL supports a plugin that interfaces with the original binary implementation of GGA [2]. Many of its optional settings control parameters of the genetic algorithm it uses to explore the space of parameter configurations. Per the recommendations of GGA's authors, the procedure always executes 8 simultaneous target algorithm runs on the local machine, and disallows performance metrics other than PAR-1 (personal communication, January 2010). Unfortunately, sources for this procedure are not available, and because of copyright restrictions we are unable to further distribute the executable supplied to us by its authors.

[1] If the user indicates that a continuous parameter domain should be explored on a logarithmic or exponential scale, the discretization is uniform in the corresponding transformed space.

	Training Set					Test Set				
Algorithm	q25	q50	q75	mean	stddev	q25	q50	q75	mean	stddev
CPLEX						26.87	109.93	360.59	9349.1	24148.9
GUROBI						13.45	71.87	244.81	1728.8	9746.0
SPEAR default						0.13	0.80	10.78	6.78	10.62
SPEAR modified						0.19	0.89	4.35	3.40	6.31
SPEAR PARAMILS	0.22	0.80	2.63	1.72	2.54	0.19	0.80	2.21	1.56	2.22
SPEAR GGA	0.22	0.90	1.96	2.00	3.36	0.16	0.90	1.72	1.72	3.39
SPEAR ROAR	0.22	0.92	2.70	1.91	2.59	0.19	0.91	2.41	1.82	2.98

Table 5.1: Summary of case study results. Reported statistics are in terms of PAR-1 for SPEAR and PAR-10 for MIP solvers; units are CPU seconds. Only the best design in terms of training set performance is reported for each configuration procedure.

5.1.2.3 Algorithm Configuration Procedure: ROAR

HAL supports the Random Online Aggressive Racing (ROAR) procedure, a simple yet powerful model-free implementation of the general Sequential Model-Based Optimization (SMBO) framework [44]. The ROAR plugin was implemented entirely in Java using the HAL API.

5.2 Case Study: Choosing a MIP Solver

In this case study, a user wants to choose between two commercial mixed-integer program (MIP) solvers, IBM ILOG CPLEX 12.1 [45] and Gurobi 3.01 [29], using the 55-instance mixed integer linear programming (MILP) benchmark suite constructed by Hans Mittelmann for the purpose of evaluating MIP solvers [61]. For all experiments the user sets a per-target-run cutoff of $\kappa = 2$ CPU hours, and uses penalized average runtime (PAR-10) as the performance metric (PAR-k counts unsuccessful runs at k times the cutoff; see Chapter 3). As the per-run cutoff is relatively long, he uses a HAL execution environment configured to distribute target runs across cluster nodes.

5.2.1 Single-Algorithm Analysis

CPLEX is the most prominent mixed-integer programming solver. In this experiment, the user is interested in characterizing its out-of-the-box performance on a set of realistic problem instances, and in particular is curious whether there is evidence to suggest that better performance might be obtained by choosing an alternative solver. To answer these questions, the user measures the performance of CPLEX on the MILP instance set using HAL's SCD-Based Analysis procedure; as CPLEX is deterministic, it is run only once per instance. The resulting summary statistics are shown in Table 5.1; from the table, we see that the mean (PAR-10) performance score is substantially higher than the median score. This suggests that CPLEX failed to solve several difficult outlier instances and was therefore penalized by the PAR-10 evaluator; indeed, the corresponding SCD in the left

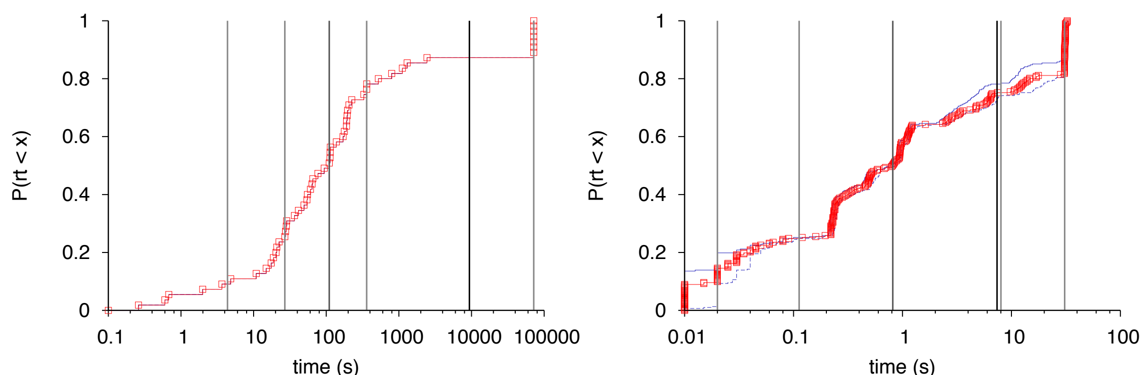

Figure 5.1: Single-algorithm analysis of CPLEX on the MILP benchmark set (left; units are PAR-10 in CPU seconds), and SPEAR on the SWV test set (right; units are PAR-1 in CPU seconds). Vertical grey lines indicate the $10^{\text{th}}, 25^{\text{th}}, 50^{\text{th}}, 75^{\text{th}}$, and 90th percentiles of the respective SCD; the vertical black line indicates the mean. Since SPEAR is randomized, the main (red) trace indicates the SCD obtained by aggregating over 20 independent runs on each instance using the median; the secondary (blue) traces indicate aggregation using the 25th and 75th quartiles.

pane of Figure 5.1 clearly shows that CPLEX timed out on a number of the benchmark instances.

5.2.2 Pairwise Comparison

Having demonstrated room for improvement over CPLEX, the user decides to investigate Gurobi, a relatively recent commercial MIP solver. In this experiment, the user is interested in determining whether Gurobi consistently outperforms CPLEX on the MILP instance set, and in particular whether it is able to solve more instances in the allotted time. He uses the Comprehensive Pairwise Comparison procedure for this task; HAL is able to automatically and transparently reuse the CPLEX runtime data collected in Section 5.2.1 for this follow-up experiment. Statistics of the performance of the two solvers are shown in Table 5.1. As can be seen from Figure 5.2, which presents the combined SCD plot and the performance correlation plot, Gurobi outperformed CPLEX on most instances; in particular, it successfully solved all of the instances that CPLEX could not, although for several other instances CPLEX remained the fastest solver, including one that Gurobi was unable to solve. The Wilcoxon signed-rank test indicates that the observed pairwise performance difference in favour of Gurobi is significant at $\alpha = 0.05$ ($p = 0.024$). These results are consistent with Mittelmann's independent observations using the MILP benchmark set [61]. Finally, a Spearman coefficient of $\rho = 0.86$ reflects the strong correlation seen in the scatter plot.

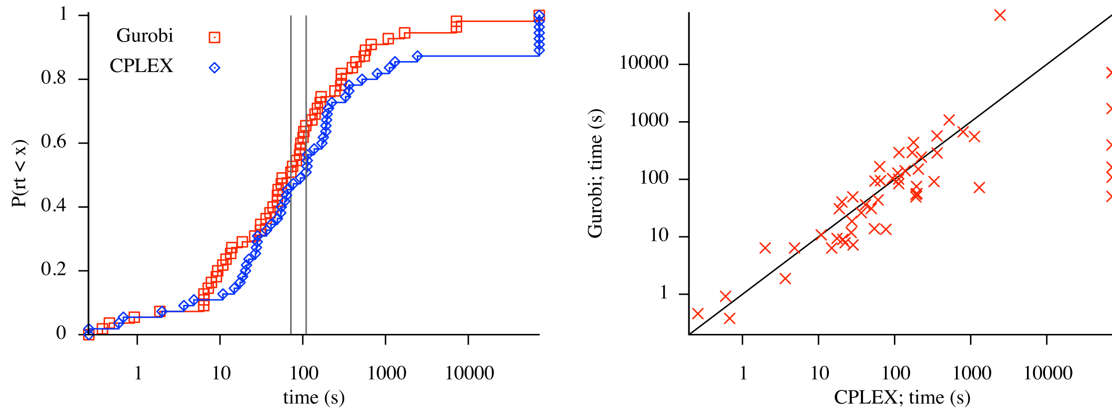

Figure 5.2: Pairwise comparison of CPLEX and Gurobi on the MILP benchmark set. In the SCD, median runtimes are indicated by vertical lines.

5.3 Case Study: Adapting a SAT Solver

In this case study, a user aims to adapt a stochastic tree search solver for SAT, version 1.2.1 of SPEAR [3], to achieve strong performance on the 302-instance industrial software verification (SWV) benchmark training and test sets used by Hutter et al. [38]. The user sets a per-target-run cutoff of 30 CPU seconds and evaluates performance by mean runtime (PAR-1). Due to this relatively short per-run cutoff and the serial nature of many of HAL's algorithm configuration procedures, he configures a HAL execution environment to perform all target runs for a given experiment on the same cluster node that the corresponding configuration procedure is run on, rather than distributing them independently across the cluster.

5.3.1 Single-Algorithm Analysis

SPEAR was originally optimized for solving SAT instances from several applications, but was later prominently used for software verification in particular (see, e.g., [38, 3]). In this phase of the case study, the user assesses the original, manually optimized version of SPEAR on the SWV test set in order to establish a performance baseline for future optimization. The summary statistics from an SCD-based analysis (performing 20 runs per instance, as SPEAR is randomized, and using the median of these to construct an empirical SCD) are shown in Table 5.1 and the SCD in the right pane of Figure 5.1. The main observation to draw from these results is that mean (PAR-1) performance was substantially – over eight times – higher than the median performance. From the SCD we see that this is largely due to instances that the default configuration could not solve in the allotted time.

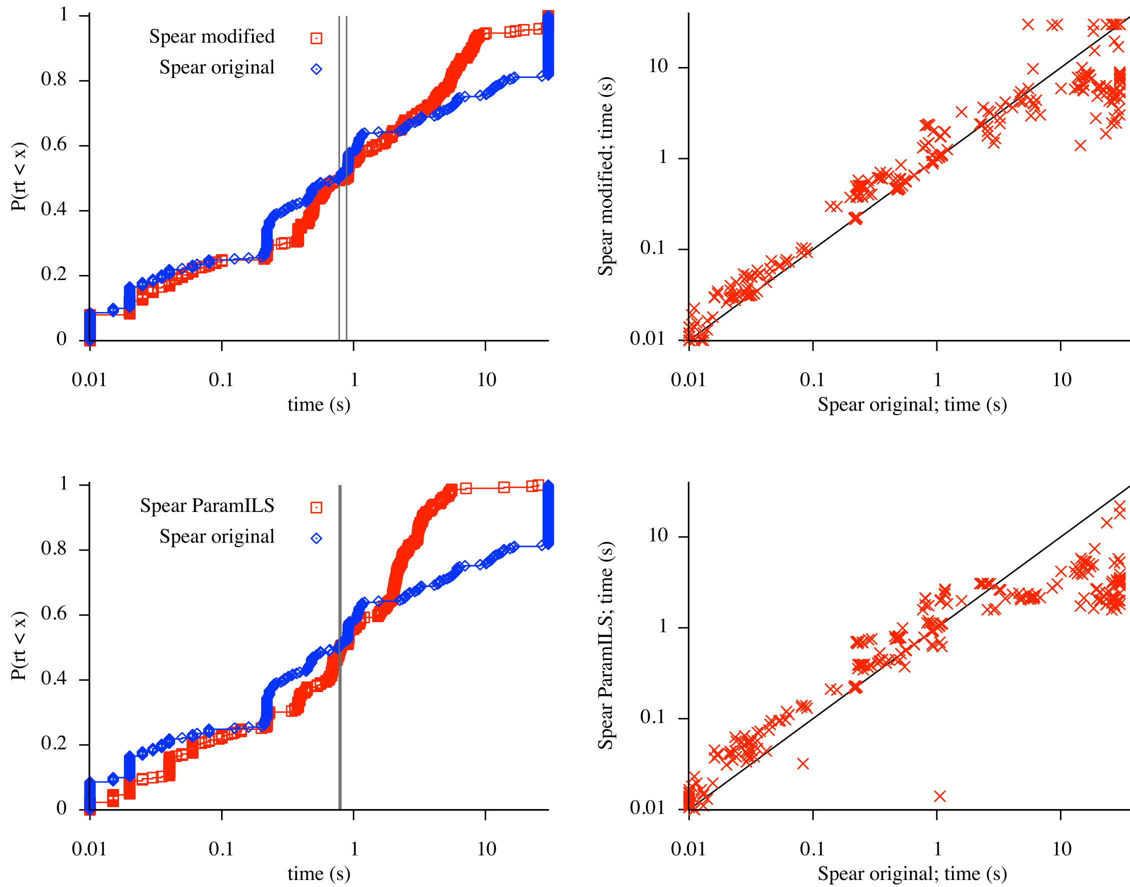

Figure 5.3: Pairwise comparisons of SPEAR designs on SWV test set. Top row, original vs. intuitively modified design; bottom row, original vs. best configured design (from PARAMILS). As SPEAR is randomized, per-instance performance is measured by the median CPU time of 20 independent runs.

5.3.2 Pairwise Comparison

When adapting an algorithm to a new class of benchmark instances, algorithm designers often apply intuition to making important design choices; these choices are often realized by setting parameters of the algorithm to certain values. For example, Hutter *et al.* [38] provide an intuitive explanation of the strong performance of one particular configuration of SPEAR in solving software verification instances:

> SWV instances prefer an activity-based heuristic that resolves ties by picking the variable with a larger product of occurrences [. . .] The SWV instances favored very aggressive restarts (first after only 50 conflicts) [. . .] A simple phase selection heuristic

> (always assign FALSE first) seems to work well for SWV [...] SWV instances prefer no randomness [...]

In an attempt to improve the performance of SPEAR on the instances that the default configuration was unable to solve, our user follows this qualitative description to manually obtain a new parameter configuration; specifically, the following parameters were changed:

- "sp-var-dec-heur" was changed from 0 to 6
- "sp-first-restart" was changed from 100 to 50
- "sp-phase-dec-heur" was changed from 5 to 0
- "sp-rand-phase-dec-freq" and "sp-rand-var-dec-freq" were changed from 10^{-3} to 10^{-6}

The user compares this modified configuration against the default on the SWV test set using the Comprehensive Pairwise Comparison procedure (again using the median of 20 runs per instance). Results are shown in Figure 5.3 and Table 5.1. As hoped, the modified configuration solved most (but not quite all) of the test instances, and ultimately achieved better performance than the default in terms of PAR-1. As clearly seen from the SCDs and from the scatter plot, this was accomplished by sacrificing performance on easy instances for gains on hard instances; as a result, median performance was slightly worse than with the default configuration. The Wilcoxon signed-rank test indicates that the median paired performance difference over the full benchmark set is *not* significantly different from zero at $\alpha = 0.05$ ($p = 0.35$). Inter-instance correlation between the two configurations is strong, with a Spearman coefficient of $\rho = 0.97$.

5.3.3 Algorithm Configuration

In Section 5.3.2, the user observed that SPEAR's performance can be improved by manually modifying its parameters. Seeking further performance gains, he turns to automatic algorithm configuration. Because configuration procedures are easily interchangeable in HAL, our user runs all three of them. He performs 10 independent runs of each configuration procedure on the SWV training set, and sets a time budget of 3 CPU days for each run.

The user validates the performance of each of the 40 final designs on the training set using the SCD-based Analysis procedure procedure[2], taking the median of 20 runs per instance. He then compares the design with the best training performance (in terms of PAR-1) found by each of the procedures against the default configuration using the Comprehensive Pairwise Comparison

[2] At the time these experiments were conducted, no *k*-way comparison procedure was available in HAL; otherwise it would have been used here.

procedure on the test set, again performing 20 runs per instance. Results are shown in Figure 5.3 and Table 5.1. The best design found by each configurator was substantially better than both the default and the intuitively-modified configuration in terms of PAR-1, with PARAMILS producing slightly better results than GGA, and with GGA in turn slightly better than ROAR. Indeed, as seen in the scatter plot, the PARAMILS design successfully solved all test instances in the allotted time, and managed to do so without affecting median performance. In all cases, pairwise performance differences with respect to the default are significant at $\alpha = 0.05$ according to the Wilcoxon signed rank test ($p = \{7.8, 0.002, 9.7\} \times 10^{-3}$ for PARAMILS, GGA, and ROAR respectively).

Finally, the user compares the best designs found by each of the three configuration procedures against each other using the Comprehensive Pairwise Comparison procedure, in order to better understand their relative strengths and weaknesses (we note that these experiments are not sufficient to make broader performance claims regarding the three configuration procedures in general). The plots resulting from these experiments are shown in Figure 5.4. Comparing the PARAMILS and GGA configurations, we see that the GGA configuration is better on all but the hardest instances; indeed, the Wilcoxon signed rank test indicates that GGA outperforms PARAMILS on a statistically significant majority of the instances at $\alpha = 0.05$ ($p = 4.3 \times 10^{-18}$), despite the fact that it performs worse in terms of the chosen performance metric (PAR-1). Comparing ROAR and GGA yields similar conclusions – GGA statistically significantly outperforms ROAR according to the Wilcoxon test ($p = 2.0 \times 10^{-23}$), and this time also in terms of PAR-1. Finally, comparing PARAMILS and ROAR shows that the corresponding configurations are similar, with a slight but significant performance advantage for PARAMILS according to the Wilcoxon test ($p = 0.0238$). Overall, these results highlight the importance of carefully choosing a performance metric, and of careful interpretation of the results of statistical tests.

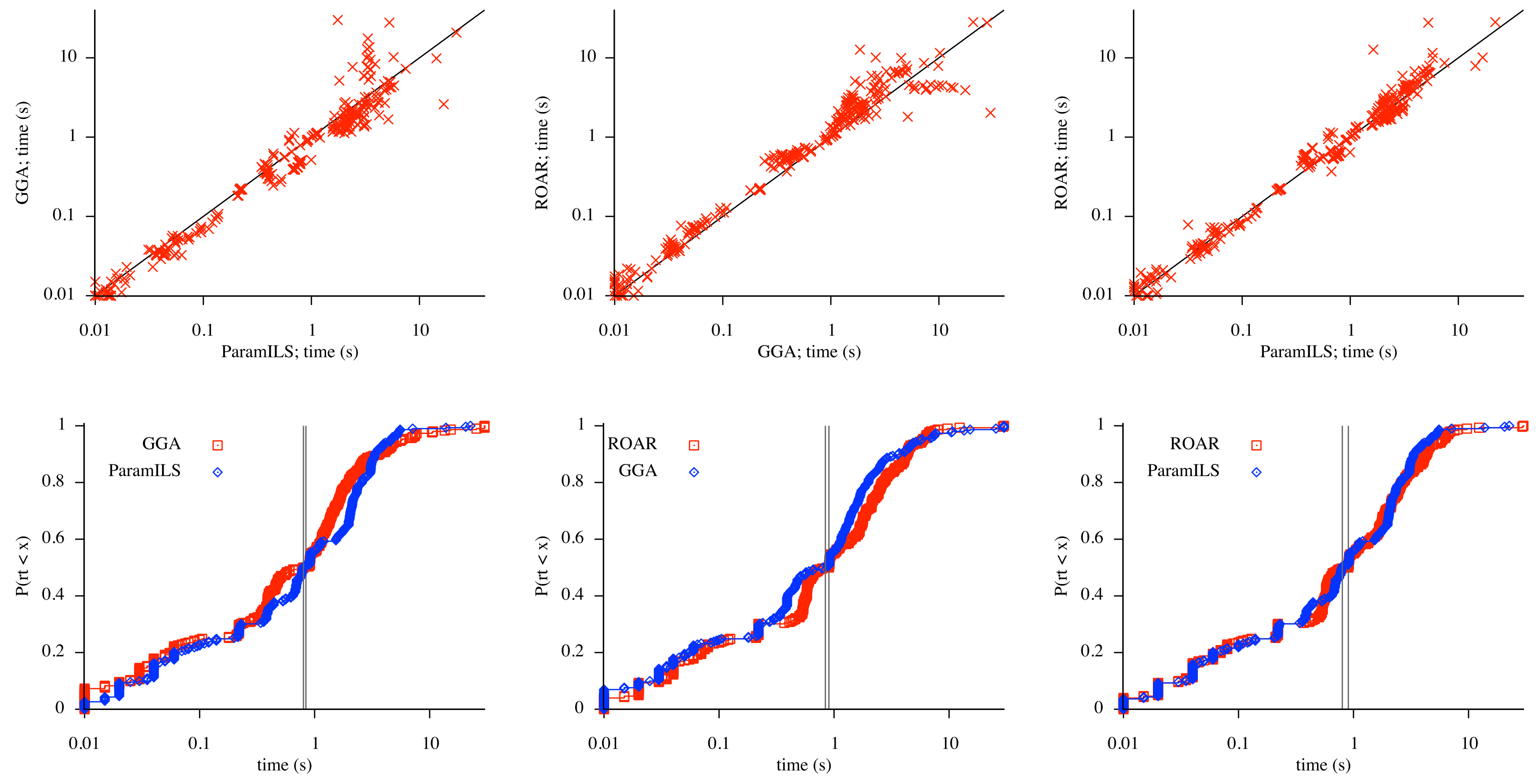

Figure 5.4: Comparison of PARAMILS, GGA, and ROAR designs for SPEAR on the SWV test set. As SPEAR is randomized, per-instance performance is measured by the median CPU time of 20 independent runs

Chapter 6

Developing Meta-algorithms with HAL

Having examined the use of existing HAL implementations of meta-algorithmic procedures for solving a variety of practical design and analysis problems in Chapter 5, we now consider the development of additional automated meta-algorithmic procedures with the HAL API. This chapter describes the design, implementation, and validation of two closely related procedures: the ∗ZILLA portfolio-based selection procedure, and the HYDRA-∗ per-instance configuration procedure; both are based on techniques used in the most recently published version of HYDRA [84] (there called HYDRA-MIP). In doing so, our goal is *not* to rigorously evaluate or improve the meta-algorithmic logic of the procedures being implemented, but rather to illustrate benefits of adopting the HAL API: in particular, abstraction of data collection and storage infrastructure, and incorporation of existing meta-algorithmic procedures as components of a larger design. These implementations constitute the first fully-automated, domain-independent procedures of which we are aware for solving either of the corresponding meta-algorithmic design problems.

In the interest of clarity and conciseness, we retain a degree of abstraction in this chapter. In particular, we describe the design of the ∗ZILLA and HYDRA-∗ procedures using the meta-algorithmic concepts and HAL components discussed in Chapters 3 and 4, but we do not exhaustively present all implementation details. A more explicit tutorial for meta-algorithm developers can be found in Appendix B, and sources for these and other plugins are available online [63].

6.1 Per-instance Portfolio-Based Selection with ∗ZILLA

In this section, we describe our adaptation of the MATLAB model-building code used by SATZILLA (or more accurately, by MIPZILLA) for operation in the HAL 1.1 framework. We use the HAL API to abstract domain dependencies and automate data collection from the core SATZILLA implementation, yielding the ∗ZILLA portfolio-based selection procedure. This demonstrates how the

HAL API can be used to extend the functionality of existing non-Java code; in the next section, we describe the development of a meta-algorithm in pure Java using other HAL procedures as components.

Although it includes optimizations to improve real-world performance and robustness, at its core ∗ZILLA is a straightforward application of machine learning techniques to the algorithm selection problem. The authors of SATZILLA discuss their multi-step process for solving the algorithm selection problem [81, 82]; these steps also describe ∗ZILLA:

1. Identify a set of target problem instances of interest.
2. Identify a set of target algorithms suspected to perform well on some target instances.
3. Identify a set of low-cost target problem instance features that are related to target algorithm performance, and associated extractors.
4. Run all feature extractors and target algorithms on each problem instance, evaluating each according to some performance metric.
5. Identify a list of one or more target algorithms to use as *pre-solvers* (step 10 below).
6. Identify a single target algorithm to use as a *backup solver* (steps 11, 12, and 14 below).
7. Learn a predictive model that identifies instances for which feature extraction is prohibitively costly, based on values for minimal-cost features and feature extractor performance data.
8. Learn a predictive model that ranks target algorithms for subsequent selection, based on instance features and target algorithm performance data.
9. Identify a subset of the target algorithms to include in the final algorithm portfolio.

In order to solve a new problem instance:

10. Run each pre-solver for a short duration, stopping if the instance is solved.
11. Extract minimal-cost features, and use them to query the feature extraction cost model. If feature extraction is predicted to be costly, run the backup solver; otherwise, continue.
12. Extract instance feature values. In case of failure, run the backup solver; otherwise, continue.
13. Query the selection model using these feature values to rank portfolio candidates.
14. Run the top-ranked target algorithm on the instance. If it fails, continue with the next-best-ranked candidate; if all candidates fail, run the backup solver.

In summary, given a per-instance portfolio-based selection instance (steps 1–3; see also Chapter 3.2.2.2), ∗ZILLA learns predictive models from algorithm performance data on a training instance set (steps 4–9), and then uses these models to perform algorithm selection in response to new problem instances (steps 10–14). Correspondingly, its implementation in HAL can be understood in terms of two related but distinct algorithms: a per-instance portfolio-based selection design procedure that implements the learning phase, and the output selector algorithm that uses learned models to solve the target problem. In the remainder of this section, we describe each of these algorithms as implemented in HAL.

6.1.1 The ∗ZILLA Selector Algorithm

Note: the discussion of this subsection refers to several predetermined components—in particular, a set of minimal-cost features, a binary classification model, a backup solver, a list of pre-solvers with associated time budgets, and a selection model. These components are provided to the ∗ZILLA *selector algorithm as parameters during instantiation; as such, specific choices for each are made by the* ∗ZILLA *design procedure described in Section 6.1.2 and are further discussed there.*

The ∗ZILLA selector performs algorithm selection based on values for a variety of instance features. While extracting these features for a particular instance is ideally much less costly than solving it with any portfolio candidate, in practice there can be instances for which this is not true, and for which it is better not to perform selection at all. In order to improve empirical performance in these cases, the selector algorithms produced by ∗ZILLA follow a three-stage design. In the first stage, values for *minimal-cost features*—features that can be extracted at effectively zero cost, as indicated by the user—are extracted from the input problem instance, and used to query a *binary classification model* that predicts whether it is better to revert to a fixed *backup solver* than to perform full selection. If so, the selector simply runs the backup solver and returns its result; otherwise, it proceeds to the second stage in which algorithms from a list of *pre-solvers* are run in sequence on the input instance for short durations, in order to quickly solve easy instances. If the instance remains unsolved after all pre-solvers have been run, the third stage involves extracting values for more costly features and using them to query a *selection model* that returns a ranking of portfolio components for the instance. These components are then run sequentially in ranked order until the one of them solves the instance or time runs out, at which point the output of the last run is returned; this means that lower-ranked algorithms are only run if all higher-ranked algorithms terminate unexpectedly. Pseudocode for this design is provided in Figure 6.1.

While the ∗ZILLA selector algorithm is not a meta-algorithm (it solves the target problem, which is typically not a meta-algorithmic problem), it nonetheless resembles a meta-algorithm in that it involves the execution of subsidiary algorithm runs: one or more feature extractors, and one or

Constructor: list of parameterless target algorithms $\mathbf{a}$;
list of target problem features $\mathbf{q}$;
list of feature extractors $\mathbf{f}$, that collectively extract values for all $q \in \mathbf{q}$;
indicator function λ that defines a sublist $\mathbf{q}'$ of minimal-cost features from $\mathbf{q}$;
binary classification model $M_{cls} : \mathbb{R}^{|\mathbf{q}'|} \to \{\text{true}, \text{false}\}$;
selection model $M_{sel} : \mathbb{R}^{|\mathbf{q}|} \to [1, |\mathbf{a}|]^m \subset \mathbb{N}^m$ for some m;
backup solver index $i_{bk} \in [1, |\mathbf{a}|] \subset \mathbb{N}$;
pre-solver indices and runtimes $\mathbf{S} = ((s_1, t_1), \ldots, (s_n, t_n)) \in ([1, |\mathbf{a}|] \times \mathbb{R})^n \subset (\mathbb{N} \times \mathbb{R})^n$ for some n
Input: target problem instance $\mathbf{x}$;
CPU time budget κ
Output: target problem solution $\mathbf{y}$

```
  let c = ⟨i_bk⟩;
1 let v' ∈ ℝ^|q'| = values for all q ∈ q' (i.e., where λ = 1) in order, extracted using f;
  if M_cls(v') = false then
      for i = 1, ..., |S| do
2         let y = a_{s_i}(x | min(t_i, κ − consumed CPU time));
          if y is a solution to x then
              return y;
3     let v ∈ ℝ^|q| = values for all q ∈ q on x in order; extracted using f;
4     c = M_sel(v) + c;
  for c = c_1, ..., c_|c| do
5     let y = a_c(x | κ − consumed CPU time);
      if y is a solution to x or time is up then
          return y;
  return y;
```

Figure 6.1: Pseudocode for ∗ZILLA portfolio-based selectors, implemented by the *PortfolioBasedSelector* class. Note that lines 1-5 involve execution of subsidiary algorithm runs; here we use $a(\mathbf{x}|\kappa)$ to indicate a run of algorithm a on instance $\mathbf{x}$ with CPU time budget κ. We also use $\mathbf{u} + \mathbf{v}$ to indicate concatenation of lists $\mathbf{u}$ and $\mathbf{v}$.

more target algorithms. As such, implementation using the HAL API allows us to easily leverage built-in support for execution and data management. Doing so provides a number of benefits; for example, if feature values have already been extracted for an instance, they will be transparently retrieved from the database rather than re-extracted to conserve wall-clock time (but the CPU time consumption recorded by HAL will still reflect feature extraction time, ensuring consistently reproducible results). Similarly, target algorithm runs performed by the selector algorithm are automatically archived and reused whenever possible. In general, HAL can also transparently distribute subsidiary algorithm runs across compute clusters; however, the ∗ZILLA selector algorithm does

not exploit HAL's distributed execution support because all of its subsidiary runs are performed sequentially.

6.1.2 The ∗ZILLA Design Procedure

The ∗ZILLA design procedure (approximately) solves the per-instance portfolio-based selection problem by constructing and returning a single selector algorithm that optimizes performance on a given instance distribution. As mentioned in Section 6.1.1 and seen in Figure 6.1, a number of components are required to instantiate a ∗ZILLA selector algorithm; indeed, some of these components are taken directly from the per-instance portfolio-based selection instance (i.e., the candidate algorithms, instance features, and feature extractors; see also Chapter 3.2.2.2). By describing how the remaining components are determined (i.e., the minimal-cost features, binary classification model, selection model, backup solver, and pre-solvers), we complete our overview of the ∗ZILLA design procedure.

∗ZILLA operates by building predictive models that associate empirical performance with instance feature values. Thus, the first task of the ∗ZILLA design procedure is to extract feature values for all problem instances, and to evaluate the performance of all portfolio component algorithms on all instances. These steps correspond to lines 1-5 of the pseudocode in Figure 6.2, and are typically the most computationally intensive stage of the design procedure. Both of these tasks involve performing subsidiary algorithm runs in parallel, and using the HAL API means result archiving/reuse and distributed execution functionality is exploited without any extra implementation effort. In HAL, users can manually identify *minimal-cost* feature extractors (for example, extractors that read instance meta-data from the instance file header); the features extracted by minimal-cost extractors are used to by ∗ZILLA to define a minimal-cost feature indicator δ—the first outstanding component required to construct a selector algorithm.

Recalling from Section 6.1.1 that pre-solvers are ultimately only ever run on problem instances that the binary classification model predicts should be solved by the full selection pipeline, that the backup solver is only run on instances that pre-solvers are not run on (except in the case of unexpected errors), and that the selection model is only queried on instances that are not solved by a pre-solver, it is clear that all remaining components are interdependent and should be optimized simultaneously. To this end, ∗ZILLA first identifies a set of candidate–pre-solver pairings and associated runtimes — specifically, it treats every permutation of two of the top three portfolio components as a candidate pre-solver pairing, and considers four discrete run durations per pre-solver for each permutation (this strategy is inherited from MIPHYDRA [83]). For each pairing, ∗ZILLA learns binary classification and selection models (detailed in the next paragraph), and chooses the backup solver that optimizes performance on instances not otherwise solved by a pre-

Constructor: $B : (RT, FT, FV, \lambda, \mathbf{ps}) \mapsto (M_{cls}, M_{sel}, b, q)$;
where $RT \in \mathbb{R}^{|\mathbf{X}|\times|\mathbf{a}|}$ contains target algorithm algorithm performance data;
$FT \in \mathbb{R}^{|\mathbf{X}|\times|\mathbf{f}|}$ contains feature extractor performance data;
$FV \in \mathbb{R}^{|\mathbf{X}|\times|\mathbf{q}|}$ contains instance feature values;
$\lambda \in \{0,1\}^{|\mathbf{q}|}$ indicates elements of $\mathbf{q}$ that define a list $\mathbf{q}'$ of minimal-cost features;
$\mathbf{ps} \in ([\mathbf{1}, |\mathbf{a}|] \times \mathbb{R})^{\mathbf{2}}$ is a list of (pre-solver index, runtime) pairs; and
$M_{cls} : \mathbb{R}^{|\mathbf{q}'|} \rightarrow \{\text{true}, \text{false}\}$ is a classification model;
$M_{sel} : \mathbb{R}^{|\mathbf{q}|} \rightarrow [1, |\mathbf{a}|]^m$ is a selection model;
$b \in [1, |\mathbf{a}|]$ is a backup solver index;
$q \in \mathbb{R}$ is the overall quality of the pre-solver combination

Input: set of parameterless target algorithms A;
set of target problem instances X;
set of target problem features Q;
set of feature extractors F, that collectively extract values for all $q \in Q$;
evaluator δ;
target algorithm CPU time budget $\kappa_{\mathbf{a}}$;
feature extraction CPU time budget $\kappa_{\mathbf{f}}$

Output: parameterless portfolio-based selector for the target problem a'

define $\mathbf{a}$ = vector of elements of A in some fixed order;
define $\mathbf{X}$ = vector of elements of X in some fixed order;
define $\mathbf{q}$ = vector of elements of Q in some fixed order;
define $\mathbf{f}$ = vector of elements of F in some fixed order;
1 run each feature extractor $f \in \mathbf{f}$ on each instance $\mathbf{x} \in \mathbf{X}$ with CPU time limit $\kappa_{\mathbf{f}}$;
2 let $E_{\mathbf{a}} \in \mathbb{R}^{|\mathbf{X}|\times|\mathbf{a}|}$ contain the evaluations of δ for runs of each $a \in \mathbf{a}$ on each $\mathbf{x} \in \mathbf{X}$;
3 run each target algorithm $a \in \mathbf{a}$ on each instance $\mathbf{x} \in \mathbf{X}$ with CPU time limit $\kappa_{\mathbf{a}}$;
4 let $E_{\mathbf{f}} \in \mathbb{R}^{|\mathbf{X}|\times|\mathbf{f}|}$ contain the evaluations of δ for runs of each $f \in \mathbf{f}$ on each $\mathbf{x} \in \mathbf{X}$;
5 let $F \in \mathbb{R}^{|\mathbf{X}|\times|\mathbf{q}|}$ contain values for each feature $q \in \mathbf{q}$ as extracted by runs of $\mathbf{f}$ on each $\mathbf{x} \in \mathbf{X}$;
6 let $\lambda \in \{0,1\}^{|\mathbf{q}|}$ indicate those $q_i \in \mathbf{q}$ extracted at minimal cost;
7 choose some $\mathbf{S} \in \operatorname*{argmax}_{\mathbf{S}' \in ([1,|\mathbf{a}|]\times\mathbb{R})^2} B(E_{\mathbf{a}}, E_{\mathbf{f}}, F, \lambda, \mathbf{S}')$; where the argmax is evaluated with respect to overall quality of the pre-solver combination;
8 let $\mathbf{b} = B(E_{\mathbf{a}}, E_{\mathbf{f}}, F, \lambda, \mathbf{S})$;
return new $PortfolioBasedSelector(\mathbf{a}, \mathbf{q}, \mathbf{f}, \lambda, b_1, b_2, b_3, \mathbf{S})$;

Figure 6.2: Pseudocode for the ∗ZILLA portfolio-based selection design procedure, implemented by the *MatlabPortfolioBasedSelectorBuilder* class (dependence on MATLAB occurs via *B*). Additional inputs and outputs associated with *B* are not indicated here, but are listed in Appendix A.
Operations at lines 1, 3, 7, and 8 are implemented via subsidiary algorithm runs. In practice, the argmax at line 7 is approximate—not every $\mathbf{S}' \in ([1, |\mathbf{a}|] \times \mathbb{R})^2$ is tested (see Section 6.1.2).

solver or through the selection pipeline. Overall selector performance for each pre-solver candidate pairing is estimated using (10-fold, by default) cross-validation, and the components associated with the top-performing pairing are used to construct the final selector algorithm that is returned as the solution to the portfolio-based selection problem instance.

The ∗ZILLA procedure implemented for HAL 1.1 performs model building, backup solver selection, and cross-validation by calling the MATLAB model-building code used by MIPZILLA [84]. As such, it supports a number of selector model classes, most notably including the linear regression models used in earlier versions of SATZILLA [64, 81] and the more robust cost-sensitive decision forest models added for MIPZILLA [84]; for more details, we refer the reader to the cited publications. In order to abstract this cross-language dependency from the Java implementation of ∗ZILLA, we confined all model-building functionality to a standalone model-building algorithm (indicated by B in Figure 6.2); ∗ZILLA simply runs this algorithm as it would a feature extractor or target algorithm. This approach has two benefits: first, it allows the potentially costly pre-solver selection task to be trivially distributed across compute clusters using existing HAL functionality for distributed algorithm execution; and second, the MATLAB dependency can eventually be removed through drop-in replacement of the model building algorithm.

We end our discussion of ∗ZILLA by noting a consideration that is important when using its current implementation. Some of the design choices made in the ∗ZILLA selector algorithm only make sense for certain classes of performance metrics. In particular, the use of pre-solvers implicitly assumes that the performance metric under optimization is positively correlated with runtime, and the MATLAB model builder further assumes that it is distributive over addition (meaning that the evaluation of runs a and b grouped together equals the sum of their independent evaluations, but admitting an overall maximum evaluation value in order to support PAR-1 and 10). The original SATZILLA and MIPZILLA implementations were designed for optimizing time-like performance metrics (see e.g., Chapter 5) for which these assumptions hold; however, they fail for many solution quality metrics. To see this, observe that the solution quality of a failed run immediately followed by a successful run would intuitively equal the solution quality of the successful run alone, but the solution quality of the failed run alone would usually be assigned a nonzero value. The situation of a failed run followed by a successful one occurs whenever a pre-solver fails to solve an instance that is then solved through selection. As a result, care must be taken when using ∗ZILLA to optimize solution quality; e.g., by disabling pre-solver selection and by ensuring that ∗ZILLA measures the solution quality of feature extractor runs as zero.

6.2 Per-instance Configuration with HYDRA-∗

Having completed our discussion of ∗ZILLA, we turn to the design of HYDRA-∗. As discussed in Chapter 3.2.2.3, HYDRA [83, 84] approximately solves the per-instance configuration problem by combining algorithm configuration and per-instance portfolio-based selection procedures. Until now, all versions of HYDRA have used PARAMILS [38, 40] for configuration and SATZILLA [64, 81] for selection. However, several configuration procedures are available in HAL, so one of our goals for this HYDRA-∗ design is to achieve independence from any specific choice of configuration or portfolio-based selection sub-procedure, in addition to achieving target problem independence. In doing so we remain faithful to the original meta-algorithmic design of HYDRA, but employ object-oriented design techniques that allow variants of the procedure to be easily implemented and investigated.

6.2.1 The HYDRA-∗ Design Procedure

The characteristic of HYDRA that distinguishes it from other per-instance configuration procedures discussed in Chapter 3.2.2.3 is its use of a dynamic performance metric to iteratively add components to a portfolio for algorithm selection. The dynamic metric depends on a static user-provided metric; when evaluating an algorithm run on instance **x**, it returns the lesser of the static metric's evaluation of that run, and the static metric's evaluation of the current portfolio performance on **x**. In this way, candidates for addition to the portfolio are not penalized for being outperformed by an existing portfolio component on some instances, so long as they can improve the portfolio for others. This metric is discussed in more detail by the authors of HYDRA [83, 84].

While in principle a procedure like HYDRA-∗ could be implemented by combining an algorithm configuration procedure, a per-instance portfolio-based selection procedure, and a dynamic performance metric, in practice such an implementation would end up wasting significant computational resources during configuration. Since an arbitrarily chosen configuration procedure would not be aware of the semantics of a given dynamic performance metric, it could end up performing runs for much longer than necessary, only to have the metric return the same evaluation as if the run been terminated earlier. For this reason, in HYDRA-∗ we implement a *dynamic target algorithm* that is able to terminate its own runs if it is established that they are outperformed by the the current portfolio, and outputs the appropriate dynamic evaluation on completion. We then configure this dynamic target algorithm using a performance evaluator that simply obtains its evaluations from that output. This procedure is specified more precisely in the pseudocode of Figure 6.3; in line 1, the function call $w(\mathscr{A}, g_i)$ obtains a dynamic target algorithm from the user-provided parameterized algorithm $\mathscr{A}$ and the portfolio performance in iteration i (represented by the function g_i).

Constructor: algorithm configuration design procedure a_{ac};
portfolio-based selection design procedure a_{pbs};
single-algorithm analysis procedure a_{saa};
portfolio candidate filter a_{pcf};
Input: parameterized target algorithm $\mathscr{A}$;
set of target problem instances X;
set of target problem features Q;
set of feature extractors F, that collectively extract values for all $q \in Q$;
performance evaluator δ;
aggregator μ;
number of iterations n_{it};
number of parallel configuration runs n_{ac};
number of portfolio components per iteration n_{cpi}
Output: parameterless portfolio-based selector for the target problem a'

define $g_0 : \mathbf{x} \in X \mapsto \infty$; the initial portfolio performance map;
define $w : \langle \mathscr{A}, g : X \to \mathbb{R} \rangle \mapsto$ algorithm that, for input $\mathbf{x}$, runs $\mathbf{r} = \mathscr{A}(\mathbf{x})$ and outputs $\min(\delta(\mathbf{r}), g(\mathbf{x}))$;
define δ' : dynamic algorithm run $\mathbf{r} \mapsto$ output of $\mathbf{r}$ corresponding to performance evaluation;

let $X_0 := X$;
let $A_0 := \emptyset$;
for $i = 1, \dots, n_{it}$ **do**
1 let $\mathscr{A}_i = w(\mathscr{A}, g_{i-1})$;
2 let $C = \{a_{ac}(\mathscr{A}_i, X_{i-1}, \delta', \mu, \mu)$ for n_{ac} different random seeds$\}$;
let $g_{pcf} : \mathbf{x} \in X \mapsto 0$ **if** $i = 1$ **else** g_{i-1};
3 $A_i = A_{i-1} \cup a_{pcf}(C, X, \delta, \mu, g_pcf, n_{cpi})$;
4 let $o_i = a_{pbs}(A_i, X, Q, F, \delta, \mu)$;
5 let $\mathbf{y}_i = a_{saa}(o_i, X, \delta, \mu, \mu)$;
define $g_i : \mathbf{x} \mapsto$ evaluation of o_i on $\mathbf{x}$ according to the original evaluator δ;
let $X_i = \{\mathbf{x} \mid \mathbf{x} \in X$, and $\mathbf{x}$ is solved by selection and not, e.g., a pre-solver$\}$;
return $o_{n_{it}}$

Figure 6.3: Pseudocode for the HYDRA-$*$ per-instance configuration procedure, implemented by the *GenericHydraBuilder* class. Additional inputs and outputs are listed in Appendix A.

Currently available algorithm configuration procedures only *approximately* solve the algorithm configuration problem, and most exhibit a large degree of variability in output design performance across independent runs. To mitigate the risk of a single "unlucky" configuration run adversely affecting final design quality, HYDRA-$*$ performs $n_{ac} > 1$ independent configuration runs in each iteration (see line 2 of Figure 6.3). The designs produced by each of these independent runs are then filtered to choose the new portfolio components (line 3); procedures for performing this filtering are discussed in Section 6.2.2.

Once the updated set of portfolio components is determined, a procedure for the per-instance portfolio-based selection problem is used to build a selector (note that this selector is constructed using the original target algorithm, performance metric, and instance set, not the dynamic versions used for configuration). This selector algorithm is analyzed and the portfolio performance measurements used for constructing the dynamic target algorithm are updated. Finally, an instance set is assembled for the next iteration of configuration, omitting any instances that are easy for the current portfolio in order to bias the configuration procedure toward designs that are likely to result in improvement. Since algorithm configuration, portfolio-based selection, and single-algorithm analysis tasks are delegated to existing procedures, only portfolio candidate filtering components remain to be described.

The reader may notice that the pseudocode of Figure 6.3 appears to waste computational effort, because feature extraction and target algorithm performance assessment is implicitly performed from scratch in each iteration during portfolio-based selection. This is indeed an accurate depiction of our HYDRA-$*$ implementation for HAL. However, thanks to the automatic run reuse provided by the HAL framework, this conceptually simple design does not actually result in significant inefficiencies; target algorithm and feature extractor runs are simply retrieved from the database if they have been performed before. This automatic run reuse also benefits HYDRA-$*$ in that runs performed by independent configuration jobs are also reused, potentially saving significant effort (e.g., when there is a particularly strong default configuration and a large training instance set).

6.2.2 Portfolio Candidate Filtering

Portfolio candidate filtering is the task of identifying a subset of algorithms with the highest potential to improve an algorithm portfolio from a list of candidates. This filtering is necessary because the cost of solving the portfolio-based selection problem typically increases at least linearly (depending on the model used) with the number of candidates under consideration. We note that portfolio candidate filtering can formally be considered a meta-algorithmic analysis problem in its own right (albeit of narrow interest), and is related to the k-way Comparison problem.

In the first version of HYDRA, one portfolio component was added per iteration [83]. In order to

choose this component, the designs of all n_{ac} configuration runs were analyzed on the full training instance set (under the same per-run CPU time budget used during configuration), and the design that provided the greatest aggregate performance improvement over the current portfolio according to the user-provided performance metric was returned.

When developing HYDRA-MIP [84], the authors identified a practical problem with this approach. Since algorithm configuration can take a long time to complete (commonly several days), and since the optimal portfolio might have several components, selecting only a single component per iteration can result in prohibitively long HYDRA runs. Observing that different configuration runs on the same instance set can yield designs that are optimized for different instance subsets, they proposed adding multiple portfolio components per iteration. Since doing so implies an increased data collection cost during subsequent portfolio-based selection with MIPZILLA (typically the per-run cutoff used in configuration is much lower than that used in portfolio-based selection), they offset this cost by changing the candidate filtering procedure to analyze designs based only on the runs that were performed during configuration. They refer to this modified procedure as $\text{HYDRA}_{M,k}$, where M indicates the model used by the MIPZILLA (LR for linear regression, DF for cost-sensitive decision forests) and k indicates the number of components added per iteration.

We implemented both of these filtering strategies, as well as some intuitive generalizations enabled by the centralized data management offered by HAL. The *independent validation candidate filter* of Figure 6.4 implements the strategy used by $\text{HYDRA}_{M,1}$, but is generalized to work for arbitrary $k \geq 1$; we note that, as usual, this filter will automatically reuse run data during independent validation when possible. The *reused-run candidate filter* of Figure 6.5 implements the strategy used in $\text{HYDRA}_{M,k>1}$, and in particular only reuses runs that were performed by the configuration run that produced the design under consideration. Finally, the *reuse-maximizing candidate filter* of Figure 6.6 can be seen as an intermediate alternative to the other two filters; it does not perform new validation runs, but instead explicitly queries the HAL data manager to reuse relevant run data regardless of the context in which it was originally acquired (for example, runs performed by unrelated configuration runs).

6.3 Performance Validation

Although a number of functional tests were used during development of the ∗ZILLA and HYDRA-∗ procedures to ensure their behaviour was consistent with the discussed designs, it is nonetheless important to validate their performance on realistic, large-scale design tasks. Since both of our procedures are based on the published version of HYDRA-MIP, we validated them by independently replicating experiments reported in that publication [84].

Input: set of algorithm configuration runs C;
performance evaluator δ;
aggregator μ;
set of problem instances X;
portfolio performance map $g : X \to \mathbb{R}$;
number of designs to return n
Output: set of parameterless algorithms A

define $u : a \mapsto \mu\left(\langle \max(\delta(\text{new run of } a(\mathbf{x})) - g(\mathbf{x}), 0) \mid \mathbf{x} \in X \rangle\right)$;
let A' = a set containing the final design of each configuration run $\mathbf{c} \in \mathbf{C}$;
let $A = \emptyset$;
while $|A| < n$ *and* $|A'| > 0$ **do**
  choose some $a^* \in \underset{a \in A'}{\operatorname{argmin}}(u(a))$;
  $A = A \cup \{a^*\}$;
  $A' = A' - \{a^*\}$;
return A

Figure 6.4: Pseudocode for the HYDRA-$*$ independent-validation candidate filter, implemented by the *IndependentValidationIncumbentSelector* class. Here we adopt a "vector-builder" notation analogous to standard set-builder notation.

Input: set of algorithm configuration runs C;
performance evaluator δ;
aggregator μ;
set of problem instances X;
portfolio performance $g : X \to \mathbb{R}$;
number of designs to return n
Output: set of parameterless algorithms A

define $\alpha : \mathbf{c} \in C \mapsto$ the final design of configuration run $\mathbf{c}$;
define $\beta : \mathbf{c} \in C \mapsto$ the set $X' \subseteq X$ on which $\mathbf{c}$ performed a full-length run of $\alpha(\mathbf{c})$;
define $\gamma : (\mathbf{r}, \mathbf{x}) \in R \times X \mapsto$ the full-length run of $\alpha(\mathbf{c})$ on $\mathbf{x}$ performed by $\mathbf{c}$;
define $u : \mathbf{c} \in C \mapsto \mu\left(\langle \max(\delta(\gamma(\mathbf{c}, \mathbf{x})) - g(\mathbf{x}), 0) \mid \mathbf{x} \in \beta(\mathbf{c}) \rangle\right)$;
let $A = \emptyset$;
while $|A| < n$ *and* $|C| > 0$ **do**
  choose some $c^* \in \underset{c \in C}{\operatorname{argmin}}(u(r))$;
  $A = A \cup \{\alpha(r^*)\}$;
  $C = C - \{c^*\}$;
return A

Figure 6.5: Pseudocode for the HYDRA-$*$ reused-run candidate filter, implemented by the *SameTuningRunArchivedRunIncumbentSelector* class. Here we adopt a "vector-builder" notation analogous to standard set-builder notation.

Constructor: data manager d
Input: set of algorithm configuration runs C;
performance evaluator δ;
aggregator μ;
set of problem instances X;
portfolio performance $g : X \to \mathbb{R}$;
number of designs to return n
Output: set of parameterless algorithms A

define $\beta : a \mapsto$ the set $X' \subseteq X$ of instances on which a full-length run of a exists in d;
define $\gamma : (a, \mathbf{x} \in X) \mapsto$ a full-length run of $a(\mathbf{x})$ from d;
define $u : a \mapsto \mu\left(\langle \max(\delta(\gamma(a,x)) - g(\mathbf{x}), 0) \mid \mathbf{x} \in \beta(a)\rangle\right)$;
let $A' =$ a set containing the final design of each configuration run $\mathbf{c} \in \mathbf{C}$;
let $A = \emptyset$;
while $|A| < n$ *and* $|A'| > 0$ **do**
  choose some $a^* \in \operatorname*{argmin}_{a \in A'}(u(a))$;
  $A = A \cup \{a^*\}$;
  $A' = A' - \{a^*\}$;
return A

Figure 6.6: Pseudocode for the HYDRA-$*$ reuse-maximizing candidate filter, implemented by the *ReuseMaximizingArchivedRunIncumbentSelector* class. Here we adopt a "vector-builder" notation analogous to standard set-builder notation.

6.3.1 Experimental Setup

The HYDRA-MIP paper [84] presents the results of a large number of computationally expensive experiments optimizing CPLEX on a variety of mixed-integer programming benchmark sets, requiring nearly 7.5 CPU years in total to complete. In order to validate our implementations at somewhat lower computational cost, we use just one of these sets, CL∪REG. From that paper:

> *CL∪REG* is a mixture of two homogeneous subset[s], CL and REG. CL instances come from computational sustainability; they are based on real data used for the construction of a wildlife corridor for endangered grizzly bears in the Northern Rockies [28] and encoded as mixed integer linear programming (MILP) problems. We randomly selected 1000 CL instances from the set used in [41], 500 for training and 500 for testing. REG instances are MILP-encoded instances of the winner-determination problem in combinatorial auctions. We generated 500 training and 500 test instances using the `regions` generator from the Combinatorial Auction Test Suite [55], with the number of bids selected uniformly at random from between 750 and 1250, and a fixed bids/goods ratio of 3.91 (following [57]).

We chose CLUREG primarily because it appears from the results published in [84] that HYDRA-MIP was consistently close to convergence on this set: test performance for both tested HYDRA versions—$\texttt{Hydra}_{\texttt{DF,1}}$ and $\texttt{Hydra}_{\texttt{DF,4}}$ —remained similar and essentially stable after 3 of 5 iterations (although $\texttt{Hydra}_{\texttt{DF,4}}$ did improve somewhat in iteration 5), which intuitively suggests they had identified strong portfolio components for each of the two homogeneous subsets by that point. This consistent convergence is important to our validation, as not much can be concluded from two runs producing designs with different performance characteristics, but which are not yet near convergence.

Our experimental setup follows the HYDRA-MIP paper [84] as closely as possible. We used CPLEX 12.1 (see also Chapter 5) with the parameter configuration space obtained from the authors of the HYDRA-MIP paper as our parameterized target algorithm; this space contains $4.75 \cdot 10^{45}$ distinct configurations. Similarly, we used a MIP feature extractor and associated features obtained from its authors.[1] Because the performance of $\texttt{Hydra}_{\texttt{DF,1}}$ and $\texttt{Hydra}_{\texttt{DF,4}}$ designs were reported to be very similar after three iterations, we elected to focus on the HYDRA-$*$ variant corresponding to $\texttt{Hydra}_{\texttt{DF,4}}$ that uses our reused-run portfolio candidate filter, and we limited our experiments to three iterations. In each HYDRA-$*$ iteration, 25 independent algorithm configuration runs were performed in parallel; each of these configuration runs optimized the PAR-10 performance metric using a per-run cutoff of 300 CPU seconds and an overall time limit of 2 CPU days. To test the ability of HYDRA-$*$ to work with arbitrary algorithm configuration procedures, we performed analogous experiments using both PARAMILS and ROAR (see also Chapter 5.1.2)[2]; since no alternative to $*$ZILLA is available, we were not able to test the ability of HYDRA-$*$ to work with arbitrary portfolio-based selection procedures. $*$ZILLA was configured to use a per-run cutoff of 3 600 CPU seconds when evaluating portfolio candidates, and to use decision forests consisting of 99 MATLAB R2010a cost-sensitive decision trees in order to learn classification and selection models.

To independently validate our $*$ZILLA implementation, and to provide a context for evaluation of HYDRA-$*$ results, we constructed a $*$ZILLA selector from the same three CPLEX configurations that were used to build the original MIPZILLA selector (namely, the default configuration, and the configurations independently optimized for CL and REG from [41]). We then compared this $*$ZILLA selector to the designs produced after each HYDRA-$*$ iteration[3] using HAL's Comprehensive *k*-way

[1] After completion of these experiments, we were informed that the feature extractor we were provided differs from the one used in the HYDRA-MIP experiments of [84] in that it extracts 9 additional features, for 148 instead of 139 in total. However, in light of our discovery of perfect instance-subset predictive features for both feature extractors (see Section 6.3.2), we elected not to spend the CPU time required to repeat our experiments with the more restricted feature extractor.

[2] We note that ROAR was not modified for use with HYDRA-$*$, and we used the the version of PARAMILS from the HYDRA-MIP paper. [84]

[3] The original HYDRA-MIP and MIPZILLA designs could not be included in the *k*-way comparisons because executable implementations of these designs are not available.

Comparison plugin (see Chapter 5.1.1.3). To investigate performance differences arising from the choice of PARAMILS versus ROAR, we used the Comprehensive Pairwise Comparison plugin (see Chapter 5.1.1) to further compare the final designs from the corresponding HYDRA-$*$ runs. All analyses used the same 3 600 CPU second per-run cutoff that was also used during $*$ZILLA runs.

All experiments were conducted with a late beta version of HAL 1.1 and its plugins; experiment designs are available as a HAL export on our website [63]. Algorithm runs were executed on a Grid Engine cluster of 55 identical dual-processor Intel Xeon 3.2GHz nodes with 2MB cache and 4GB RAM running openSUSE Linux 11.1. Except where otherwise noted, reported runtimes indicate total CPU time used, as measured by HAL. Data management was performed using a MySQL database running on a dedicated server.

6.3.2 Results

In total, our experiments required just over 1.5 CPU years to complete. Our HYDRA-$*$ experiments required 277 CPU days with PARAMILS (160 for PARAMILS, 117 for $*$ZILLA), and 271 CPU days using ROAR (151 for ROAR, 120 for $*$ZILLA). Our standalone $*$ZILLA experiment required 12 CPU days, and our analysis experiments required 8 CPU days (4 each for the training and test sets). However, thanks to HAL's automatic run reuse, overall CPU usage was approximately 120 CPU days lower than indicated by these figures. For instance, the training set analyses consumed negligible CPU time, as runs for all selections were reused from previous $*$ZILLA runs. Also, since the feature extractors and portfolio components executed by $*$ZILLA in iteration i of a HYDRA-$*$ run are reused in iterations $j > i$, the reported $*$ZILLA runtime for a j-iteration HYDRA-$*$ run overstates actual CPU usage by a factor linear in j. Using the notation of Figure 6.3, CPU usage by feature extractors is $c_f \approx \sum_{f \in \mathbf{f}} \sum_{\mathbf{x} \in \mathbf{X}}$(CPU time of f on $\mathbf{x}$), and by components is $c_a(j) \propto \sum_{a \in \mathbf{a}_j} \sum_{\mathbf{x} \in \mathbf{X}}$(CPU time of a on $\mathbf{x}$); whereas reported times are $c'_f \approx j \cdot c_f$ and $c'_a(j) \propto \sum_{i \le j} c_a(i) \approx \frac{1}{2}(j+1) \cdot c_a(j)$ respectively. For our HYDRA-$*$ experiments, where $j = 3$ and $c_f \ll c_a(3)$, reported $*$ZILLA times are thus approximately double actual CPU usage. Finally, minor CPU time savings arise from incidental run reuse during parallel configuration, mostly during exploration near the default CPLEX configuration.

The results of our validation experiments are summarized in Table 6.1. Because the results reported in [84] only consider the cost of target algorithm and feature extractor execution (and not overheads due, for example, to loading and querying models), but HAL's performance metrics account for all consumers of CPU time, we report scores both including and neglecting overheads to facilitate comparison. Inspecting these results, we see that the qualitative behaviour observed in the HYDRA-MIP paper holds for our implementations as well—namely, that the $*$ZILLA design

Procedure	Train PAR1		Train PAR10		Train Solved	Test PAR1		Test PAR10		Test Solved
∗ZILLA oracle	-	**33**	-	**33**	**100%**	-	**33**	-	**33**	**100%**
MIPZILLA oracle (from [84])	-	**33**	-	**33**	**100%**	-	**33**	-	**33**	**100%**
∗ZILLA $_{DF}$	43	41	75	74	99.9%	**39**	**37**	**39**	**37**	**100%**
MIPZILLA$_{DF}$ (from [84])	-	**39**	-	**39**	**100%**	-	39	-	39	**100%**
HYDRA-∗ $_{DF,4}$ w/PARAMILS	**60**	**49**	**60**	**49**	**100%**	65	54	65	54	**100%**
HYDRA-∗ $_{DF,4}$ w/ROAR	62	53	62	53	**100%**	**60**	**52**	**60**	**52**	**100%**
HYDRA-MIP$_{DF,4}$ (from [84])	-	55	-	55	**100%**	-	61	-	61	**100%**

Table 6.1: Performance of ∗ZILLA and HYDRA-∗ designs on the CL∪REG MIP benchmark set. All HYDRA results are reported for designs after 3 iterations. PAR1 and PAR10 scores are listed in CPU seconds; the first entry indicates real-world performance as measured by HAL (including all overhead costs), and the second indicates feature extraction and target algorithm runtimes only (for comparison with the results of [84]).

consistently outperformed the best HYDRA-∗ designs (indeed, the *k*-way comparison's Friedman test and *post-hoc* Wilcoxon tests identified the ∗ZILLA design as the unambiguous best, with $p \approx 0.0 \times 10^{16}$). Also as expected, all ∗ZILLA and HYDRA-∗ designs were significantly outperformed by the corresponding hypothetical oracle. We also see that oracle performance was identical for ∗ZILLA and MIPZILLA, despite the fact that all performance data was independently collected, strongly suggesting that the basic data collection mechanisms in HAL are accurate and that the MATLAB random forest models are used correctly. Finally, we see that in nearly all other cases (except for the ∗ZILLA selector on the training set, where our design failed to solve one instance and was penalized by the PAR-10 metric), designs produced by HAL implementations slightly outperformed those produced by the corresponding non-HAL procedures; we speculate that this apparent trend was largely due to chance, as it is well known that the designs produced by independent runs of both PARAMILS and ROAR are highly variable in terms of quality. Overall, these results support the hypothesis that our ∗ZILLA and HYDRA-∗ procedures for HAL are correct.

It is informative to investigate the evolution of HYDRA-∗ designs as the iteration count increases. Figure 6.7 displays this progress in terms of test set performance (neglecting overhead costs to enable comparison with HYDRA-MIP data from [84]). In the left pane, PAR10 performance is displayed at each iteration and compared with the per-iteration performance reported for HYDRA-MIP; from this plot, it is again clear that the performance of both HYDRA-∗ designs is consistent with that of HYDRA-MIP$_{DF,4}$, and that the performance of the ∗ZILLA design is consistent with that of MIPZILLA. It is also evident that no improvements occurred for either HYDRA-∗ run after the second iteration. In the right pane, a scatter plot comparing the performance of HYDRA-∗ using ROAR after the first and second iterations shows that the second iteration significantly improves performance for almost all REG instances without affecting performance on CL instances (which

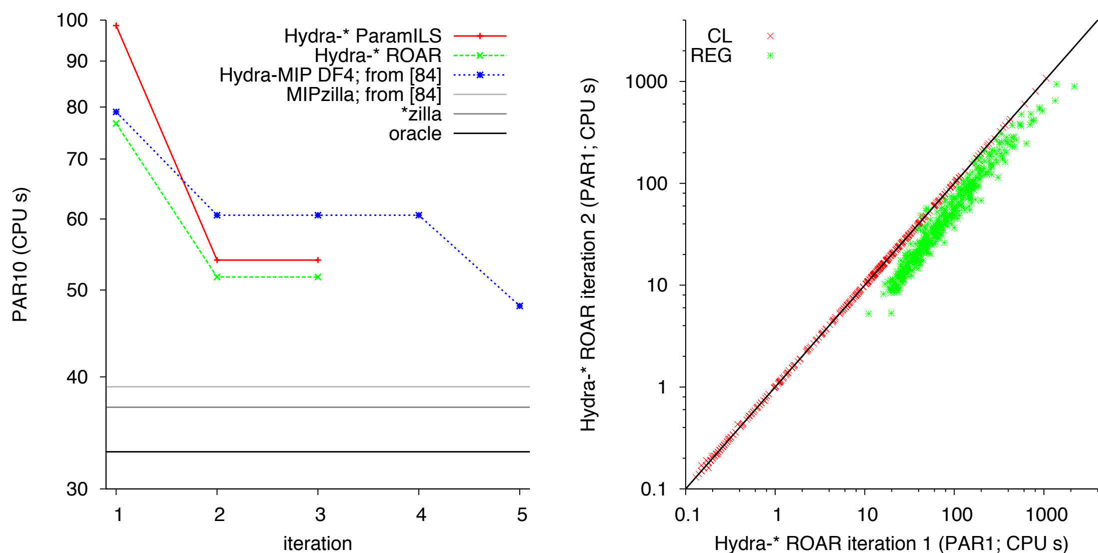

Figure 6.7: Performance evolution of HYDRA designs for CPLEX on the CL∪REG test set. Plots reflect feature extraction and target algorithm runtimes but exclude overheads (e.g., due to model loading and inference). (Left) PAR-10 performance as a function of HYDRA iteration; data for HYDRA-MIP and MIPZILLA is taken from [84]. (Right) Per-instance performance of HYDRA-$*$ with ROAR after 1 and 2 iterations; PAR-1 (or equivalently PAR10, as no instances timed out) in CPU seconds.

are still solved by first-iteration configurations). On closer analysis, we discovered that instance subset membership for CL∪REG can be determined perfectly using the values of certain extracted features, which explains the ability of $*$ZILLA to leverage the second-iteration HYDRA-$*$ configurations so accurately. The fact that no further improvement was observed in the third iteration shows that the first-iteration configurations being selected for CL instances were already quite strong.

Unfortunately, the performance illustrated in Figure 6.7 does not accurately reflect real-world performance, because overhead costs are neglected. The overlaid SCDs produced by HAL's Comprehensive k-Way Comparison procedure on the algorithm designs is illustrated in the left pane of Figure 6.8, and include all overhead costs; for comparison, analogous SCDs neglecting overheads are presented in the right pane. These plots show that overhead costs increase with the number of portfolio components (and thus, iteration), and are significant—on the order of 10 seconds (or approximately one third of mean oracle performance) for third-iteration designs. On further investigation, it became clear that these overheads are dominated by the cost of loading decision forest models into MATLAB; since the number of portfolio candidates increases after each iteration, so does the cost of model loading. However, unlike feature extraction, model loading costs are not yet considered during the $*$ZILLA model building process. In the future we hope this issue can be

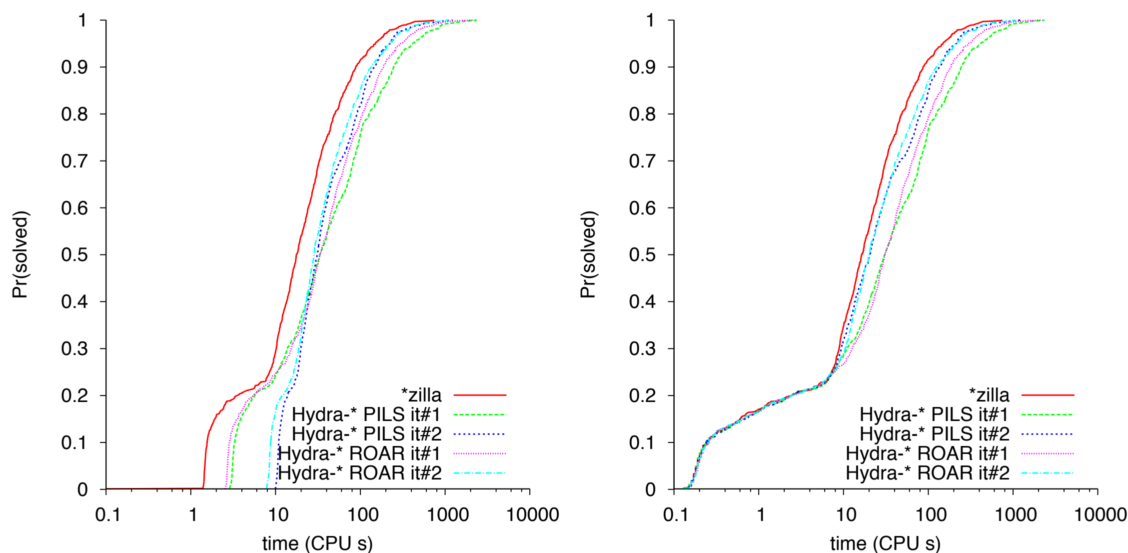

Figure 6.8: Effect of overhead costs on ∗ZILLA selector algorithm performance for CPLEX on the CLUREG test set. The left pane includes overheads (e.g., model loading and inference); the right pane only reflects feature extraction and target algorithm execution. In these plots, ∗ZILLA refers to the MIPZILLA-like selector constructed using configurations from [41].

handled automatically, but in the meantime, users of ∗ZILLA are advised take this cost into consideration when deciding between selector model classes (for example, parametric models like linear regression have negligible loading costs), setting associated model parameters (such as the number of trees in a decision forest), and choosing portfolio components.

Finally, we also see that the final HYDRA-∗ design using ROAR outperforms the final design using PARAMILS in terms of PAR-10 on the test set, despite slightly worse training performance. The Wilcoxon signed-rank test performed during pairwise comparison similarly indicates that the ROAR design statistically significantly outperforms the PARAMILS design at $\alpha = 0.05$ ($p = 1.6 \times 10^{-26}$); however, it also indicates that ROAR outperforms PARAMILS on the training set ($p = 1.9 \times 10^{-10}$). This discrepancy in training set performance arises from the fact that the Wilcoxon test considers the median paired performance difference, not PAR-10 (i.e., mean) performance. The correlation plots of Figure 6.9 show that the ROAR design performs better than the PARAMILS design on easier instances but slightly worse on harder instances (this is also visible, but less obvious, in the overlaid SCDs of Figure 6.8), and that the training set has slightly more hard instances than the test set. This is a good example of why statistical tests should be used with care.

Despite the performance differences observed between HYDRA-∗ with PARAMILS versus ROAR on the runs conducted for this specific per-instance configuration problem instance, we cannot draw

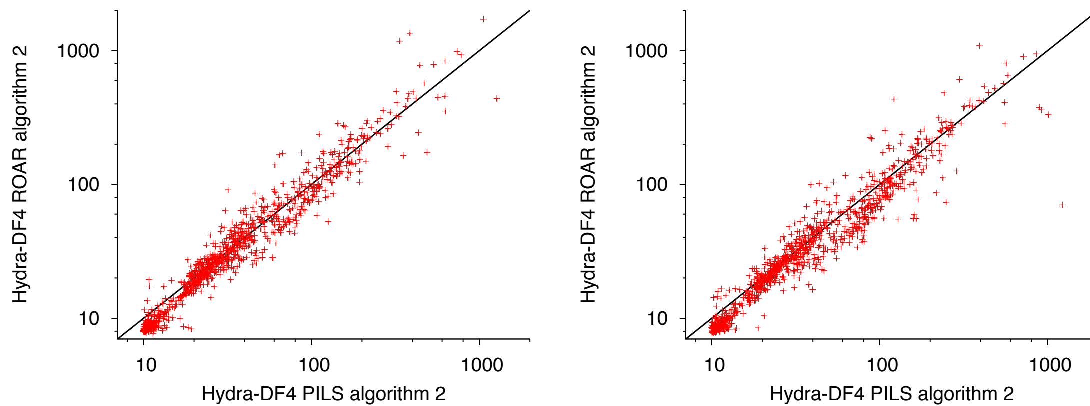

Figure 6.9: Performance correlation between designs found by HYDRA-∗ using ROAR versus using PARAMILS, for CPLEX on the CLUREG training (left) and test (right) sets. Units are PAR-10 (CPU s), including overheads (e.g., model loading and inference). A Spearman coefficient of $\rho = 0.97$ (both plots) reflects the correlation seen here.

the conclusion that ROAR is a better choice overall; indeed we suspect (but cannot confirm) that the differences are simply due to chance—since HYDRA-∗ is randomized, it is possible that PARAMILS might outperform ROAR in another independent HYDRA-∗ run; similarly, it is quite possible that PARAMILS might outperform ROAR on a different per-instance configuration problem instance. To be able to make a more conclusive claim, one would need to perform a pairwise performance comparison not between the two final HYDRA-∗ designs, but between the two randomized HYDRA-∗ design procedure variants themselves, using a benchmark set of per-instance configuration problem instances. While in principle HAL supports conducting such an experiment in its current version, in practice the computational cost of doing so with realistic target algorithms like CPLEX is prohibitive (recall that evaluating performance on this single per-instance configuration problem instance just once for each variant took 1.5 CPU years).

Chapter 7

Conclusions & Future Work

This thesis has attempted to address several technical challenges commonly encountered when working with sophisticated empirical methods for analysis and design of high-performance solvers for hard computational problems. To this end we introduced HAL, a versatile and extensible environment for empirical algorithmics. It is built upon a novel conceptual framework that formalizes notions of meta-algorithmic problems and procedures, and provides a unified basis for studying a wide variety of empirical design and analysis techniques. HAL facilitates the automated application of empirical techniques involving computationally intensive analysis and design tasks, and it supports the development and critical assessment of novel empirical analysis and design procedures.

The current implementation of our framework, HAL 1.1, can be used to analyze and design algorithms for arbitrary target problems. HAL 1.1 can execute experiments on local machines, remote machines, or distributed across compute clusters; it offers fine-grained experiment monitoring and control before, during, and after execution; and it automatically archives and efficiently reuses experimental data. HAL 1.1 also provides a versatile API for developing and deploying new meta-algorithmic analysis and design procedures. This API was used to develop plugins that together provide eight state-of-the-art procedures for solving three performance analysis problems (single-algorithm analysis; pairwise comparison; *k*-way comparison) and three algorithm design problems (algorithm configuration with PARAMILS, GGA, and ROAR; per-instance portfolio-based selection with ∗ZILLA; per-instance configuration with HYDRA-∗). We demonstrated all eight procedures in experiments involving prominent solvers for SAT (SPEAR [3]) and MIP (GUROBI [29] and CPLEX [45]), and illustrated the process of building upon existing procedures to develop more complex ones by describing how our HYDRA-∗ procedure for per-instance configuration integrates subsidiary procedures for algorithm configuration and per-instance portfolio-based selection.

We believe that automated meta-algorithmic design techniques have tremendous potential to

influence algorithm design in the future, and we hope the community finds our work useful as one of the first efforts to collectively describe and support these techniques. In the months since HAL was initially released, we have indeed seen promising results: three independent publications involving researchers from our group have used HAL to conduct automated design experiments [44, 79, 21], and in this thesis we use it to provide the first fully-automated, domain-independent procedures for per-instance portfolio-based selection and per-instance configuration, two high-profile computer-aided design techniques.

Our group continues to actively develop, extend, and employ the HAL framework. Several new meta-algorithmic procedures are under development, including advanced procedures for model-based algorithm configuration and for bootstrapped single-algorithm analysis. As we gain experience developing these procedures with HAL, we continue to refine the HAL API, making it more powerful and more intuitive to use. We have also begun to develop useful general-purpose tools for meta-algorithmic developers, including components for building predictive models that are useful, for example, to simulate *surrogate* target algorithms that promise to greatly improve the efficiency with which meta-algorithmic procedures themselves can be evaluated. We are also actively improving core infrastructure performance and functionality, for example optimizing data manager query efficiency and adding support for algorithm execution in a wider variety of common high-performance computing environments. Finally, we continue to focus on streamlining the end-user experience through improvements to HAL's documentation and web interface.

As HAL continues to improve and to support successful applications, and as meta-algorithmic analysis and design techniques themselves gain traction, we anticipate a demand for tighter integration between HAL and the integrated development environments used to implement the target algorithms upon which it operates. This integration would facilitate application of automated meta-algorithmic techniques from the very beginning of the algorithm design process (instead of after initial implementation, as in the experiments of this thesis), and could support computer-aided algorithm design techniques wherein algorithms are explicitly designed for optimization by meta-algorithmic tools (see, e.g., [33, 53, 34, 78]). Ultimately, we hope that HAL will help to inspire many more researchers to adopt state-of-the-art computer-aided design methods and empirical best practices in their own work.

Bibliography

[1] R. K. Ahuja and J. B. Orlin. Use of representative operation counts in computational testing of algorithms. *Informs Journal on Computing*, 8(3):318–330, 1996. → pages 6

[2] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pages 142–157. Springer Verlag, 2009. → pages 2, 11, 26, 43, 84

[3] D. Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, Department of Computer Science, University of British Columbia, Vancouver, Canada, 2008. → pages 46, 70

[4] P. Balaprakash, M. Birattari, and T. Stützle. Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In *Proceedings of the 4th International Workshop on Hybrid Metaheuristics*, pages 108–122. Springer Verlag, 2007. → pages 2, 11, 25

[5] A. Balint, D. Gall, G. Kapler, R. Retz, D. Diepold, and S. Gerber. EDACC: An advanced platform for the experiment design, administration, and analysis of empirical algorithms. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (to appear)*. Springer Verlag, 2011. URL http://sourceforge.net/projects/edacc. → pages 9

[6] T. Bartz-Beielstein, C. Lasarczyk, and M. Preuss. Sequential parameter optimization. In *IEEE Congress on Evolutionary Computation*, pages 773–780. IEEE, 2005. → pages 26

[7] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann Publishers Inc., 2002. → pages 2, 11, 25

[8] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stutzle. F-race and iterated F-race: An overview. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss, editors, *Empirical Methods for the Analysis of Optimization Algorithms*, chapter 13, pages 311–336. Springer, 2010. → pages 11, 25

[9] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: an emerging direction in modern search technology. In *Handbook of Metaheuristics*, pages 457–474. Kluwer Academic Publishers, 2003. → pages 10

[10] E. Burke, T. Curtois, M. Hyde, G. Kendall, G. Ochoa, S. Petrovic, and J. Vazquez-Rodriguez. HyFlex: A flexible framework for the design and analysis of hyper-heuristics. In *Proceedings of the 4th Multidisciplinary International Scheduling Conference: Theory and Applications*, pages 790–797, 2009. → pages 8, 10

[11] M. Chiarandini, C. Fawcett, and H. H. Hoos. A modular multiphase heuristic solver for post enrollment course timetabling (extended abstract). In *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling*, 2008. → pages 1, 11

[12] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. In *IEEE Proceedings - Software*, volume 150, pages 161–175, 2003. → pages 10

[13] P. R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, 1995. → pages 1, 6

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. → pages 9, 15

[15] H. Crowder, R. S. Dembo, and J. M. Mulvey. On reporting computational experiments with mathematical software. *ACM Transactions on Mathematical Software*, 5:193–203, 1979. → pages 1, 6

[16] C. Demetrescu, A. Goldberg, and D. Johnson, editors. *Data Depth: Shortest Path Computations: Ninth DIMACS Challenge*, 2009. AMS. → pages 5

[17] L. Di Gaspero and A. Schaerf. EasyLocal++: An object-oriented framework for the flexible design of local-search algorithms. *Software - Practice and Experience*, 33:733–765, 2003. → pages 10

[18] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. → pages 10

[19] DIMACS – Center for Discrete Mathematics & Theoretical Computer Science. DIMACS implementation challenges, 2011. URL http://http://dimacs.rutgers.edu/Challenges. → pages 5

[20] R. Ewald and A. M. Uhrmacher. Automating the runtime performance evaluation of simulation algorithms. In *Proceedings of the 2009 Winter Simulation Conference*, WSC '09, pages 1079–1091, 2009. → pages 8

[21] C. Fawcett, M. Helmert, and H. H. Hoos. FD-Autotune: Domain-specific configuration using Fast Downward. In *Proceedings of the 3rd Workshop on Planning and Learning, Co-located with ICAPS 2011*, 2011. → pages iii, 11, 71

[22] A. Fukunaga. Automated discovery of composite SAT variable selection heuristics. In *Proceedings of the National Conference on Artificial Intelligence*, pages 641–648. AAAI Press, 2002. → pages 11

[23] M. Gagliolo and J. Schmidhuber. Dynamic algorithm portfolios. In *Ninth International Symposium on Artificial Intelligence and Mathematics*, 2006. → pages 12

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. → pages 14, 18

[25] A. Gerevini, A. Saetti, and M. Vallati. An automatically configurable portfolio-based planner with macro-actions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, pages 191–199. AAAI Press, 2009. → pages 12

[26] F. Glover and G. Kochenberger, editors. *Handbook of Metaheuristics*. Kluwer Academic Publishers, Jan. 2003. → pages 10

[27] C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001. → pages 12

[28] C. P. Gomes, W. J. V. Hoeve, and A. Sabharwal. Connections in networks: A hybrid approach. In *Proceedings of the 5th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, pages 303–307. Springer Verlag, 2008. → pages 63

[29] Z. Gu, E. Rothberg, R. Bixby, et al. *Gurobi Optimizer 3.01*. Gurobi Optimization, 2010. URL http://www.gurobi.com. → pages 44, 70

[30] A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 475–479. IOS Press, 2004. → pages 12

[31] W. E. Hart, J. W. Berry, R. T. Heaphy, and C. A. Phillips. EXACT: The experimental algorithmics computational toolkit. In *Proceedings of the 2007 Workshop on Experimental Computer Science*. ACM, 2007. → pages 9

[32] S. Hert, T. Polzin, L. Kettner, and G. Schäfer. ExpLab: A tool set for computational experiments. Technical Report MPI-I-2002-1-004, Max-Planck-Institut für Informatik, 2002. URL http://explab.sourceforge.net/. → pages 8

[33] H. H. Hoos. Computer-aided design of high-performance algorithms. Technical Report TR-2008-16, University of British Columbia, Computer Science, 2008. → pages iii, 1, 11, 71

[34] H. H. Hoos. Programming by optimisation. Technical Report TR-2010-14, University of British Columbia, Computer Science, 2010. → pages 71

[35] H. H. Hoos and T. Stützle. *Stochastic Local Search—Foundations and Applications*. Morgan Kaufmann Publishers, USA, 2004. URL http://www.sls-book.net. → pages 1, 2, 6, 10, 83

[36] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 265:51–54, 1997. → pages 12

[37] F. Hutter. *Automating the Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, Department of Computer Science, University of British Columbia, Vancouver, Canada, 2009. → pages 20

[38] F. Hutter, D. Babić, H. H. Hoos, and A. Hu. Boosting verification by automatic tuning of decision procedures. In *Proceedings of 7th International Conference on Formal Methods in Computer-Aided Design*, pages 27–34. IEEE Computer Society, 2007. → pages 1, 11, 46, 47, 58

[39] F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 1152–1157. AAAI Press, 2007. → pages iv, 2, 11, 25, 84

[40] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009. → pages iv, 2, 11, 17, 25, 43, 58, 84

[41] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated configuration of mixed integer programming solvers. In *Proceedings of the 7th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, pages 186–202. Springer Verlag, 2010. → pages 1, 11, 63, 64, 68

[42] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligence, Special Issue on Learning and Intelligent Optimization*, pages 1–25, 2010. → pages 20, 23

[43] F. Hutter, H. H. Hoos, K. Leyton-Brown, and K. Murphy. Time-bounded sequential parameter optimization. In *Proceedings of the 4th International Conference on Learning and Intelligent Optimization*, pages 281–298. Springer Verlag, 2010. → pages 26

[44] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (to appear)*. Springer Verlag, 2011. → pages iv, 11, 26, 44, 71, 85

[45] International Business Machines Corp. *IBM ILOG CPLEX Optimizer 12.1*, 2009. URL http://www.ibm.com/software/integration/optimization/cplex-optimizer. → pages 44, 70

[46] M. Järvisalo, D. Le Berre, and O. Roussel. SAT Competition 2011, 2002. URL http://www.satcompetition.org/2011. → pages 12

[47] D. Johnson and C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, 1993. AMS. → pages 5

[48] D. Johnson and L. McGeoch. Experimental analysis of heuristics for the stsp. In *The Traveling Salesman Problem and its Variations*, pages 369–443. Kluwer Academic Publishers, 2002. → pages 5

[49] D. Johnson and M. Trick, editors. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, 1996. AMS. → pages 5

[50] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492, 1998. → pages 26

[51] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open source scientific tools for Python*, 2001–. URL http://www.scipy.org. → pages 8

[52] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC – instance-specific algorithm configuration. In *Proceedings of the 19th European Conference on Artificial Intelligence*, pages 751–756. IOS Press, 2010. → pages 2, 11, 12, 30, 34

[53] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 517–524. Morgan Kaufmann Publishers Inc., 2009. → pages 11, 71

[54] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005. → pages 9

[55] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pages 66–76. ACM, 2000. → pages 63

[56] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, pages 899–903. Springer Verlag, 2003. → pages 12

[57] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, 2009. → pages 63

[58] Lindawati, H. C. Lau, and D. Lo. Instance-based parameter tuning via search trajectory similarity clustering. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (to appear)*. Springer Verlag, 2011. → pages 2, 12, 30, 34

[59] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Non-model-based algorithm portfolios for SAT (extended abstract). In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing*, pages 369–370. Springer Verlag, 2011. → pages 2, 12, 28

[60] MathWorks, Ltd. *MATLAB*, 2011. URL http://www.matlab.com. → pages 8

[61] H. D. Mittelmann. Decision tree for optimization software, 2010. URL http://plato.asu.edu/guide.html. → pages 44, 45

[62] H. D. Mittelmann and A. Pruessner. A server for automated performance analysis of benchmarking data. *Optimization Methods and Software*, 21(1):105–120, 2006. → pages 8

[63] C. Nell, C. Fawcett, H. H. Hoos, and K. Leyton-Brown. HAL: A framework for the automated analysis and design of high-performance algorithms. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (to appear)*. Springer Verlag, 2011. URL http://hal.cs.ubc.ca. → pages iii, 41, 51, 65

[64] E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. H. Hoos. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 438–452, 2004. → pages 2, 27, 57, 58, 85

[65] I. H. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63:511–623, 1996. → pages 10

[66] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via http://lulu.com, 2008. URL http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza). → pages 11

[67] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957. → pages 10

[68] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2011. URL http://www.r-project.org. → pages 8

[69] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976. → pages 12

[70] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 117–133. ACM, 1998. → pages 34

[71] L. Simon and P. Chatalic. SatEx: A web-based framework for SAT experimentation. In *Electronic Notes in Discrete Mathematics*, page 3. Elsevier Science Publishers, 2001. → pages 8

[72] L. Simon, D. L. Berre, and E. A. Hirsch. The SAT 2002 Competition, 2002. URL http://www.satcompetition.org. → pages 8

[73] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. → pages 15

[74] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16:1024–1043, 1990. → pages 10

[75] M. J. Streeter and S. F. Smith. New techniques for algorithm portfolio design. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pages 519–527. AUAI Press, 2008. → pages 12

[76] D. A. D. Tompkins and H. H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *SAT*, pages 306–320. Springer Verlag, 2004. → pages 8

[77] D. A. D. Tompkins and H. H. Hoos. Dynamic scoring functions with variable expressions: New SLS methods for solving SAT. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, pages 278–292. Springer Verlag, 2010. → pages 11

[78] D. A. D. Tompkins, A. Balint, and H. H. Hoos. Captain Jack: New variable selection heuristics in local search for SAT. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing*, pages 302–316. Springer Verlag, 2011. → pages 1, 11, 71

[79] M. Vallati, C. Fawcett, A. E. Gerevini, H. H.Hoos, and A. Saetti. Generating fast domain-optimized planners by automatically conguring a generic parameterised planner. In *Proceedings of the 3rd Workshop on Planning and Learning, Co-located with ICAPS 2011*, 2011. → pages iii, 71

[80] S. Westfold and D. Smith. Synthesis of efficient constraint-satisfaction programs. *The Knowledge Engineering Review*, 16(1):69–84, 2001. → pages 10

[81] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. → pages 1, 2, 12, 18, 27, 52, 57, 58, 85

[82] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla2009: an automatic algorithm portfolio for SAT. Solver description, SAT competition 2009, 2009. → pages 52

[83] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 201–216. AAAI Press, 2010. → pages 2, 11, 12, 29, 34, 55, 58, 60, 85

[84] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *Proceedings of the 18th RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2011. → pages iv, 2, 11, 12, 28, 29, 51, 57, 58, 61, 63, 64, 65, 66, 67, 85

Appendix A

HAL 1.1 User Reference

This reference is designed to help new users come up to speed with HAL 1.1 as quickly as possible, and to act as a reference for the many meta-algorithmic procedures it contains. Updated versions of this guide will be maintained at http://hal.cs.ubc.ca.

A.1 Installation

Basic installation

- download the full zip package from http://hal.cs.ubc.ca.
- extract the zip to a working directory
- double click the `hal_<version>.jar` file, or type `java -jar hal_<version>.jar` from the command line to start the web server
- navigate to `http://<machine address>:8080/hal` to see the web UI

Note that the default port of 8080 can be changed by adding the command-line argument `-p <port>` when starting HAL, or by editing the `hal.json` file (below).

Suggested next steps:

- download the sample experiment package; import it using the HAL web interface, and try analysing the provided sample algorithm (note the algorithm requires Linux) to ensure the installation is working
- try defining your own algorithm and instance set, and analysing it

- try using other plugins

Setting HAL options

After the first start of HAL, a file called `hal.json` will be created. Basic settings can be modified by editing this file.

RECOMMENDED: if you have access to a MySQL server, use it instead of the default SQLite database. SQLite will likely cause errors if you attempt to run an experiment where more than one process accesses the same database simultaneously, and can cause issues with automatic refreshing in the GUI.

To do so, install the MySQL plugin (see below), and then edit the `hal.json` file; change:
`"database":"jdbc:sqlite:hal.db"`
to something like:
`"database":"jdbc:mysql://<user>@<mysql.server.url>:<port>/hal_db"`

We also recommend different HAL users maintain different HAL databases; you may even find it useful to maintain separate databases for separate "sets" of experiments. Import/export functionality can be used to quickly populate a database with required instances, algorithms, etc.

Plugin installation

To install a new meta-algorithmic procedure in HAL, simply place the corresponding `.jar` file into the `plugins` folder; no additional configuration is required. To upgrade a plugin, replace the existing plugin with the newer one. You must also delete the correspondingly-named subfolder in the `pluginlib` folder, if it exists.

Note that HAL must not be running when plugins are installed or upgraded.

A.2 Running Experiments

The HAL web user interface is designed to be intuitive; thus, in this section we provide guidance for some of the subtleties of the system, but do not provide step-by-step usage instructions. Information about the various settings for each of the meta-algorithmic procedures supported in HAL can be found in Section A.3 and A.4.

A.2.1 Instance Sets

To specify an instance set, use the Create New InstanceDistribution wizard. Notes for use:

- distribution name must be unique
- tags are used to associate problem instances with compatible algorithms (see below), but are otherwise completely up to the user
- an instance is exportable if it is completely-defined by a single file
- the file path must be valid on the system running HAL, not the system running the web browser
- the expression for matching files is a regular expression, not a command-line-style wildcard expression. For example, ".*" matches all files, not "*".

A.2.2 Algorithms

To add a new algorithm for use in HAL, use the "Define a new algorithm" wizard. Notes for use:

1. Algorithm specification:
   - The name+version combination must be unique in the database.
   - Supported instance tag sets are used to restrict which algorithms can be run on particular instances. A tag set such as {cnf, 3sat} marks the algorithm as compatible with any instance with both "cnf" and "3sat" tags. An algorithm can have multiple instance tag sets; at least one of them must be satisfied by an instance in order for it to be considered compatible. If no tags are specified, the HAL will assume the algorithm is compatible with all instances.
   - Advanced properties: deterministic: straightforward; note that if your algorithm has a seed as an input, mark it as *not* deterministic, and explicitly identify the seed parameter (below).
   - Advanced properties: exportable: An algorithm is exportable if and only if HAL can compress the entire directory tree rooted at the executable file, extract it on a different machine, and the algorithm will work on that second machine.
   - Advanced properties: cutoff-independence: Set this to true if the algorithm's behaviour will not change if cutoff inputs are changed but all other inputs remain the same. HAL uses this property to decide whether it can infer cutoffs or completions from previously-performed runs.
   - Executable: should be the first command to run the algorithm; if the algorithm is an interpreted script this might be `python` or `ruby`. Note that this path must exist *on the*

system running HAL, not the system running the browser that is interfacing with HAL (if the two differ). A utility is provided to upload resources to the server system from the UI.

- Command-line argument string: see example in the UI. Note that all input variables (specified like `$name$` in the command string) will need to be further defined in later steps. (NOTE: HAL does not currently support generating input files for target algorithms.)
- Algorithm output specification. Format is the same as the command-line argument specification; and as in that case, all identified output variables will require further specification. HAL will use this string to parse outputs from the standard output and error streams of the algorithm during execution.

2. Identifying configurable parameters:
   - A configurable parameter is a parameter whose value may be tuned by a configurator to optimize performance; all "other" settings will be left unchanged.
3. Specifying input domains:
   - For every input parameter marked as configurable, the domain of valid values must be specified. For other inputs, the "semantic default" can be used, or the domain can be specified manually.
   - Conditionalities between parameters can be specified, where conditionality refers to "activation" of parameters. For example, parameter X can be marked as active only if parameter Y takes some particular value. Note that HAL does not support domains that change depending on the values of other parameters (for example, x is in the domain [1, value of y]).
   - Prohibited configurations are similarly specified.

A.3 Analysis Procedures

In this section we briefly describe the meta-algorithmic procedures available as plugins for HAL 1.1, itemize the parameters, settings, and outputs of each (as visible in the user interface), and note any other important implementation-specific considerations.

A.3.1 SCD-Based Analysis

This comprehensive approach to single-algorithm analysis takes as input a single-algorithm analysis instance (containing a single target algorithm, a set of benchmark instances, and a performance metric) and additional settings including a maximum number of runs per target instance, a maximum CPU time per target run, a maximum number of total target runs, and a maximum aggregate runtime budget. It collects runtime data for the target algorithm on the instance distribution (in parallel, when specified) until a stopping criterion is satisfied. Summary statistics are computed over the instance distribution, and a solution cost distribution plot (see, e.g., Ch. 4 of [35]) is produced to illustrate (median) performance across all target runs on each instance.

Inputs/outputs of the SCD-Based Analysis procedure are listed in Table A.1.

A.3.2 Comprehensive Pairwise Comparison

The Comprehensive Pairwise Comparison procedure performs SCD-Based Analysis on two given algorithms, generates a scatter plot illustrating paired performance across the given instance set, and performs Wilcoxon signed-rank and Spearman rank correlation tests. The Wilcoxon signed-rank test determines whether the median of the paired performance differences between the two algorithms across the instance set is significantly different from zero; if so, it identifies the better-performing algorithm. The Spearman rank correlation test determines whether a significant monotonic performance correlation exists between them. Both tests are non-parametric, and so appropriate for the non-Gaussian performance data frequently encountered in empirical algorithmics.

Inputs/outputs of the Comprehensive Pairwise Comparison procedure are listed in Table A.2.

A.3.3 Comprehensive k-way Comparison

The Comprehensive k-way Comparison procedure performs SCD-Based Analysis for each of the k provided algorithms. It performs a Friedman test to determine whether any of the k algorithms significantly outperforms any of the others; if so, it performs Comprehensive Pairwise Comparison between the best-performing angorithm and the remaining algorithms, and identifies all algorithms that do not exhibit statistically significantly worse performance.

Inputs/outputs of the Comprehensive k-way Comparison procedure are listed in Table A.3.

A.4 Design Procedures

A.4.1 PARAMILS

ParamILS [39, 40] is a model-free configuration procedure based on an iterated local search over the space of possible parameter configurations. Its most commonly-used variant, FocusedILS, aggressively limits computational resource usage by capping algorithm runs based on the performance of the best incumbent design at any particular point during its sequential search process. It also exploits knowledge of parameter conditionality (i.e., parameters that only take effect if another parameter takes a certain value) to improve performance when many conditional parameters corresponding to parameterized heuristics are present. ParamILS is not limited to small parameter spaces, but any continuous-valued parameters must be discretized before it can be used. The ParamILS plugin for HAL implements the FocusedILS variant of ParamILS.

The ParamILS plugin is built on top of version 2.3.5 of the Ruby implementation of ParamILS. As such, it requires a Ruby interpreter be installed to be used. ParamILS may not support all custom performance metrics, but works with all metrics included with HAL by default. In particular, ParamILS only supports metrics with mean or median aggregators, and capping is only supported for PAR1 and PAR10 CPU time evaluators.

Inputs/outputs of the ParamILS procedure are listed in Table A.4.

A.4.2 GGA

GGA [2] is a model-free configuration procedure based on a genetic algorithm. Like ParamILS, it is not limited to small parameter spaces; unlike ParamILS, it is able to configure both discrete- and continuous valued parameters directly. It is also able to exploit a relatively limited form of parameter conditionality. The author-provided compiled implementation of GGA is highly parallelized, and is used as the basis for the HAL plugin.

The GGA plugin inherits several limitations from the binary it is based upon. In particular, it can only be used on 32-bit Linux platforms, and it also requires a Ruby interpreter to be installed. It will always attempt to run 8 target algorithms in parallel. Finally, it only supports the PAR1 performance metric.

Inputs/outputs of the GGA procedure are listed in Table A.5.

A.4.3 ROAR

ROAR is a simple, model-free implementation of the SMBO framework [44] that exploits an aggressive run-capping intensification strategy similar to that used by FocusedILS, and supports large numbers of both real-valued and discrete parameters, and complex parameter conditionality structures. ROAR is implemented purely in Java.

Inputs/outputs of the ROAR procedure are listed in Table A.6.

A.4.4 *ZILLA

SATZILLA [64, 81] is arguably the most successful application of solving the portfolio-based selection problem for algorithm design. In its original application to SAT, SATZILLA was designed by learning a performance prediction model for each of the portfolio components (target algorithms $a \in \mathbf{a}$) based on a set of SAT features. When SATZILLA is run on a new instance, it extracts features for that instance, predicts runtimes for each of the portfolio solvers, and executes the one with the lowest predicted runtime. Subsequent developments have focused on improved performance in real-world applications (e.g., by avoiding the cost of feature extraction for particularly easy instances, and by improving robustness to unstable target algorithms), alternative selection models (based on both classification and regression), and applications to other domains (mixed-integer programming). In Chapter 6 we generalize and completely automate the design procedure used to build SATZILLA, making it available for arbitrary problem domains.

The *ZILLA procedure is based on the original MATLAB SATZILLA implementation, so it requires an appropriate version of MATLAB or the MATLAB MCR be installed. The default build of the *ZILLA plugin uses MATLAB 7.10 (R2010a) for Linux, as this is the license that was available to us during development. While the *ZILLA plugin is not distributed with the MCR, we are able to provide it upon request.

*ZILLA currently only supports performance metric evaluators that are distributive over addition; i.e. score(run A then run B) = score(run A) + score(run B). This is true of time-like evaluators, but is only true of solution quality evaluators if the solution quality of a feature extractor is zero or a failed run is zero. Also, its support for pre-solver selection should usually be disabled for use with solution quality metrics.

Inputs/outputs of the *ZILLA procedure are listed in Table A.7.

A.4.5 HYDRA-*

HYDRA [83, 84] is a per-instance configuration procedure that, in its initial implementation, iteratively used PARAMILS to design new parameterless algorithms for addition to a SATZILLA selec-

tion portfolio. HYDRA adds one or more solvers per iteration to its vector of portfolio candidates $\mathbf{a}$ by configuring parameterized algorithm $\mathscr{A}$ to optimize performance on instance set S according to a dynamic performance metric that estimates net contribution to the current portfolio. In this way, the Hydra procedure is able to achieve robust performance without any *a priori* knowledge about the instance distribution other than a suspicion of heterogeneity. The HYDRA-$*$ plugin for HAL is a pure Java implementation of HYDRA that can work with arbitrary procedures for algorithm configuration and portfolio-based selection in order to automatically solve the per-instance configuration problem in a domain-independent manner.

Inputs/outputs of the HYDRA-$*$ procedure are listed in Table A.8.

Table A.1: Inputs/outputs of the SCD-Based Analysis procedure for HAL 1.1. The **bold** description text is displayed in the UI.

	domain	*default*	*description*
Parameters			*– none –*
Scenario Settings			
MAX_CPUTIME	$[0,\infty) \subset \mathbb{R}$	10^{300}	**Run time limit (CPU s)** for the entire experiment
SEED	$[1,2^{31}-1] \subset \mathbb{N}$	1	**Seed** used to initialize random number generator
maxParallelRuns	$[0,2^{63}-1] \subset \mathbb{N}$	100	**Maximum runs to launch in parallel** (number of simultaneous target algorithm runs); subject to parallelization support in the environment used.
MAX_RUNLENGTH	$[0,2^{63}-1] \subset \mathbb{N}$	10^{18}	The maximum total number of target algorithm runs
combineIdenticalInstances	[true, false]	true	Whether or not to **collapse instances with identical hash**. Some benchmark sets may inadvertently contain duplicate instances. If this setting is set to true, then HAL will only analyse the algorithm for each *u*nique instance; otherwise, it will analyse for each *l*isted instance which may skew results.
minOutputInterval	$[0,\infty) \subset \mathbb{R}$	60	The **minimum logged output interval (s)** at which intermediate outputs will be updated in the database. Final outputs will always be written to the database regardless of this setting.
Other Settings			
MAX_SUBRUN_RUNLENGTH	$[1,2^{63}-1] \subset \mathbb{N}$	10^{18}	**Subrun runlength limit (steps)**, used to cap target algorithm runs
MAX_SUBRUN_CPUTIME	$[0,\infty) \subset \mathbb{R}$	60	**Subrun time limit (CPU s)**, used to cap target algorithm runs
RUNS_PER_INSTANCE	$[1,2^{63}-1] \subset \mathbb{N}$	1	**Subruns per instance**, for analysing randomized algorithms. Note that this is ignored if the target algorithm is deterministic.
Outputs			
SCD	*Plot*		**Solution cost distribution** of performance according to the chosen metric
SOLUTION_QUALITY	$\mathbb{R}$		**Solution quality per provided metric** across per-instance performance scores
MEAN	$\mathbb{R}$		**Mean performance** across per-instance performance scores
STDDEV	$\mathbb{R}$		**Performance standard deviation** across per-instance performance scores
Q90	$\mathbb{R}$		**90th performance percentile** across per-instance performance scores
Q75	$\mathbb{R}$		**75th performance percentile** across the per-instance performance scores
Q50	$\mathbb{R}$		**Median performance** across per-instance performance scores
Q25	$\mathbb{R}$		**25th performance percentile** across the per-instance performance scores
Q10	$\mathbb{R}$		**10th performance percentile** across the per-instance performance scores
RUNLENGTH	$[0,2^{63}-1] \subset \mathbb{N}$		number of target algorithm runs completed

Table A.2: Inputs/outputs of the Comprehensive Pairwise Comparison procedure for HAL 1.1. The **bold** description text is displayed in the UI.

	domain	*default*	*description*
Parameters			*– none –*
Scenario Settings			
MAX_CPUTIME	$[0,\infty) \subset \mathbb{R}$	10^{300}	**Run time limit (CPU s)** for the entire experiment
SEED	$[1, 2^{31}-1] \subset \mathbb{N}$	1	**Seed** used to initialize random number generator
maxParallelRuns	$[0, 2^{63}-1] \subset \mathbb{N}$	100	**Maximum runs to launch in parallel** (number of simultaneous target algorithm runs); subject to parallelization support in the environment used.
MAX_RUNLENGTH	$[0, 2^{63}-1] \subset \mathbb{N}$	10^{18}	The maximum total number of target algorithm runs
combineIdenticalInstances	[true, false]	true	Whether or not to **collapse instances with identical hash**. Some benchmark sets may inadvertently contain duplicate instances. If this setting is set to true, then HAL will only analyse the algorithm for each *u*nique instance; otherwise, it will analyse for each *l*isted instance which may skew results.
minOutputInterval	$[0,\infty) \subset \mathbb{R}$	60	The **minimum logged output interval (s)** at which intermediate outputs will be updated in the database. Final outputs will always be written to the database regardless of this setting.
Other Settings			
MAX_SUBRUN_RUNLENGTH	$[1, 2^{63}-1] \subset \mathbb{N}$	10^{18}	**Subrun runlength limit (steps)**, used to cap target algorithm runs
MAX_SUBRUN_CPUTIME	$[0,\infty) \subset \mathbb{R}$	60	**Subrun time limit (CPU s)**, used to cap target algorithm runs
RUNS_PER_INSTANCE	$[1, 2^{63}-1] \subset \mathbb{N}$	1	**Subruns per instance**, for analysing randomized algorithms. Note that this is ignored if the target algorithm is deterministic.
Outputs			
SCD_OVERLAY	*Plot*		**Solution cost distributions** of performance according to the chosen metric; algorithms overlaid
SCATTER	*Plot*		**Correlation plot** of performance according to the chosen metric for the two algorithms
WILCOXON_WINNER	*ParameterlessAlgorithm*		Target algorithm indicated by Wilcoxon test as having higher performance
WILCOXON_W	$[0, 2^{63}-1] \subset \mathbb{N}$		Wilcoxon signed-rank test W statistic
WILCOXON_P	$[0,1] \subset \mathbb{R}$		p-value indicating statistical significance of Wilcoxon W
SPEARMAN_RHO	$[-1,1] \subset \mathbb{R}$		ρ; Spearman correlation statistic
SPEARMAN_P	$[0,1] \subset \mathbb{R}$		p-value indicating statistical significance of Spearman ρ
SCD_1	*Plot*		**Alg. 1 solution cost distribution** of performance according to the chosen metric
SOLUTION_QUALITY_1	$\mathbb{R}$		**Alg. 1 solution quality per provided metric** across per-instance performance scores
MEAN_1	$\mathbb{R}$		**Alg. 1 mean performance** across per-instance performance scores
STDDEV_1	$\mathbb{R}$		**Alg. 1 performance standard deviation** across per-instance performance scores

(continued on next page...)

Table A.2 – continued

Q90_1	$\mathbb{R}$	**Alg. 1 90th performance percentile** across per-instance performance scores
Q75_1	$\mathbb{R}$	**Alg. 1 75th performance percentile** across the per-instance performance scores
Q50_1	$\mathbb{R}$	**Alg. 1 median performance** across per-instance performance scores
Q25_1	$\mathbb{R}$	**Alg. 1 25th performance percentile** across the per-instance performance scores
Q10_1	$\mathbb{R}$	**Alg. 1 10th performance percentile** across the per-instance performance scores
SCD_2	*Plot*	**Alg. 2 solution cost distribution** of performance according to the chosen metric
SOLUTION_QUALITY_2	$\mathbb{R}$	**Alg. 2 solution quality per provided metric** across per-instance performance scores
MEAN_1	$\mathbb{R}$	**Alg. 2 mean performance** across per-instance performance scores
STDDEV_2	$\mathbb{R}$	**Alg. 2 performance standard deviation** across per-instance performance scores
Q90_2	$\mathbb{R}$	**Alg. 2 90th performance percentile** across per-instance performance scores
Q75_2	$\mathbb{R}$	**Alg. 2 75th performance percentile** across the per-instance performance scores
Q50_2	$\mathbb{R}$	**Alg. 2 median performance** across per-instance performance scores
Q25_2	$\mathbb{R}$	**Alg. 2 25th performance percentile** across the per-instance performance scores
Q10_2	$\mathbb{R}$	**Alg. 2 10th performance percentile** across the per-instance performance scores
RUNLENGTH	$[0, 2^{63} - 1] \subset \mathbb{N}$	number of target algorithm runs completed

Table A.3: Inputs/outputs of the Comprehensive k-way Comparison procedure for HAL 1.1. The **bold** description text is displayed in the UI.

	domain	*default*	*description*
Parameters			*– none –*
Scenario Settings			
MAX_CPUTIME	$[0,\infty) \subset \mathbb{R}$	10^{300}	**Run time limit (CPU s)** for the entire experiment
SEED	$[1, 2^{31}-1] \subset \mathbb{N}$	1	**Seed** used to initialize random number generator
SIGNIFICANCE	$[0,1] \subset \mathbb{R}$	0.05	**Significance level** α used to interpret the Friedman and Wilcoxon tests
maxParallelRuns	$[0, 2^{63}-1] \subset \mathbb{N}$	100	**Maximum runs to launch in parallel** (number of simultaneous target algorithm runs); subject to parallelization support in the environment used.
MAX_RUNLENGTH	$[0, 2^{63}-1] \subset \mathbb{N}$	10^{18}	The maximum total number of target algorithm runs
combineIdenticalInstances	[true, false]	true	Whether or not to **collapse instances with identical hash**. Some benchmark sets may inadvertently contain duplicate instances. If this setting is set to true, then HAL will only analyse the algorithm for each *u*nique instance; otherwise, it will analyse for each *l*isted instance which may skew results.
minOutputInterval	$[0,\infty) \subset \mathbb{R}$	60	The **minimum logged output interval (s)** at which intermediate outputs will be updated in the database. Final outputs will always be written to the database regardless of this setting.
Other Settings			
MAX_SUBRUN_RUNLENGTH	$[1, 2^{63}-1] \subset \mathbb{N}$	10^{18}	**Subrun runlength limit (steps)**, used to cap target algorithm runs
MAX_SUBRUN_CPUTIME	$[0,\infty) \subset \mathbb{R}$	60	**Subrun time limit (CPU s)**, used to cap target algorithm runs
RUNS_PER_INSTANCE	$[1, 2^{63}-1] \subset \mathbb{N}$	1	**Subruns per instance**, for analysing randomized algorithms. Note that this is ignored if the target algorithm is deterministic.
Outputs			
SCD_OVERLAY	*Plot*		**Solution cost distributions** of performance according to the chosen metric; algorithms overlaid
FRIEDMAN_Q	$[0, 2^{63}-1] \subset \mathbb{N}$		Friedman test Q statistic
FRIEDMAN_P	$[0,1] \subset \mathbb{R}$		p-value indicating statistical significance of Friedman Q
WILCOXON_WINNERS	$ParameterlessAlgorithm^m;\ m \leq k$		Target algorithms having performance indistinguishable from the best, per Wilcoxon test
WILCOXON_WS	$[0, 2^{63}-1] \subset \mathbb{N}$		Per-algorithm Wilcoxon signed-rank test W statistics
WILCOXON_PS	$[0,1] \subset \mathbb{R}$		p-values indicating statistical significance of Wilcoxon Ws
SOLUTION_QUALITIES	$\mathbb{R}^k$		**Per-algorithm solution quality per provided metrics** across per-instance performance scores
MEANS	$\mathbb{R}^k$		**Per-algorithm mean performances** across per-instance performance scores
STDDEVS	$\mathbb{R}^k$		**Per-algorithm performance standard deviations** across per-instance performance scores
Q90S	$\mathbb{R}^k$		**Per-algorithm 90th performance percentiles** across per-instance performance scores

(continued on next page…)

Table A.3 – continued

Q75S	$\mathbb{R}^k$	**Per-algorithm 75th performance percentiles** across the per-instance performance scores
Q50S	$\mathbb{R}^k$	**Per-algorithm median performances** across per-instance performance scores
Q25S	$\mathbb{R}^k$	**Per-algorithm 25th performance percentiles** across the per-instance performance scores
Q10S	$\mathbb{R}^k$	**Per-algorithm 10th performance percentiles** across the per-instance performance scores
RUNLENGTHS	$[0, 2^{63}-1]^k \subset \mathbb{N}^k$	number of runs completed for each target algorithm

Table A.4: Inputs/outputs of the ParamILS procedure for HAL 1.1. The **bold** description text is displayed in the UI.

	domain	*default*	*description*
Parameters			*– none –*
Scenario Settings			
domainDiscretizationFactor	$[0, 2^{63} - 1] \subset \mathbb{N}$	5	Number of discrete values to discretize each continuous domain into. Note that large discrete valued domains are not re-discretized.
MAX_CPUTIME	$[0, \infty) \subset \mathbb{R}$	86400	**Run time limit (CPU s)** for the entire experiment
SEED	$[1, 2^{31} - 1] \subset \mathbb{N}$	1	**Seed** used to initialize random number generator
domainDiscretizationFactor	$[0, 2^{63} - 1] \subset \mathbb{N}$	5	Number of discrete values to discretize each continuous domain into. Note that large discrete valued domains are not re-discretized.
maxEvalsPerConfig	$[0, 2^{63} - 1] \subset \mathbb{N}$	2000	The maximum number of runs per candidate configuration
MAX_RUNLENGTH	$[0, 2^{63} - 1] \subset \mathbb{N}$	10^{18}	The maximum total number of target algorithm runs
resampleInstances	[true, false]	true	Whether to resample instances (with replacement) when constructing an ordered list of (instance, seed) pairs for training. If false, order will follow iteration order of provided *InstanceDistribution*
trainingSetSize	$[0, 2^{63} - 1] \subset \mathbb{N}$	2000	The number of distinct (instance, seed) pairs to use as a training set for configuration
stopOnTies	[true, false]	true	Whether to discard candidate parameter settings that initially tie the current incumbent in terms of performance, or to continue investigating on further instances. Note that parameter settings that are initially outperformed by the incumbent are always discarded. Useful for parameter spaces with many plateaus.
Other Settings			
MAX_SUBRUN_RUNLENGTH	$[1, 2^{63} - 1] \subset \mathbb{N}$	10^{18}	**Subrun runlength limit (steps)**, used to cap target algorithm runs
MAX_SUBRUN_CPUTIME	$[0, \infty) \subset \mathbb{R}$	60	**Subrun time limit (CPU s)**, used to cap target algorithm runs
Outputs			
DESIGN	*ParameterlessAlgorithm*		The algorithm design found by the design procedure
SOLUTION_QUALITY	$\mathbb{R}$		Estimated solution quality of the output design on the training set
incumbentRuns	$[0, \infty) \subset \mathbb{R}$		number of runs used to estimate output design's performance
incumbentTime	$[0, \infty) \subset \mathbb{R}$		total CPU time spent running the output design
incumbentCutoff	$[0, \infty) \subset \mathbb{R}$		runtime cutoff associated with the output design

Table A.5: Inputs/outputs of the GGA procedure for HAL 1.1. The **bold** description text is displayed in the UI.

	domain	*default*	*description*
Parameters			
numRunsStart	$[1,2^{31}-1] \subset \mathbb{N}$	5	number of target algorithm runs per individual at generation 0
populationSize	$[1,2^{31}-1] \subset \mathbb{N}$	100	number of candidate parameter configurations in the population
numGenerations	$[1,2^{31}-1] \subset \mathbb{N}$	100	number of iterations to run the genetic algorithm for
numRunsEnd	$[1,2^{31}-1] \subset \mathbb{N}$	0	number of target algorithm runs per individual at last generation; 0 is automatic
Scenario Settings			
MAX_CPUTIME	$[0,\infty) \subset \mathbb{R}$	86400	**Run time limit (CPU s)** for the entire experiment
SEED	$[1,2^{31}-1] \subset \mathbb{N}$	0	**Seed** used to initialize random number generator
resampleInstances	[true, false]	true	Whether to resample instances (with replacement) when constructing an ordered list of (instance, seed) pairs for training. If false, order will follow iteration order of provided *InstanceDistribution*
trainingSetSize	$[0,2^{63}-1] \subset \mathbb{N}$	2000	The number of distinct (instance, seed) pairs to use as a training set for configuration
Other Settings			
MAX_SUBRUN_CPUTIME	$[0,\infty) \subset \mathbb{R}$	60	**Subrun time limit (CPU s)**, used to cap target algorithm runs
Outputs			
DESIGN	*ParameterlessAlgorithm*		The algorithm design found by the design procedure
SOLUTION_QUALITY	$\mathbb{R}$		Estimated performance (per provided metric) of the output design on the training set
RUNLENGTH	$[0,\infty) \subset \mathbb{R}$		total number of target algorithm runs conducted
incumbentTime	$[0,\infty) \subset \mathbb{R}$		total CPU time spent running the output design

Table A.6: Inputs/outputs of the ROAR procedure for HAL 1.1. The **bold** description text is displayed in the UI.

	domain	*default*	*description*
Parameters			*– none –*
Scenario Settings			
MAX_CPUTIME	$[0, \infty) \subset \mathbb{R}$	86400	**Run time limit (CPU s)** for the entire experiment
SEED	$[1, 2^{31} - 1] \subset \mathbb{N}$	0	**Seed** used to initialize random number generator
Other Settings			
MAX_SUBRUN_CPUTIME	$[0, \infty) \subset \mathbb{R}$	60	**Subrun time limit (CPU s)**, used to cap target algorithm runs
Outputs			
DESIGN	*ParameterlessAlgorithm*		The algorithm design found by the design procedure
SOLUTION_QUALITY	$\mathbb{R}$		Estimated performance (per provided metric) of the output design on the training set

Table A.7: Inputs/outputs of the ∗ZILLA procedure for HAL 1.1

	domain	*default*	*description*
Parameters			*– none –*
Scenario Settings			
MAX_CPUTIME	$[0,\infty) \subset \mathbb{R}$	10^{300}	Run time limit (CPU s) for the entire experiment
SEED	$[1,2^{31}-1] \subset \mathbb{N}$	1	Seed used to initialize random number generator
NumCrossValidation	$[0,2^{63}-1] \subset \mathbb{N}$	10	The number of cross-validation runs to use when training the selector model
MaxPreTime	$[0,1] \subset \mathbb{R}$	0.1	The maximum fraction of the total runtime budget to consider running presolvers before performing selection
MaxFeatureTime	$[0,1] \subset \mathbb{R}$	0.1	The maximum fraction of the total runtime budget to spend extracting features before reverting to the backup solver
minTargetRuntime	$[0,\infty) \subset \mathbb{R}$	0.005	All target runs with evaluation lower than this get this score
DoPresolverSelection	[true, false]	false	Whether or not to do investigate running presolvers before performing selection. This can improve performance of the design but is compute-intensive (true is over 100× slower than false), and is not appropriate for many solution quality metrics.
PredictLogRuntime	[true, false]	true	Whether to predict log performance instead of performance; true is appropriate for runtime in most situations.
maxParallelRuns	$[0,2^{63}-1] \subset \mathbb{N}$	100	Maximum runs to launch in parallel (number of simultaneous target algorithm runs)
Predictor	*String*	"LinearRegression"	Chooses the type of model used to perform selection. Options are "LinearRegression", "DecisionTree", and "DecisionForest"
Cost	*String*	"RAW"	Transformation to use on raw performance scores, for decision trees/forests only. Options include "RAW", "UNIFORM", "SQRT"
NumTrees	$[0,2^{63}-1] \subset \mathbb{N}$	99	Number of trees for decision forests
VariablesPerDecison	$[0,2^{63}-1] \subset \mathbb{N}$	0	number of features to consider in each tree, for random forests only. 0 is auto.
MaxModelSize	$[1,2^{63}-1] \subset \mathbb{N}$	20	Maximum model size for linear regression model.
DoQuadraticExpansion	[true, false]	true	Whether or not to do quadratic expansion of features in linear regression
LinearModelSize	$[1,2^{63}-1] \subset \mathbb{N}$	30	Default linear regression model size.
Other Settings			
MAX_SUBRUN_CPUTIME	$[0,\infty) \subset \mathbb{R}$	60	Subrun time limit (CPU s), used to cap target algorithm runs
Outputs			
DESIGN	*ParameterlessAlgorithm*		The algorithm design found by the design procedure

(continued on next page. . .)

Table A.7 – continued

SOLUTION_QUALITY	$\mathbb{R}$	Estimated performance (per provided metric) of the output design on the training set
oracleQuality	$\mathbb{R}$	Estimated performance (per provided metric) of the best possible selector on the training set
predictedInstanceSelections	$\mathbb{N}^{\lvert S \rvert}$	Index of the algorithm that would be selected for each of the training instances, or *null* if it would be solved by a presolver/backup solver.
predictedInstanceEvaluations	$Evaluation^{\lvert S \rvert}$	The predicted evaluation for each of the training instances, if the selector was run on it
componentQuality	$\mathbb{R}^{\lvert \mathbf{a} \rvert}$	Estimated performance (per provided metric) of each of the portfolio components
featureCost	$\mathbb{R}$	Estimated performance cost of extracting features on the training instance distribution, per provided metric.

Table A.8: Inputs/outputs of the HYDRA-$*$ procedure for HAL 1.1

	domain	default	description
Parameters			*– none –*
Scenario Settings			
MAX_CPUTIME	$[0,\infty) \subset \mathbb{R}$	10^{300}	Run time limit (CPU s) for the entire experiment
SEED	$[1, 2^{31}-1] \subset \mathbb{N}$	1	Seed used to initialize random number generator
maxIterations	$[0, 2^{63}-1] \subset \mathbb{N}$	5	The maximum number of iterations to run
tuningRunsPerIteration	$[0, 2^{63}-1] \subset \mathbb{N}$	5	The maximum number of iterations to run
solversPerIteration	$[0, 2^{63}-1] \subset \mathbb{N}$	5	The maximum number of iterations to run
maxTunerCPUTime	$[0,\infty) \subset \mathbb{R}$	10^{300}	Run time limit (CPU s) for each algorithm configuration run
maxTuningSolverCPUTime	$[0,\infty) \subset \mathbb{R}$	10^{300}	Run time limit (CPU s) for solvers during each algorithm configuration run
maxPBSSolverCPUTime	$[0,\infty) \subset \mathbb{R}$	10^{300}	Run time limit (CPU s) for solvers during each selector builder run
includeDefault	[true, false]	true	Whether or not to include the default configuration as a portfolio candidate
configurationProcedure	$ConfiguratorImplementation$		The algorithm configuration procedure to use.
selectorBuildingProcedure	$PortfolioBasedSelectorBuilder$		The portfolio-based selection procedure to use.
Other Settings			*– none –*
Outputs			
DESIGN	*ParameterlessAlgorithm*		The algorithm design found by the design procedure
SOLUTION_QUALITY	$\mathbb{R}$		Estimated performance (per provided metric) of the output design on the training set
oracleQuality	$\mathbb{R}$		Estimated performance (per provided metric) of the best possible selector on the training set
componentQuality	$\mathbb{R}^{\lvert\mathbf{a}\rvert}$		Estimated performance (per provided metric) of each of the portfolio components
featureCost	$\mathbb{R}$		Estimated performance cost of extracting features on the training instance distribution, per provided metric.
RUNLENGTH	$[0,\infty) \subset \mathbb{R}$		total number of iterations completed

Appendix B

HAL 1.1 Developer Reference

This reference is designed to serve as a technical resource for developers working with the HAL API. It is not intended as detailed low-level documentation of the kind one might expect from Javadocs; nor is it intended to provide a complete high-level overview of the system (for that, see Chapter 4.2). Instead, we explicitly highlight the interfaces (and where important, concrete implementations) that a developer will typically interact with when performing common tasks using the HAL 1.1 API. In doing so, we aim to demonstrate in a practical manner how the major subsystems of HAL are implemented and used.

We first refer the reader to Chapter 4.2 of the main text; in particular, Figure 4.1 illustrates the composition of HAL's major subsystems. We then suggest reviewing the sections of this appendix in the order they appear. The documentation presented here focuses on interactions between components of the Experiment Modelling subsystem (Chapter 4.2.1), and of the Execution and Data Management subsystem (Chapter 4.2.2), as implemented in HAL 1.1. Diagrams were generated from HAL 1.1 source; all examples are included in the HAL 1.1 source distribution.

B.1 Tutorial: Developing Meta-algorithms with HAL

Perhaps the most common development task undertaken in HAL is the implementation of a new meta-algorithmic procedure. While Chapter 6 describes the HAL plugin for HYDRA-$*$, many fundamental tasks in its implementation are delegated to component meta-algorithms for configuration and portfolio-based selection. Rather than describing the HYDRA-$*$ implementation in detail here, we instead investigate a simple single-algorithm analysis meta-algorithm that provides a more explicit example of fundamental HAL tasks and programmatic idioms. The concepts explored in this example apply to all of the meta-algorithmic plugins provided for HAL 1.1, and we encourage the interested reader to view the source codes for one or more of these after working through this

tutorial.

The tutorial is divided into three subsections. First, we design and implement a simple algorithm analysis procedure; second, we verify this procedure by writing automated unit tests; and third, we improve our procedure's parallel performance and extend its functionality. When working though the first subsection, the reader will develop familiarity with the fundamental components of HAL's experiment modelling subsystem as they apply to meta-algorithms. Target algorithms and problem instances are explored in the second subsection, as is the relationship between meta-algorithms and execution and data management components. Advanced usage of these HAL components is discussed in the third subsection.

The goal of this tutorial is to construct a single-algorithm analysis meta-algorithm that investigates whether a randomized target algorithm's performance on an instance distribution is independent of the seed it is provided. In terms of the single-algorithm analysis problem it solves, we seek to implement:

Inputs: $\langle a, I, m \rangle$, where:

a is a parameterless randomized target algorithm

I is a set of target problem instances

m is a performance metric

Outputs: $\langle \mathbf{s}, \mathbf{t} \rangle$, where:

$\mathbf{s}$ is a two-dimensional vector of scalars where s_0 and s_1 are the test statistic and p-value respectively of a Friedman test assessing whether the empirical solution cost distribution of a on I according to m is drawn from the same underlying distribution regardless of random seed

$\mathbf{t}$ is a one-dimensional vector of plots where t_0 is an overlay of empirical solution cost distributions for a on I according to m using different seeds

B.1.1 Implementation Basics

In keeping with a first-example tutorial, we begin by implementing a very simple meta-algorithm that returns a performance metric's evaluation of a target algorithm run on a set of problem instances. In particular, our initial procedure will (implicitly) implement the following output specification:

Outputs: $\langle \mathbf{s}, \mathbf{t} \rangle$, where:

$\mathbf{s}$ is a one-dimensional vector; s_0 is the performance of a on I according to m

$\mathbf{t}$ is an empty vector of plots

In Section B.1.3 we modify this implementation to achieve our original goal.

Recall from Section 4.2 that meta-algorithmic procedures correspond to *MetaAlgorithmImplementations*, which in turn are (abstract) subclasses of *AlgorithmImplementations*; this relationship is illustrated in Figure B.2. In HAL, a *MetaAlgorithmImplementation* describes the inputs and outputs of a meta-algorithmic procedure, and acts as a factory for instantiating *MetaAlgorithmRun* objects that actually execute the procedure on individual *MetaProblemInstances*.

We begin by creating a new *MetaAlgorithmImplementation* subclass called *ExampleMetaAlgorithm-Implementation*. For reasons discussed in Section B.1.1.1, we also implement the *ImmutableJson-Serializable* interface. We finally create an enclosed *MetaAlgorithmRun* subclass called *Example-MetaAlgorithmRun*. Doing this using Eclipse yields the following stub code, which we proceed to flesh out.

```
public class ExampleMetaAlgorithmImplementation extends
    MetaAlgorithmImplementation implements
    ImmutableJsonSerializable {

  @Override
  public Set<Set<String>> getRequiredTags() {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  public JSONObject buildSpec() {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  public MetaAlgorithmRun getRun(AlgorithmRunRequest request,
      SubrunRunner runner, ReadOnlyDataManager datamanager,
      Statistics stats) {
    // TODO Auto-generated method stub
    return null;
  }

  private class ExampleMetaAlgorithmRun extends MetaAlgorithmRun {
    @Override
    public double getFractionCompleted() {
      // TODO Auto-generated method stub
      return 0;
    }
    @Override
    protected void start() {
      // TODO Auto-generated method stub
    }
  }
}
```

B.1.1.1 The *MetaAlgorithmImplementation* constructor

Our *ExampleMetaAlgorithmImplementation* requires a constructor, which in turn must call a superclass constructor with several arguments:

Type	Name
String	name
String	version
Map⟨String, Object⟩	properties
ParameterSpace	defaultConfigurationSpace
ParameterSpace	defaultScenarioSpace
ParameterSpace	defaultOutputSpace
ParameterSpace	supportedOptionSpace

We use the class name and version 1.0.0 to identify our procedure, and define a map that specifies values for the properties introduced in Appendix A: our meta-algorithm is not *deterministic*, is *exportable*, and is *cutoff-agnostic*. Keys for these and other semantics are defined in the *Semantics* class. In code:

```
private static final String NAME = ExampleMetaAlgorithmImplementation.class.getSimpleName
    ();
private static final String VERSION = "1.0.0";
private static final Map<String, Object> PROPERTIES = Collections.unmodifiableMap(Misc.
    asMap(
      Semantics.DETERMINISTIC, (Object) false,
      Semantics.EXPORTABLE, true,
      Semantics.CUTOFF_AGNOSTIC, true
    ));
```

We proceed to specify the four *ParameterSpaces* that collectively describe input and output domains for our meta-algorithm. *ParameterSpaces* in HAL 1.1 are immutable, so we use a mutable *ParameterSpaceBuilder*. *ParameterSpaces*, *ParameterSpaceBuilders*, and *Domains* are illustrated in Figure B.3, and the various *Domains* supported in HAL 1.1 are illustrated in Figure B.4.

Some of the inputs and outputs of our meta-algorithm correspond to well-defined semantics that are important to identify. Global semantics and associated default domains are identified using fields in the *Semantics* class; additional problem-specific semantics may also be defined in corresponding *Problem* classes. For example, HAL needs to know which input corresponds to a CPU time limit; this semantic is associated with *Semantics.MAX_CPUTIME*, and its default domain can be obtained with *Semantics.getDomain(Semantics.MAX_CPUTIME)*. Standardized HAL semantics are summarized in Table B.1.

As our meta-algorithm has no configurable parameters, the first space is empty. We allow users to specify a random seed, a maximum CPU time, and a maximum number of target runs; these

Context	Symbol	Dflt. Domain	Semantics
scenario	MAX_CPUTIME	$\mathbb{R}^+$	run time limit (CPU seconds)
	MAX_RUNLENGTH	$\mathbb{Z}^+$	run length limit (steps)
	SEED	$[1, 2^{32}-1] \subset \mathbb{Z}$	random seed
	P_LEVEL	$[0,1] \subset \mathbb{R}$	target significance level (α)
output	CPUTIME	$\mathbb{R}^+$	total run time (CPU seconds)
	RUNLENGTH	$\mathbb{Z}^+$	total run length (steps)
	TIMEOUT	{true, false}	terminated due to run time/length limit
	P_VALUE	$[0,1] \subset \mathbb{R}$	observed significance level
	SOLUTION_QUALITY	$\mathbb{R}$	solution quality; lower is better
	DESIGN	*Algorithm*	design produced by design procedure
instance	INSTANCE_FILE	*File*	file containing target problem instance
algorithm properties	DETERMINISTIC	{true, false}	deterministic
	EXPORTABLE	{true, false}	can be zipped and used on new hosts
	CUTOFF_AGNOSTIC	{true, false}	short runs are inferable from long runs
meta-instance options	MAX_SUBRUN_CPUTIME	$\mathbb{R}^+$	target run time limit
	MAX_SUBRUN_RUNLENGTH	$\mathbb{Z}^+$	target run length limit
	RUNS_PER_INSTANCE	$\mathbb{Z}^+$	independent runs of randomized targets

Table B.1: Pre-defined input and output semantics in HAL 1.1. ‘Context’ refers to the *Space* the semantic is associated with.

settings correspond to the second space. We specify our procedure’s sole output (the metric evaluation) in the third space, and assign it *solution quality* semantics. Finally, the fourth space indicates options embedded in a meta-problem instance that our meta-algorithm can accept; we indicate the ability to interpret maximum per-target-run CPU time and run length limits, as well as a maximum number of runs per-target-instance. Again, in Java code:

```
private static final ParameterSpace SCN_SPACE, CFG_SPACE, OUT_SPACE, SUPP_OPT_SPACE;
static {
  ParameterSpaceBuilder psb = new ParameterSpaceBuilder();
  // set up default configuration space -- no configurable params
  CFG_SPACE = psb.build();

  // set up default scenario space
  psb.put(Semantics.SEED, new IntegerDomain(0, null));
  psb.put(Semantics.MAX_CPUTIME, Semantics.getDomain(Semantics.MAX_CPUTIME));
  psb.put("maxTargetRuns", Semantics.getDomain(Semantics.MAX_RUNLENGTH));
  psb.addAlias(Semantics.MAX_RUNLENGTH, "maxTargetRuns");
  SCN_SPACE = psb.build();

  // set up default output space -- only output is metric value
  psb.clear();
  psb.put("metricScore", Semantics.getDomain(Semantics.SOLUTION_QUALITY));
  psb.addAlias(Semantics.SOLUTION_QUALITY, "metricScore");
  OUT_SPACE = psb.build();
```

```
  // set up supported instance options
  psb.clear() ;
  psb.put(Semantics.MAX_SUBRUN_CPUTIME, Semantics.getDomain(Semantics.MAX_CPUTIME));
  psb.put(Semantics.MAX_SUBRUN_RUNLENGTH, Semantics.getDomain(Semantics.MAX_RUNLENGTH));
  psb.put(Semantics.RUNS_PER_INSTANCE, Semantics.getDomain(Semantics.RUNS_PER_INSTANCE));
  SUPP_OPT_SPACE = psb.build();
}
```

Finally, we define a simple nullary constructor:

```
public MetricEvaluatingMetaAlgorithmImplementation() {
  super(NAME, VERSION, PROPERTIES, CFG_SPACE,
      SCN_SPACE, OUT_SPACE, SUPP_OPT_SPACE);
}
```

B.1.1.2 *MetaAlgorithmImplementation* methods

Having completed the constructor, we must implement the remaining methods defined in the *MetaAlgorithmImplementation* class. In particular, three instance methods remain unimplemented by superclasses — *getRequiredTags()*, *buildSpec()*, and *getRun(⋯)* — and by HAL 1.1 convention all *JsonSerializable* objects must implement the static deserialization method *fromSpec(String)*.

AlgorithmImplementation.getRequiredTags() is used by the framework to infer what kinds of instances an algorithm can accept. The $Set\langle Set\langle String\rangle\rangle$ it returns indicates compatibility with any instance exhibiting *all* of the tags from *at least one* of the contained $Set\langle String\rangle$ objects. We require an input *MetaProblemInstance* that contains target instances and a performance metric; these are available from any *InstanceMetricMetaProblemInstance*. Further, we require a single target algorithm. Tags indicating these and other requirements can be found in the *MetaProblemInstance* class and its subclasses; Java code for indicating these requirements is:

```
private static final Set<Set<String>> TAGS = Misc.asSet(
    Misc.asSet( InstanceMetricMetaProblemInstance.TAG,
        MetaProblemInstance.getNumAlgsTag(1) ));
public Set<Set<String>> getRequiredTags() {
  return TAGS;
}
```

JsonSerializable.toSpec() and *fromSpec(String)* are used to serialize and deserialize instances of nearly all HAL 1.1 objects, for example when archiving runs to a database. This serialized form is also used to generate a hash that is used in many equality comparisons. Fortunately, our *MetaAlgorithmImplementation* is in effect an immutable singleton, in that it is stateless and all instances are identical; we have indicated this to HAL by implementing the *ImmutableJsonSerializable* interface. This immutability simplifies serialization and deserialization.

The required *buildSpec()* method is used by a *JsonSerializable.JsonHelper* object in the superclass to construct an appropriate serialization for *toSpec()*. An implementation of *buildSpec()* follows; since our constructor is nullary, the only information to be serialized is the version of our procedure:

```
public JSONObject buildSpec() {
  JSONObject out = super.buildSpec();
  out.put("version", getVersion());
  return out;
}
```

Similarly, our static deserialization method needs only to verify the version string, and then use the nullary constructor. We adopt a versioning scheme of the form *majorVersion.minorVersion.revision*, and establish a compatibility contract between revisions of the same minor version:

```
private static final Pattern p = Pattern.compile("^[^\\s.]+[.][^\\s.]+");
private static boolean isSameMinorVersion(String v1, String v2) {
  Matcher m1 = p.matcher(v1), m2 = p.matcher(v2);
  return m1.find() && m2.find() && m1.group().equals(m2.group());
}
public static MetricEvaluatingMetaAlgorithmImplementation fromSpec(String spec) {
  JSONObject o = JsonHelper.readSpecStub(MetricEvaluatingMetaAlgorithmImplementation.class
      , spec);
  String jsonVersion = o.optString("version");
  if (isSameMinorVersion(VERSION, jsonVersion)) {
    return new MetricEvaluatingMetaAlgorithmImplementation();
  }
  throw new UnsupportedOperationException("Incompatible version; current procedure is " +
      MetricEvaluatingMetaAlgorithmImplementation.VERSION +
      " but specification was " + jsonVersion);
}
```

An important point to note is that HAL 1.1 uses the JSON serialized form of primitive objects – including *MetaAlgorithmImplementations* – to generate strong hash codes used to accelerate identity checking. Since the hash is based on the syntax of the serialization string, and not the semantics of the string, care must be taken to ensure that all keys and values in the JSON object are consistently ordered. In particular, this means that objects from an unordered collection (like a set or a map) should be sorted according to some repeatable criteria before serialization to ensure consistent behaviour.

Finally, we consider *MetaAlgorithmImplementation.getRun(⋯)*. This factory method is used by HAL to obtain a *MetaAlgorithmRun* instance that executes the procedure on specific inputs, and is trivial assuming we have completed implementing the *ExampleMetaAlgorithmRun* class:

```
public MetaAlgorithmRun getRun(AlgorithmRunRequest request, SubrunRunner runner,
    Statistics stats) {
```

```
  return new ExampleMetaAlgorithmRun(request, runner, stats);
}
```

B.1.1.3 The *MetaAlgorithmRun* constructor

Once again, we need a constructor that in turn calls a superclass constructor. In addition, the constructor should ensure fail-fast behaviour for incompatible inputs; this means unpacking inputs from the provided *InstanceMetricMetaProblemInstance* and the specific *ParameterSettings* corresponding to the *ParameterSpaces* we defined earlier. This straightforward process is shown in Java code below. Note that our constructor does *not* perform further computation; actual execution should not begin until the *start()* method is called.

```
private final ParameterlessAlgorithm target;
private final InstanceDistribution instances;
private final Random rng;
private final long maxSubrunLength, maxRuns, runsPerInst;
private final double maxSubrunTime, maxTime;
private final PerformanceMetric<AlgorithmRun> metric;
private final SubrunRunner runner;

public ExampleMetaAlgorithmRun(AlgorithmRunRequest request, SubrunRunner runner) {
  super(request, runner);

  // object used to request target runs
  this.runner = runner;

  // unpack target algorithm and instances
  InstanceMetricMetaProblemInstance inst = (InstanceMetricMetaProblemInstance) request.
      getProblemInstance();
  target = inst.getAlgorithms().get(0);
  instances = inst.getInstanceDistribution();
  metric = inst.getMetric();

  // a random number generator based on the requested seed
  Number n = (Number) request.getScenarioValue(Semantics.SEED);
  long seed = n == null ? (Long) SCN_SPACE.get(Semantics.SEED).getDefaultValue() : n.
      longValue();
  rng = Global.getRandom(seed);

  // read runtime & length limits
  n = (Number) request.getScenarioValue(Semantics.MAX_CPUTIME);
  maxTime = n == null ? (Double) SCN_SPACE.get(Semantics.MAX_CPUTIME).getDefaultValue() :
      n.doubleValue();
  n = (Number) inst.getOption(Semantics.MAX_SUBRUN_CPUTIME);
  maxSubrunTime = n == null ? (Double) SUPP_OPT_SPACE.get(Semantics.MAX_SUBRUN_CPUTIME).
      getDefaultValue() : n.doubleValue();
  n = (Number) inst.getOption(Semantics.MAX_SUBRUN_RUNLENGTH);
  maxSubrunLength = n == null ? (Long) SUPP_OPT_SPACE.get(Semantics.MAX_SUBRUN_RUNLENGTH).
```

```
      getDefaultValue() : n.longValue();
  n = (Number) inst.getOption(Semantics.RUNS_PER_INSTANCE);
  runsPerInst = n == null ? (Long) SUPP_OPT_SPACE.get(Semantics.RUNS_PER_INSTANCE).
      getDefaultValue() : n.longValue();
  n = (Number) request.getScenarioValue(Semantics.MAX_RUNLENGTH);
  maxRuns = Math.min(instances.size() * runsPerInst,
    n == null ? (Long) SCN_SPACE.get(Semantics.MAX_RUNLENGTH).getDefaultValue() : n.
        longValue();
}
```

B.1.1.4 *MetaAlgorithmRun* methods

Of the two methods to be defined, *getFractionCompleted()* is shorter, so we deal with it first. HAL uses this method to gauge the progress of an algorithm run. Since our procedure has (1) a fixed runtime limit, and (2) a fixed maximum number of total runs, it is easy to estimate this progress. In the following code, we assume the volatile *runsDone* field will be updated as execution progresses, and we use the superclass method *getTotalCpuTime()* to obtain the cumulative time spent running our meta-algorithm and its target runs.

```
private volatile int runsDone = 0;
public double getFractionCompleted() {
  double timeFrac = super.getTotalCpuTime() / maxTime;
  double runsFrac = 1.0 * runsDone / maxRuns;
  return Math.max(timeFrac, runsFrac);
}
```

More interesting is the *start()* method, as it is here that we finally implement the logic of our procedure. As a first implementation, we will serially run the target algorithm on each of the target instances, repeating multiple times per instance in the case of a stochastic target algorithm. We will then use the provided metric to evaluate the runs, and report the result as an output. As the method is relatively long, we interleave its implementation with more detailed explanations.

First, we mark the procedure as started using a superclass method and standardized status values from the *RunStatus* class, and set the target algorithm's cutoffs as specified in the meta-algorithm's inputs:

```
protected void start() {
  super.setStatus(RunStatus.RUNNING);
  // set target algorithm cutoffs
  if (target.hasScenarioVariable(Semantics.MAX_CPUTIME)) {
    target.setScenarioValue(Semantics.MAX_CPUTIME, maxSubrunTime);
  }
  if (target.hasScenarioVariable(Semantics.MAX_RUNLENGTH)) {
    target.setScenarioValue(Semantics.MAX_RUNLENGTH, maxSubrunLength);
  }
```

Next, we begin to iterate over instances and seeds, remaining sensitive to alternative termination conditions; namely, performing the maximum number of target runs or receiving an external termination request. Note that if our procedure exceeds its CPU time budget, HAL will automatically request it to terminate by causing *getTerminationStatus()* to return a non-null value and then interrupting the thread executing this method. We must ensure our code is responsive to such a termination request.

```
Map<Object, Map<ProblemInstance, Double>> scores = new HashMap<Object, Map<
    ProblemInstance, Double>>();
OUTERLOOP: for (int i=0; i<runsPerInst; i++) {
  Object seed = ((SampleableDomain<?>)target.getScenarioDomain(Semantics.SEED)).
      getSample(rng);
  target.setScenarioValue(Semantics.SEED, seed);
  for (ProblemInstance instance: instances) {
    // be sensitive to termination conditions
    if (super.getTerminationStatus() != null || runsDone>= maxRuns) {
      break OUTERLOOP;
    }
```

We proceed to configure the target algorithm to solve the next instance, and generate an immutable *AlgorithmRunRequest* that captures its current inputs. Since we are executing runs serially, we disable distributed execution for the request. We then use the *SubrunRunner* that was passed by HAL into the constructor to get an *AlgorithmRun* satisfying the request, and start the retrieved run in a new thread (drawn from a shared thread pool; using this pool both saves overhead from thread creation and also automatically logs any exceptions in the running thread). We wait for the run and any registered visitors to complete before proceeding (more on visitors in Section B.1.3).

```
    target.setProblemInstance(instance);
    AlgorithmRunRequest req = target.getAlgorithmRunRequest();
    req.setDistributedExecutionOK(false);
    AlgorithmRun run = runner.fetchRun(req);
    Global.getThreadPool().execute(run);
    while (!run.visitorsFinished()) {
      try {
        run.waitForVisitors();
      } catch (InterruptedException e) {
        if (super.getTerminationStatus() != null) {
          run.terminate();
        }
      }
    }
```

We use provided the metric to evaluate the completed run, and record the result for future reporting. Finally, we increment the run counter and end the loop.

```
    Double score = metric.evaluate(run);
    if (!scores.containsKey(seed)) {
```

```
        scores.put(seed, new HashMap<ProblemInstance, Double>());
      }
      scores.get(seed).put(instance, score);
      runsDone++;
    }
  }
```

Finally, with all target runs completed, we use the metric to aggregate scores; first across independent runs of each instance, and again across the various instances. We use a superclass method to report this final scalar as an output of our procedure, and mark the meta-algorithm run as finished with the appropriate status. This completes both the method and our initial meta-algorithm implementation.

```
  Map<String, Object> outputs = new HashMap<String, Object>();
  List<Double> aggregatedScores = new LinkedList<Double>();
  for (Map<ProblemInstance, Double> insScores: scores.values()) {
    aggregatedScores.add(metric.aggregate(insScores()));
  }
  outputs.put("metricScore", metric.aggregate(aggregatedScores));
  super.updateOutput(outputs);
  double status = (super.getTerminationStatus() != null) ? super.getTerminationStatus() :
      RunStatus.FINISHED;
  super.setStatus(status);
}
```

B.1.2 Testing

Before deploying our procedure, we must verify that it operates as expected. We achieve this by writing a corresponding unit test suite, in class *ExampleMetaAlgorithmImplementationTest*. Specifically, we test our meta-algorithm when analyzing a target algorithm on an instance set for which behaviour is predictable. To do this, we must set up an execution environment and a meta-problem instance.

B.1.2.1 Target *Algorithms*

For the purposes of this example, we write a target algorithm in our test class which simply reports a runtime in seconds as a randomized function of instance file size b in bytes, via $f(b) = 0.001b + X$ where $X \sim \mathcal{N}(\mu = 0, \sigma = 0.1)$:

```
public static final double SIGMA = 0.1;
public static void main(String[] args) {
  if (args.length != 3) {
    System.err.println("usage: java " + ExampleMetaAlgorithmImplementationTest.class.
        getCanonicalName() + " <instance file> <seed> <maxTime>");
```

```
    System.exit(-1);
  }
  File f = new File(args[0]);
  if (!f.canRead() || !f.isFile()) {
    System.err.println(f + " is not a readable regular file");
    System.exit(-2);
  }
  long seed = Long.valueOf(args[1]);
  double maxtime = Double.valueOf(args[2]);
  double time = Math.max(0, Math.min(0.001*f.length() + SIGMA*new Random(seed).
      nextGaussian(), maxtime));
  System.out.println("Reporting " + time + " s");
}
```

To use our test algorithm as an external, black-box target algorithm in HAL 1.1, we first need to instantiate a corresponding *ExternalAlgorithmImplementation* object, and then instantiate an *Algorithm* object using that implementation. *Algorithms* in HAL 1.1 are illustrated in Figure B.6.

The *ExternalAlgorithmImplementation* constructor has several arguments, some of which are familiar from discussion of *MetaAlgorithmImplementations*:

Type	Name
File	executable
File	workingDirectory
String	name
String	version
Set⟨*Set*⟨*String*⟩⟩	requiredTags
ParameterSpace	instanceSpace
Map⟨*String*, *List*⟨*String*⟩⟩	inputFormat
Map⟨*String*, *List*⟨*String*⟩⟩	outputFormat
Map⟨*String, Object*⟩	properties

The instanceSpace is a parameter space that identifies any instance features required as input to the external algorithm. The input- and outputFormat maps describe invocation and output syntax for the algorithm , and the properties map is also as before. We specify that our algorithm requires external problem instances (so that it can measure file sizes), and specify inputs and outputs as required by the *main(·)* method implemented above:

```
private static ParameterlessAlgorithm getAlgorithm() {
  File exe = new File("java"), workingDir = new File("bin/");
  String name = "unitTestImpl", version = "1.0.0";
  Set<Set<String>> requiredTags = Misc.asSet(Misc.asSet(ExternalProblemInstance.TAG));
  ParameterSpaceBuilder psb = new ParameterSpaceBuilder();
  psb.put(Semantics.INSTANCE_FILE, Semantics.getDomain(Semantics.INSTANCE_FILE));
  ParameterSpace instSpace = psb.build();
  String cmdString = ExampleMetaAlgorithmImplementationTest.class.getCanonicalName() + " $
      " +
```

```
    Semantics.INSTANCE_FILE + "$ $" + Semantics.SEED + "$ $" + Semantics.MAX_CPUTIME + "
        $";
  String outFmt = "Reporting $time$ s";
  Map<String, List<String>> inputFormat = Misc.asMap(Semantics.CALLSTRING, Arrays.asList(
      cmdString));
  Map<String, List<String>> outputFormat = Misc.asMap(Semantics.STDOUT, Arrays.asList(
      outFmt));
  Map<String, Object> properties = Misc.asMap(
      Semantics.EXPORTABLE, (Object)false,
      Semantics.DETERMINISTIC, false,
      Semantics.CUTOFF_AGNOSTIC, true);
  AlgorithmImplementation imp = new ExternalAlgorithmImplementation(
      exe, workingDir, name, version, requiredTags,
      instSpace, inputFormat, outputFormat, properties);
```

We then create a *ParameterlessAlgorithm* using this implementation. To do so, we need to define one *ParameterSpace* for the algorithm's settings (random seed and CPU time budget), and another for its outputs.[1] These are separate from the implementation to allow different spaces be used with the same algorithm. We also name the *ParameterlessAlgorithm*. In Java:

```
  psb.clear();
  psb.put(Semantics.MAX_CPUTIME, Semantics.getDomain(Semantics.MAX_CPUTIME));
  psb.put(Semantics.SEED, Semantics.getDomain(Semantics.SEED));
  ParameterSpace scnSpace = psb.build();
  psb.clear();
  psb.put("time", new RealDomain(0., null));
  psb.setSemanticParameter(Semantics.CPUTIME, "time");
  ParameterSpace outSpace = psb.build();
  return new ParameterlessAlgorithm("unitTestAlg", imp, scnSpace, outSpace);
}
```

B.1.2.2 Target *ProblemInstances*

We also require a set of target instances to run our test instance on. *InstanceDistribution* and *ProblemInstances* in HAL 1.1 are illustrated in Figure B.7. We create an *InstanceList* of dummy *FileProblemInstances*, choosing file sizes to achieve an expected mean reported runtime of 0.75s:

```
private static InstanceList getInstances() throws Exception {
  List<ProblemInstance> il = new LinkedList<ProblemInstance>();
  for (int i=0; i<10; i++) {
    File inst = File.createTempFile("test", "inst");
    inst.deleteOnExit();
    byte[] data = new byte[500+(i+1)*50];
    Arrays.fill(data, (byte)1);
    FileUtils.writeByteArrayToFile(inst, data);
```

[1]Were we defining a *ParameterizedAlgorithm*, we would need one more *ParameterSpace* describing the configurable parameters.

```
    il.add(new FileProblemInstance(inst));
  }
  return new InstanceList("test instances", il);
}
```

B.1.2.3 *Environments*

We also need to specify the environment in which our test should take place. *Environments* in HAL 1.1 are compositions of *ExecutionManagers* and *DataManagers*, as illustrated in Figure B.5. We configure an environment that archives data to a temporary SQLite database, and that executes algorithm runs on the local machine. We also specify that up to 2 target algorithm runs may be executed in parallel.

```
private static Environment getEnvironment() throws Exception {
  // archive to a temporary SQLite database
  File dbfile = File.createTempFile(ExampleMetaAlgorithmImplementationTest.class.
      getCanonicalName(), ".db");
  dbfile.deleteOnExit();
  FullAccessDataManager dm = DataManagerFactory.getDataManager(URI.create("jdbc:sqlite:"+
      dbfile.getPath()));
  // an execution manager which does runs on the local machine, and which echos all output
  LocalExecutionManager localEx = new LocalExecutionManager(OutputHandlingOption.ECHO,
      OutputHandlingOption.ECHO);
  return new Environment("example environment", localEx, dm, Preference.
      MAX_SIMULT_TARGET_RUNS, 2);
}
```

B.1.2.4 Performance Metric

We also must decide on a performance metric for our meta-algorithm to use. We could write an arbitrary new implementation of the *PerformanceMetric* class and use it instead. However, several commonly-used metrics are provided in the PerformanceMetric class, including those listed in Table B.2. For our unit test, we will use *ReportedPAR1* which evaluates runs based on the reported runtime, and aggregates across runs using the mean.

B.1.2.5 Unit test

We now have all we need to complete a unit test. We first construct an *InstanceMetricMetaProblemInstance* that contains our test algorithm and instances, the *PerformanceMetric.ReportedPAR1* metric, and appropriate options ($n = 4$ runs per instance, 1.5s per-run cutoff). Next, we instantiate our *ExampleMetaAlgorithmImplementation*, construct the default *ParameterizedAlgorithm* using it, and run it under the environment we configured. When the run completes, we verify that its reported

Field	Evaluation		Aggregation
	Base Case	Exceptional Case	
ASQ	*SOLUTION_QUALITY*	*SOLUTION_QUALITY* or *null*	mean
PAR1	measured CPU time	*MAX_CPUTIME*	mean
ReportedPAR1	*CPUTIME*	*MAX_CPUTIME*	mean
MaxPAR1	max(measured CPU time, *CPUTIME*)	*MAX_CPUTIME*	mean
PAR10	measured CPU time	10 × *MAX_CPUTIME*	mean
ReportedPAR10	*CPUTIME*	10 × *MAX_CPUTIME*	mean
MaxPAR10	max(measured CPU time, *CPUTIME*)	10 × *MAX_CPUTIME*	mean
MEDR	measured CPU time	*MAX_CPUTIME*	median
ReportedMEDR	*CPUTIME*	*MAX_CPUTIME*	median
MaxMEDR	max(measured CPU time, *CPUTIME*)	*MAX_CPUTIME*	median

Table B.2: Summary of metrics defined in *PerformanceMetric* class. Evaluation semantics refer to those listed in Table B.1. Run timeouts and crashes are considered exceptional cases.

solution quality (i.e., the mean reported runtime) is within $2\sigma_\mu = 2\sigma/n^{1/2}$ of the expected value of 0.75s (corresponding to an $\alpha \approx 0.05$ confidence level). Additional assertions verify that the expected number of target runs were executed, and that the run's final status is correct.

```java
@Test
public void TestRunsSimpleAlgorithmLocalExecution() throws Exception {
  // setup meta-instance
  int n = 4;
  ParameterlessAlgorithm alg = getAlgorithm();
  InstanceList instances = getInstances();
  ParameterlessAlgorithmList alglist = new ParameterlessAlgorithmList(alg);
  ParameterSettingBuilder psb = new ParameterSettingBuilder();
  psb(Semantics.MAX_SUBRUN_CPUTIME, 1.5);
  psb(Semantics.RUNS_PER_INSTANCE, n);
  InstanceMetricMetaProblemInstance metaInst = new InstanceMetricMetaProblemInstance(
      alglist, instances, PerformanceMetric.ReportedPAR1, "test metainst", psb());
  // run meta-algorithm on instance
  ExampleMetaAlgorithmImplementation metaimpl = new ExampleMetaAlgorithmImplementation();
  Algorithm metaAlg = metaimpl.getDefaultParameterizedAlgorithm();
  metaAlg.setProblemInstance(metaInst);
  metaAlg.setScenarioValue(Semantics.MAX_CPUTIME, 75.0);
  Environment env = getEnvironment();
  AlgorithmRun r = env.fetchRun(metaAlg.getAlgorithmRunRequest());
  r.run();
  r.waitForVisitors();
  // verify results
  assertEquals(0.75, (Double)r.getLastOutputValueOnly(Semantics.SOLUTION_QUALITY), 2*SIGMA
      /Math.sqrt(n));
  assertEquals(n*10, r.getSubrunCount());
  assertTrue(RunStatus.finishedWithoutError(r.getStatus()));
}
```

B.1.3 Additional Outputs and Improvements

Although we have tested our simple meta-algorithm and demonstrated it is working correctly, it is not particularly efficient and does not yet fulfill our stated design goals. In this subsection we will resolve both of these issues by parallelizing our implementation and by adding plot and statistical test outputs.

B.1.3.1 Improving Parallelism with *AlgorithmRunVisitors*

One problem with our initial implementation is that it provides a serial solution to an inherently parallel task. We can remove this limitation in several ways, but perhaps the most interesting involves *AlgorithmRunVisitors*: runnable objects that are registered with *AlgorithmRuns* to execute on certain events; i.e., whenever the algorithm produces output, changes status, or completes. By starting many runs in parallel and registering completion visitors to score them, we improve the efficiency of our implementation.

We begin by promoting the *scores* map from the *start()* method to a final field of the *ExampleMetaAlgorithmRun* class, which will be used shortly when overriding the *getOutputSnapshot()* method. We also add a new volatile double field called *myVisitorTime*, and set it to 0.0. We then add the following code defining a completion visitor to the *start()* method before the target algorithm execution loop:

```
final List<AlgorithmRun> runs = new LinkedList<AlgorithmRun>();
final ExampleMetaAlgorithmRun thisrun = this;
AlgorithmRunVisitor visitor = new AlgorithmRunVisitor() {
  public void visit(AlgorithmRun run) {
    synchronized(scores) {
      try {
        Double score = metric.evaluate(run);
        Object seed = run.getAlgorithmRunRequest().getScenarioValue(Semantics.SEED);
        ProblemInstance instance = run.getProblemInstance();
        if (!scores.containsKey(seed)) {
          scores.put(seed, new HashMap<ProblemInstance,Double>());
        }
        scores.get(seed).put(instance, score);
      } finally {
        runs.remove(run);
        runsDone++;
        scores.notifyAll();
        thisrun.myVisitorTime += this.getRunTime();
      }
    }
  }
};
```

Next, we modify the main execution loop to register the completion visitor on each run before it is started, and to start multiple runs in parallel. Note, however, that we still limit the total number of parallel runs, as otherwise we risk hitting system thread limits. By adding an additional setting to the *ExampleMetaAlgorithmImplementation*, this limit can be user-specified.

```
OUTERLOOP:  for (int i=0; i<runsPerInst; i++) {
  Object seed = ((SampleableDomain<?>)target.getScenarioDomain(Semantics.SEED)).getSample(
      rng);
  target.setScenarioValue(Semantics.SEED, seed);
  for (ProblemInstance instance: instances) {
    // wait for a free run slot
    synchronized(scores) {
      while (runs.size() >= maxParallelRuns && super.getTerminationStatus() == null) {
        try {
          scores.wait();
        } catch (InterruptedException e) {}
      }
    }
    if (super.getTerminationStatus() != null) {
      break OUTERLOOP;
    }
    target.setProblemInstance(instance);
    AlgorithmRunRequest req = target.getAlgorithmRunRequest();
    req.setDistributedExecutionOK(true);
    AlgorithmRun run = runner.fetchRun(req);
    run.registerCompletionVisitor(visitor);
    synchronized(scores) {
      runs.add(run);
    }
    Global.getThreadPool().execute(run);
  }
}
```

We must add code to wait until all target runs finish before producing output:

```
synchronized(scores) {
  while (runs.size() > 0) {
    if (super.getTerminationStatus() != null) {
      runner.terminate(RunStatus.TERMINATED);
    }
    try {
      scores.wait();
    } catch (InterruptedException ignored) {}
  }
}
super.updateOutput(getOutputSnapshot());
```

Note that this code calls a method named *getOutputSnapshot()* in order to produce outputs. This method is also used by HAL to obtain on-demand output updates during run execution. The im-

plementation inherited from *MetaAlgorithmRun* simply returns the most recently-produced outputs; we override it with:

```
public Map<String, Object> getOutputSnapshot() {
  Map<String, Object> out = new HashMap<String, Object>();
  List<Double> aggregatedScores = new LinkedList<Double>();
  synchronized(scores) {
    for (Map<ProblemInstance, Double> insScore: scores.values()) {
      aggregatedScores.add(metric.aggregate(insScore.values()));
    }
  }
  out.put("metricScore", metric.aggregate(aggregatedScores));
  return out;
}
```

Finally, we must make a change to how runtime for our meta-algorithm is computed. By default, HAL measures the runtime of a meta-algorithm (excluding target runs) as the time taken by the thread executing its *start()* method, and counts time taken by visitors as miscellaneous overhead. However, our completion visitor is a core part of the algorithm, not an overhead. Thus, we use the *myVisitorTime* field to correct this runtime accounting, by overriding two superclass methods:

```
public double measureCpuTime() {
  return super.measureCpuTime() + myVisitorTime;
}

public double measureVisitorCpuTime() {
  return super.measureVisitorCpuTime() - myVisitorTime;
}
```

This completes the changes needed to allow parallel execution of target runs performed by our meta-algorithm, and to allow real-time result output monitoring. Our unit test shows that the implementation remains valid, and completes more quickly in terms of wall-clock time.

B.1.3.2 *Plots* and *Statistics*

While our initial meta-algorithm is correct and (now) fairly efficient, it does not produce the desired output. We now show how to include plots and statistical test results in the meta-algorithm's output. In particular, we output an overlay of solution cost distributions for each of the random seeds used, and report the results of a Friedman test run on the collected data, paired by seeds.

We first add the new outputs to the *ParameterSpace* in the *ExampleMetaProblemImplementation*:

```
psb.clear();
psb.put("metricScore", Semantics.getDomain(Semantics.SOLUTION_QUALITY));
psb.addAlias(Semantics.SOLUTION_QUALITY, "metricScore");
psb.put("SCD", new ObjectClassDomain(Plot.class));
psb.put("Friedman Q", new RealDomain(0., null));
```

```
psb.put("Friedman p", new RealDomain(0., 1.));
OUT_SPACE = psb.build();
```

We next deal with producing the overlaid SCD plots. There are many useful utility methods in the *Plot* class for building common plots. We first construct a *Plot* object, and then add layers to it, where each layer is a solution cost distribution over runs with a single seed. The modified *getOutputSnapshot()* method is:

```
public Map<String, Object> getOutputSnapshot() {
  Map<String, Object> out = new HashMap<String, Object>();
  List<Double> aggregatedScores = new LinkedList<Double>();
  Plot p = new Plot();
  p.setKey(false);
  synchronized(scores) {
    for (Map<ProblemInstance, Double> instanceScores: scores.values()) {
      Collection<Double> vals = instanceScores.values();
      p.addLayer(Plot.makeCDF(vals, true), null, null, null, Plot.HISTOGRAM, Plot.ANY,
          null);
      aggregatedScores.add(metric.aggregate(vals));
    }
  }
  out.put("metricScore", metric.aggregate(aggregatedScores));
  out.put("SCD", p);
  return out;
}
```

To perform the statistical tests, we can use the *Statistics* object passed into the *ExampleMetaAlgorithmRun* constructor. We first create a volatile double field named *extraCpuTime*, and then again modify the *getOutputSnapshot()* method. This time, we collect data blocks for each seed that has been run on every instance, and run the Friedman test on this blocked data. Since the test may be computed outside of the JVM (e.g., in R), we also record any external CPU usage.

```
public Map<String, Object> getOutputSnapshot() {
  Map<String, Object> out = new HashMap<String, Object>();
  List<Double> aggregatedScores = new LinkedList<Double>();
  Plot p = new Plot();
  p.setKey(false);
  List<List<Double>> data = new LinkedList<List<Double>>();
  synchronized(scores) {
    for (Map<ProblemInstance, Double> instScores: scores.values()) {
      List<Double> vals = new ArrayList<Double>(instScores.values());
      if (vals.size() == instances.size()) data.add(vals);
      p.addLayer(Plot.makeCDF(vals, true),null,null,null,Plot.HISTOGRAM,Plot.ANY, null);
      aggregatedScores.add(metric.aggregate(vals));
    }
  }
  out.put("metricScore", metric.aggregate(aggregatedScores));
  out.put("SCD", p);
```

```
  if (data.size() > 1) {
    System.out.println(data);
    TestResult result = stats.friedmanTest(data);
    out.put("Friedman Q", result.getStatistic());
    out.put("Friedman p", result.getP());
    extraCpuTime += result.getExternalComputeTime();
  }
  return out;
}
```

Finally, we must include the extra time taken in computing the Friedman test to the time reported for our meta-algorithm run:

```
public double measureCpuTime() {
  return super.measureCpuTime() + myVisitorTime + extraCpuTime;
}
```

B.1.3.3 Testing

This completes our meta-algorithm, including all originally-desired functionality. However, we need to verify that the plots and test work as expected. If we consider our test target algorithm, we see that the random noise is determined exclusively by the provided seed, and is independent of the problem instance. As such, we expect that the Friedman test to reject the null hypothesis at $\alpha = 0.01$ confidence, and to see clear dominance in the overlaid SCD. We modify our unit test to display the plot using Gnuplot, and to verify the expected test result:

```
Plotter plotter = new GnuplotPlotter();
plotter.display((Plot)r.getLastOutputValueOnly("SCD"));
assertEquals(0.0, (Double)r.getLastOutputValueOnly("Friedman p"), 0.01);
```

All tests pass as expected ($p \approx 1E-6$), and the resultant overlaid SCD is shown in Figure B.1 (left). Note that if we change our test algorithm slightly to add an instance-dependency to the noise (as below), the Friedman test no longer rejects at $\alpha = 0.01$ ($p \approx 0.2$) and the SCD plot looks much more random, as in Figure B.1 (right). As this behaviour is expected, it provides evidence that our implementation is correct.

```
long seed = Long.valueOf(args[1])*f.length();
```

B.1.4 Plugin Distribution

To share our new meta-algorithm as a plugin, we must export it as a self-contained JAR. In addition to the `ExampleMetaAlgorithmImplementation.class`, this JAR must contain the file

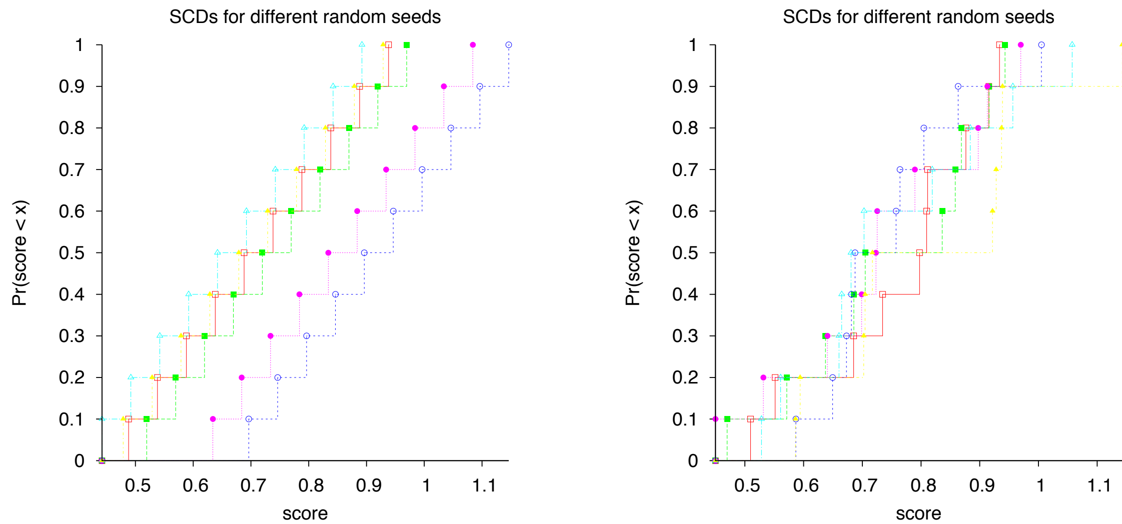

Figure B.1: SCDs showing performance across test instance distribution for two algorithms, and several random seeds. Clearly for the target algorithm on the left, expected performance is not independent of the seed.

`/plugin.json` that identifies the packaged objects HAL should import. This file contains the output of *toSpec()* in a JSON array:

```
[{"classname": "ca.ubc.cs.beta.hal.samples.ExampleMetaAlgorithmImplementation", "version":
    "1.0.0"}]
```

Making the plugin available for use in the HAL UI requires explicitly editing servlet code for the corresponding specification and monitoring schemes in the HAL 1.1 core; it is not presently possible to include this additional UI code in the plugin itself. Support for more modular UI code is planned for HAL 1.2.

B.2 Execution Management

As a meta-algorithmic framework, algorithm execution is the fundamental task performed by HAL. However, the details of actually executing an algorithm are complex, depending on the specifics of the algorithm to be executed and (especially) on the particular computing environment to be used. As such, one of the most important abstractions in HAL is the *ExecutionManager* interface, and the design or refinement of special-purpose *ExecutionManager* implementations is anticipated to be a relatively common task in HAL framework development (as opposed to meta-algorithm development). This section provides a more detailed overview of *ExecutionManagers* and their role in the HAL execution pipeline; as well as specifics of the three concrete implementations provided in HAL

1.1. Unlike the previous section, the following is intended to be a useful guide for understanding the existing execution components, rather than a tutorial for implementing new ones. As such, no code samples are provided; we instead refer the reader to the sources for the discussed components.

We also point out that meta-algorithm developers never interact directly with the classes discussed here. Instead, meta-algorithms are provided a *SubrunRunner* instance at runtime that moderates interaction with the underlying execution management infrastructure. However, it is important for meta-algorithm developers to at least be familiar with the execution pipeline, if not the details of the components that implement it.

B.2.1 HAL 1.1 Execution Pipeline

ExecutionManagers, along with a *DataManager* instance (discussed in the next section) and a *Statistics* instance (used to perform statistical tests), are used to construct an execution *Environment* in HAL, as illustrated in Figure B.5. An *Environment* instance captures the overall structure of an execution environment, and is responsible for initial handling of *AlgorithmRunRequests*.

The primary task of the *Environment* is to delegate *AlgorithmRunRequests* to appropriate *ExecutionManagers*, which in turn initiate *AlgorithmRuns*. An *Environment* selects the *ExecutionManager* to use for each request according to its position in the tree of execution traversed during a meta-algorithmic experiment:

- **parent** meta-algorithm runs correspond to top-level experiments specified by the user, each of which defines the root of a separate execution tree
- **target** algorithm runs correspond to leaves of an execution tree
- **target meta-algorithm** runs correspond to internal nodes of an execution tree

This makes it possible, e.g., to define an environment in which a HYDRA-$*$ run is submitted to a compute cluster, its configurator runs are again distributed across the cluster, but target algorithms are executed on the same host as the corresponding configurator run rather than being distributed a third time.

Secondary responsibilities of the *Environment* object include:

- limiting the maximum number of concurrent target algorithm runs
- specifying a minimum runtime cutoff for runs to be reused from the data manager
- configuring how output trajectories should be logged
- restricting execution hosts (regular expression on hostname or MAC address)

These options are set using keys from the *Environment.Preference* enumeration.

Finally, the environment provides remote access via RPC to information about currently-active runs (including completion progress and CPU time consumed) and relays remote run termination requests; the RPC interface is defined in *EnvironmentRequestInterface*. The UI uses RPC to display live run progress; as a result, all execution hosts must be directly reachable over TCP/IP from the UI server.

The *ExecutionManager* interface itself is conceptually quite simple, and can be viewed as an *AlgorithmRun* factory. It defines two related methods: $fetchRun(\cdots)$ and $queueRun(\cdots)$. Both of these methods fulfill a provided *AlgorithmRunRequest*; the difference is that whereas *fetchRun* returns an *AlgorithmRun* object that can be used to initiate and interact with the corresponding run, *queueRun* returns only a data manager ID that can be used to retrieve an *AlgorithmRun* later as required. This means that *queueRun* is a low-overhead option suitable for use when launching quantities of loosely-coupled algorithm runs, and that *fetchRun* is appropriate when tighter integration is required.

The *AlgorithmRun* object returned by an *ExecutionManager* implements a more complex interface, as illustrated in Figure B.2 and discussed earlier in the context of *MetaAlgorithmImplementations*. For *InternalAlgorithmImplementations* such as these, the *ExecutionManager* will typically re-use the corresponding *InternalAlgorithmRun* object (possibly via a wrapper); for *ExternalAlgorithm-Implementations* corresponding to external target algorithms, it will typically return an instance of a proprietary *ExternalAlgorithmRun* subclass. In all cases, the *AlgorithmRun* object has five key responsibilities. It is responsible for:

Execution: initiating eventual execution of an underlying algorithm

Termination: allowing termination of the algorithm run once executing, as well as any sub-runs that may also be executing

CPU time: measuring total CPU time consumed during execution, and for automatic termination of the run if this time exceeds a specified budget.[2]

Outputs: providing access to the status and sequenced outputs of that algorithm run, in the format specified by the associated output *ParameterSpace*

An additional responsibility is that of ensuring all registered *AlgorithmRunVisitors* are executed when the corresponding events (output production, status change, completion) occur. However, as

[2] In practice, a run is terminated if it exceeds $ExecutionManager.MULT_TIME_BUFFER \times request.getScenarioValue(Semantics.MAX_CPUTIME) + ExecutionManager.ADD_TIME_BUFFER$ CPU seconds; by default these buffer constants are set to $4/3$ and $1s$ respectively.

this functionality is entirely implemented in the abstract class, and is not overridden by any concrete implementations, we do not focus on it further.

B.2.1.1 Algorithm Transformations

An important concept directly affecting *ExecutionManagers* is that of algorithm transformations. A *TransformedAlgorithm* is used to provide a view of a *ParameterizedAlgorithm* with properties that differ according to some invertible function of the properties of the underlying algorithm. In particular, transformations are used in HAL to alter the effective parameter space presented to configuration meta-algorithms; for example discretizing continuous spaces, or normalizing or applying logarithmic transformations to continuous spaces. When dealing with a *TransformedAlgorithmRun-Request*, an *ExecutionManager* must execute the underlying untransformed request, but return an *AlgorithmRun* that refers to the original transformed request. The *AbstractTransformSupporting-ExecutionManager* class provides this functionality, and is extended by all HAL 1.1 execution managers.

B.2.1.2 Limitations

In order to avoid problems deploying their meta-algorithms to end-users with novel execution environments, HAL meta-algorithm developers should understand the limitations of HAL execution, and in particular should not make incorrect behavioural assumptions about *AlgorithmRuns*.

First, it is incorrect to assume that an algorithm run will start immediately, or almost immediately, upon request. In fact, the *AlgorithmRun* does not begin to execute until its *run*() method is called; this is done to allow time for visitors to be registered on the run before any triggering events occur. Even after *run*() is called, the request may be queued for arbitrarily long by the *Environment* (depending on its configuration), and the *ExecutionManager* may involve it being queued for still longer once released by the *Environment*. Similarly, it is incorrect to assume that parallel algorithm run requests will be executed in the order requested, as some cluster queueing systems do not provide this guarantee.

Second, it is incorrect to assume a specific *AlgorithmRun* subclass will be returned. Even if the *ExecutionManager* classes are known, the returned object may be wrapped (for example, if the *Environment* queues the run instead of starting it immediately) or may be retrieved from a database instead of being executed directly. As such, the *WrappedAlgorithmRun* and *DatabaseAlgorithmRun* interfaces are important ones to be aware of.

Finally, it is incorrect to assume that CPU time measurements made during the progress of a run are exact; in some environments (such as some clusters) high-resolution progress monitoring is not

available, so mid-run measurements are in general lower bounds at best. One side effect of this is that HAL's automatic termination of target runs may not always occur as quickly as expected. The relative sequence of outputs is accurate, however, as is the post-completion CPU time measurement.

B.2.1.3 *ExecutionManager* Implementations

In the following subsections, we discuss the specific *ExecutionManager* implementations provided in HAL 1.1: *LocalExecutionManager*, *SSHExecutionManager*, and two cluster execution managers, *SGEClusterExecutionManager* and *TorqueClusterExecutionManager*. In particular, we describe each implementation in terms of how the external runs it produces fulfill the stated responsibilities of an *AlgorithmRun*, and also in terms of how internal runs are executed.

B.2.2 *LocalExecutionManagers*

The *LocalExecutionManager* is the most fundamental execution manager in HAL 1.1. First, we describe the *LocalExternalAlgorithmRuns* it produces:

Execution: The command string used to execute an external algorithm is assembled from the provided input format string and the parameter and setting values from the request. This string is then tokenized and the Java *ProcessBuilder* class is used to execute the run on the local machine.

Termination: The Java *Process* object returned by the *ProcessBuilder* is retained and could in principle be used to terminate runs. However, this method does not support termination signals (such as SIGTERM and SIGKILL), which we may need in some cases (for example, when processes become unresponsive). Thus, the Sigar [3] library which provides such support is used to terminate algorithm runs.

CPU time: Sigar is also used to accurately measure an algorithm run's CPU time during execution. For measurement of final CPU time, the POSIX `time` command is used (for Windows, an analogous binary called `timerun.exe` is provided and used).

Outputs: Outputs of the executing algorithm are parsed from the Java *Process* using a parser thread to consume and parse each of the standard output and standard error streams. User-provided output format strings are converted into regular expressions that are used to perform this parsing. Depending on *OutputHandlingOption* values provided to the *ExecutionManager* constructor, raw output values may also be echoed to the console as they are read, or recorded for later inspection.

[3] http://www.hyperic.com/products/sigar

InternalAlgorithmImplementations can be run by the *LocalExecutionManager* in two ways: either in the same JVM that is currently running HAL, or in a new JVM dedicated to the run. Of these, the former is trivial – the *InternalAlgorithmRun* returned by the implementation's *getRun*$(\cdots)$ method is returned directly. The latter is less trivial. First, the *LocalExecutionManager* fetches a subsidiary external algorithm run (calling a *main* method in `Hal.java`) that, when launched, starts a new instance of HAL that in turn loads the original run request and runs the internal algorithm in its local JVM. The *LocalExecutionManager* then constructs an *UpdatableWrappedDatabase-AlgorithmRun* that wraps this subsidiary external run. A status change visitor is registered on the subsidiary algorithm run that replaces the "core" of the wrapped run with a *DatabaseAlgorithm-Run* providing access to the *InternalAlgorithmRun* executing in the second JVM, as soon as the subsidiary run is started. Finally, it returns the wrapped algorithm run.

The main reason for this method of launching an internal run in a new process is so that the external algorithm run does not start until the caller actually executes the *run*() method of the returned wrapped run; if we ran the subsidiary algorithm directly and only returned the subsequent database run, the run would start before the *AlgorithmRun* returns. This is problematic, as it means visitors subsequently registered on the returned run could miss important events. The idea of combining updatable wrapped algorithm runs with algorithm run visitors in order to support multi-stage execution processes is used in all other execution managers.

The *LocalExecutionManager* also supports a third run class called *NonHALAlgorithmRun*, which was used to enable support for GGA via its closed-source binary implementation. We do not recommend that this class be used for other purposes and do not further document it here.

B.2.3 *SSHExecutionManagers*

The *SSHExecutionManager* is the perhaps the least directly-used execution manager in HAL 1.1. It is primarily intended to support the implementation of cluster-based execution managers (see below). While it can also be used to run arbitrary algorithms, its described limitations regarding termination and CPU time measurement lead us to discourage such usage.

An *SSHExecutionManager* instance is associated with a single SSH host. This host is specified by a URI of the form:

```
ssh://[user[:password]@]ssh.server.name/path/to/hal.jar
```

If no username or password are provided, the SSH execution manager will attempt to log in using the current user's name and local RSA or DSA SSH key. The path to HAL should point to a HAL installation that is isomorphic to the calling HAL installation in the sense that valid relative paths to instances and algorithm binaries on one host remain valid on the other. As this is most commonly achieved when both machines mount the same shared file system, we refer to this as a

shared filesystem requirement. Both hosts must also be able to connect to the same data manager (a shared data manager requirement).

Many helper commands needed on the remote machine (such as `time`, `cd`, `ls`, etc.) can be specified via the *SSHExecutionManager* constructor; default values should work on any POSIX host and can be found in *SSHExecutionManager.DFLT_CMDS*. In addition, arbitrary initialization commands can be specified which will be run before the target execution call is made. The SSH session is assumed to use a BASH shell; other shells may work but are not recommended.

SSHExternalAlgorithmRuns are quite similar to *LocalExternalAlgorithmRuns*:

Execution: A command string is generated using the same methods used by the *LocalExecutionManager*. Further commands are added to obtain the hostname, MAC addresses, start time, and exit codes for the host that ultimately runs the job. This set of commands is executed on the remote host using the JSch library. [4]. No instance or algorithm files are uploaded before execution; this is the main reason for the shared filesystem requirement.

Termination: JSch supports sending of signals over SSH and this is used for terminating runs; unfortunately, many common OpenSSH `sshd` server implementations ignore signals, so as a backup HAL simply closes the connection running the request. On most platforms this forces associated processes to exit, terminating the run. Note that this has the unfortunate side effect of immediately terminating target algorithm runs for which the controlling connection is unintentionally closed.

CPU time: Incremental runtime measurement during execution is not supported by *SSHExternalAlgorithmRuns*; as a result, enforcing of runtime limits is not possible. By default, the POSIX `time` command is used for measurement of final CPU time, as in the *LocalExecutionManager*.

Outputs: Outputs of the executing algorithm are parsed from the standard output and standard error streams from the SSH connection via the same parsing threads used by the *LocalExecutionManager*.

InternalAlgorithmImplementations are run by the *SSHExecutionManager* in exactly the same way as separate-JVM internal algorithms are run by the *LocalExecutionManager*, except that the subsidiary algorithm is launched over SSH as a *SSHExternalAlgorithmRun* instead of locally as a *LocalExternalAlgorithmRun*.

[4] http://www.jcraft.com/jsch

B.2.4 *SGE-* and *TorqueClusterExecutionManagers*

The two cluster execution managers available for HAL 1.1 (*SGEClusterExecutionManager* and *TorqueClusterExecutionManager*) are commonly used when running large-scale experiments. As their names suggest, the SGE manager requires access to a Sun (or Oracle) Grid Engine managed compute cluster [5], and the Torque manager requires access to a TORQUE-managed cluster [6].

Unlike the execution managers described previously, cluster execution managers require a subsidiary execution manager to be specified as a constructor argument. This subsidiary execution manager is used to run *Algorithms* corresponding to the common `qsub`, `qstat`, and `qdel` cluster management commands. Specific syntax can be modified by providing alternative *Algorithm* objects to the constructors; defaults are available in the corresponding *DFLT_QSUB*, *DFLT_QSTAT*, and *DFLT_QDEL* static fields. Of course, in order to work properly the subsidiary execution manager must execute these commands on a valid cluster submit host; a common choice is an *SSHExecutionManager* pointing to the head node.

Cluster execution managers have both a shared filesystem requirement (in this case, the file system must *actually* be shared, not just isomorphic) and a shared data manager requirement. They also similarly require helper commands valid on the compute nodes, and allow these commands to be specified via the constructor. Initialization commands are also supported. Cluster submission scripts are assumed to be executed using a BASH shell.

ClusterExternalAlgorithmRuns are somewhat more complex than their local and SSH counterparts:[7]

Execution: A command string is generated using the same methods used by the *SSHExecutionManager*. This command string is written into a script file that located in a `cluster/` subdirectory. Finally, an *Algorithm* corresponding to the appropriate `qsub` command for the cluster is launched via the subsidiary execution manager to queue the script. No instance or algorithm files are uploaded before execution; this is one of two reasons for the shared filesystem requirement.

Much like *InternalAlgorithmRuns* launched by previous execution managers, *ExternalAlgorithmRuns* returned by cluster execution managers are actually *UpdatableWrappedAlgorithmRuns*. At first, the algorithm wraps the submission algorithm run; once submission is completed, a

[5] http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html

[6] http://www.clusterresources.com/products/torque-resource-manager.php

[7] Note that in the most recent versions of HAL, the Environment redirects fetchRun calls on cluster execution managers to queueRun calls, and returns corresponding *DatabaseAlgorithmRuns* instead of *ClusterExternalAlgorithmRuns*. This (hopefully temporary) workaround ensures a dedicated HAL process monitors every cluster job, and was necessary to improve robustness in scenarios with many simultaneous cluster-distributed target algorithm runs.

completion visitor changes updates the wrapped run to delegate to a different *ClusterExternalAlgorithmRun* instance.

Termination: Runs are terminated by running another *Algorithm* with the subsidiary execution manager, this time corresponding to the `qdel` command.

CPU time: Incremental runtime measurement during execution is performed using a third *Algorithm*, corresponding to the `qstat` command. This command often only has a CPU time resolution of 30-60s, so measured runtimes are lower bounds and enforcement of runtime limits can be off by up to this amount. By default, the POSIX `time` command is used for measurement of final CPU time, as in the *LocalExecutionManager*.

Outputs: Raw outputs of the executing algorithm are read from cluster-generated standard output and standard error logs in the `cluster/` subdirectory. These logs are streamed into the same parsing threads used by the other execution managers. The assumption that these log files are readable from the requesting machine is the second reason for the shared filesystem requirement.

InternalAlgorithmImplementations are run by the cluster execution managers in a similar way to internal algorithms run by the *SSHExecutionManager*. The main difference is that two updates to the wrapped run occur instead of one: first, an update from the cluster submission run to a subsidiary run that launches the new HAL instance (analogous to the update performed for *ClusterExternalAlgorithmRuns*); and later, an update from the subsidiary run to a database algorithm run (analogous to the update performed for SSH *InternalAlgorithmRuns*).

B.3 Data Management

Data management is central to the HAL framework. In addition to storing all experimental data, much of the inter-process and inter-machine communication necessary for operation in distributed environments is accomplished through the *DataManager* interfaces. Moreover, much of this functionality is provided to developers and end-users in a transparent manner. As a result of the wide variety of data types stored and functionality provided, the data management interfaces are some of the most complex components of the HAL framework (the class diagram of Figure B.5 does not show nearly all methods), and as a result of their central role they are important to understand when developing core HAL components. As such, while we do not expect many users will have need to develop entirely new data manager implementations, we nonetheless provide an overview of their high-level functionality and in particular detail the SQL-based implementation that underlies both of the data managers included in HAL 1.1.

B.3.1 Data Manager Interfaces

All data managers must implement the *FullAccessDataManager* interface. This interface defines a large number of methods, but many can be classified as one of four types related to archiving of HAL primitives:

add— methods insert new HAL primitives into the data manager; e.g., *addImplementation(AlgorithmImplementation)*, *addInstance(ProblemInstance)*, *addEnvironment(Environment)*

get—[Keys] methods list primitives in the data manager in terms of their human-readable keys; e.g., *getImplementationNameVersions()*, *getInstanceHashes()*, *getEnvironmentNames()*

get— methods retrieve HAL primitives from the data manager given a key; e.g., *getImplementation(name, version)*, *getInstance(hash)*, *getEnvironment(name)*

delete— methods remove HAL primitives from the data manager given a key; e.g., *deleteImplementation(name, version)*, *deleteInstance(hash)*, *deleteEnvironment(name)*

Most HAL primitives additionally support user-specified textual descriptions; there are also data manager methods for working with these descriptions:

set—Description methods annotate a HAL primitive from the data manager given a key; e.g., *setImplementationDescription(name, version, description)*, *setInstanceDescription(hash, description)*, *setEnvironmentDescription(name, description)*

get—Description methods get the description of a primitive from the data manager given a key; e.g., *getImplementationDescription(name, version)*, *getInstanceDescription(hash)*, *getEnvironmentDescription(name)*

A third category of methods facilitates the use of tags to identify compatibility between, for example, problem instances and algorithm implementations:

tag— methods associate tags with a HAL primitive; e.g., *tagInstance(hash, tag)*, *tagDistribution(name, tag)*

get—Tags methods retrieve the set of tags associated with a HAL primitive; e.g., *getInstanceTags(hash)*, *getDistributionTags(name)*

delete—Tag methods disassociate a tag and a HAL primitive; e.g., *deleteInstanceTag(hash, tag)*, *deleteDistributionTag(name, tag)*

get—RequiredTags methods return the set of tag sets that define the problem instance compatibility of a primitive; e.g., *getFeatureRequiredTags(name)*, *getImplementationRequiredTags(name, version)*

getCompatible—[Keys] methods list primitives in the data manager that are compatible with a set of tag sets, in terms of their human-readable keys; e.g., *getCompatibleFeatureNames(tagSets)*, *getCompatibleImplementationNameVersions(tagSets)*

In addition to storing HAL primitives, data managers are responsible for archiving log output produced by arbitrary infrastructure code. HAL 1.1 uses Java's built-in logging package, so this means that data managers must provide a factory method called *getLogHandler()* that returns a *LogHandler* object; the *Environment* will register this handler with the logging framework.

Most remaining data manager methods are directly related to algorithm execution. Some correspond to *AlgorithmRun* methods and are fairly intuitive; e.g., *getRunStatus(runId)*, *getMeasuredCpuTimeFromSource(runId)*, and *terminate(runId, status)*.[8] Other, more interesting methods are discussed further in Section B.3.2; in particular, methods for obtaining *AlgorithmRun* objects corresponding to existing runs, and the *decorate(ExecutionManager)* method used to augment *ExecutionManagers* with automatic result archiving and run reuse functionality.

B.3.1.1 *ReadOnlyDataManagers*

The *FullAccessDataManager* interface is actually a super-interface of the more restricted *ReadOnlyDataManager* interface. All *get—* methods are defined in the read-only interface, and *add—* and *delete—* methods in the full-access interface. The read-only interface is used to safely provide meta-algorithms with access to the data currently available in HAL. A wrapper class defined in *ReadOnlyDataManager* is used to prevent meta-algorithm developers from simply up-casting the provided *ReadOnlyDataManager* instance back to a *FullAccessDataManager*.

B.3.2 Experimental Data Management

While the storage of HAL primitives such as target algorithms and problem instances is an important role for a data manager, another major responsibility is the automation of archiving and re-use (where appropriate) of experimental data, in the form of *AlgorithmRuns*. To this end, data managers "decorate" arbitrary *ExecutionManagers* with additional functionality, and support a flexible query method for retrieving archived *DatabaseAlgorithmRuns* according to arbitrary criteria.

[8] Both *terminate* and *get—FromSource* methods use an RPC connection to the executing environment to communicate with remote runs; this is planned to change as RPC usage is phased out.

B.3.2.1 *DecoratedExecutionManagers*

The *decorate(ExecutionManager)* method is called during the construction of a HAL *Environment* to add data management functionality to existing *ExecutionManager* implementations. This method returns a new *DecoratedExecutionManager* object that has the following responsibilities:

- to ensure all HAL primitives in a run request are stored in the data manager
- to check whether incoming run requests can be fulfilled using the results of previous runs; and if so, to return a *DatabaseAlgorithmRun* that does so. The criteria run r must satisfy to be reused for request q are:
    1. q corresponds to an algorithm implementation for which the "cutoff-agnostic" property is false
    2. The CPU time cutoff for q is greater than the minimum required for run re-use in the current *Environment* (see Appendix B.2.1)
    3. q and r share the same implementation, instance, and configuration
    4. q and r have compatible settings (e.g., the CPU time cutoff of q is the same or lower than the CPU time of r)
    5. r either completed successfully or ended due to timeout
    6. r was executed on a valid host for the current *Environment*
- to delegate launching of new runs to the decorated *ExecutionManager*
- to track and record the outputs and status of returned runs in real time; typically accomplished by registering *AlgorithmRunVisitors* for output production and status change on returned runs

B.3.2.2 Run Filters

In addition to implicitly querying a data manager for past runs through the use of a *Decorated-ExecutionManager*, it is often useful to explicitly query for all previous runs that satisfy certain criteria. HAL supports such queries in a data manager-independent way through the *getRun(Filter⟨DataBaseAlgorithmRun⟩ ...)* method.

The *Filter⟨T⟩* interface defines a single method, *contains(T)*, that returns a boolean indicating acceptance or rejection of a single instance of type T. Callers can query with arbitrary acceptance criteria by implementing appropriate filters; for example, the following code will return all runs that took 1.0 ± 0.01 CPU seconds (excluding HAL overhead), and will work for any data manager:

```
List<AlgorithmRun> runs = dataManager.getRuns(
  new Filter<DatabaseAlgorithmRun>() {
    public boolean contains(DatabaseAlgorithmRun r) {
      return Math.abs(r.getMeasuredCpuTimeIfStarted() - 1.0) <= 0.01;
    }
  });
```

Note that this query is very expensive in practice, as its evaluation requires the data manager to check every run ever executed against the filter. For specific data manager implementations, more efficient *Filter* subclasses can be used to significantly improve query efficiency; an equivalent example is given in Section B.3.3.3.

Also note the use of *DataBaseAlgorithmRun.getMeasuresCpuTimeIfStarted()* instead of *AlgorithmRun.getMeasuresCpuTime()*. The reason this method is used is that until the *run()* method of any *AlgorithmRun* is called, 0s of CPU time will be reported. Since the filter acts on runs before they are returned (and thus, before the run() method can be called), this alternative method is used to "peek" at the eventual value. Similar methods exist for overhead and total CPU time, and for run status.

B.3.3 SQL Data Managers

The previous subsection described data manager interfaces and behavioural contracts that apply to all possible HAL data managers, regardless of the storage paradigm ultimately used. One could imagine various data manager implementations backed by SQL databases (such as MySQL[9] or SQLite [10]), key-value stores (such as HBase [11] or memcached [12]), or even a carefully managed custom directory structure. However, in practice both data manager implementations available for HAL 1.1 are backed by SQL databases (MySQL and SQLite), and both extend the *AbstractSQLDataManager* class that we discuss here.

As its name suggests, the *AbstractSQLDataManager* class provides infrastructure to implement a *FullAccessDataManAger* backed by a SQL database. Apache DBCP [13] is used to implement performance-enhancing connection pooling functionality, and queries are handled with Apache DBUtils. [14] Nearly all data manager functionality is implemented in this abstract class; the concrete MySQL and SQLite implementations simply connect to their respective data sources, and override a few queries where proprietary syntax allows for more efficient evaluation.

[9] http://www.mysql.com
[10] http://www.sqlite.org
[11] http://hbase.apache.org
[12] http://memcached.org
[13] http://commons.apache.org/dbcp
[14] http://commons.apache.org/dbutils

B.3.3.1 Database Schema

The *AbstractSQLDataManager* class defines the SQL schema illustrated in Figure B.8. It is able to automatically instantiate this schema when pointed to an empty database, so it is easy for users to maintain separate databases (e.g., for different projects). There are tables in the database corresponding to the discussed HAL primitives, as well as tables storing information about the relationships between them.

B.3.3.2 Query Handlers and Beans

Rather than work with Java's somewhat cumbersome JDBC database infrastructure directly, *AbstractSQLDataManager* uses the Apache DBUtils library. This requires that for each HAL primitive represented in the SQL schema, a corresponding Java bean class be provided to act as a temporary container for all related column values. Specialized *ResultSetHandler* classes are then used to convert raw JDBC *ResultSet* data into special purpose beans (or lists of beans), or to update the database with the data contained in a bean. Each of these beans also has a *create()* method that is able to instantiate the corresponding HAL primitive from these column values. As such, details of both HAL primitive instantiation and low-level JDBC functionality is abstracted away in the *AbstractSQLDataManager* class, considerably streamlining its implementation.

Most of the methods identified in Section B.3.1 are implemented in *AbstractSQLDataManager* via at least two methods: a private method that implements the SQL queries and related logic given a database connection, and a public method that initiates a database connection, calls the private method, closes the connection, and returns results. This factorization allows use of several such methods without repeatedly opening and closing database connections. However, it also leads to the rather large total number of methods defined in *AbstractSQLDataManager*.

B.3.3.3 *SQLRunFilters*

Recall from Section B.3.2.2 that naive use of the *getRuns(⋯)* method is apt to be extremely slow. In order to exploit knowledge of the common SQL schema to speed up common run queries, a *Filter⟨DatabaseAlgorithmRun⟩* subclass called *SQLRunFilter* is provided. This subclass allows implementors to restrict the runs returned by a *getRun(⋯)* call by adding WHERE clauses to the SQL query used to generate the initial list of *DatabaseAlgorithmRuns* that is then further filtered. The shared SELECT clause of this query is in *SQLRunFilter.SELECT_CLAUSE*, and reads:

```
SELECT R.id AS runId, R.status AS originalStatus,
    R.measuredCpuTime AS originalMeasuredTime,
    R.overheadCpuTime AS originalOverheadTime,
    R.startTime AS startTime, R.finishTime AS finishTime,
```

```
    Q.id AS id, Q.parentId AS parentId, Q.parentCpuTime AS parentCpuTime,
    Q.measuredCpuTime AS measuredCpuTime, Q.overheadCpuTime AS overheadCpuTime,
    Q.requestTime AS requestTime, Q.status AS status,
    A.name AS implementationName, A.version AS implementationVersion,
    I.hash AS instanceHash,
    C.hash AS configurationHash,
    coalesce(S2.hash, S1.hash) AS scenarioHash,
    O.hash AS outputSpaceHash,
    E.name AS environmentName,
    H.name AS hostName,
    V.rpc AS rpcAddress,
    L.runId AS nonHalId,
    N.name AS name,
    D.description AS description,
    max(P.sequence) as lastSequence
  FROM Run AS R
    INNER JOIN Implementation AS A ON R.implementationId = A.id
    INNER JOIN Instance AS I ON R.instanceId = I.id
    INNER JOIN Setting AS S1 ON R.scenarioId = S1.id
    INNER JOIN Setting AS C ON R.configurationId = C.id
    INNER JOIN Space AS O ON R.outputSpaceId = O.id
    LEFT OUTER JOIN Request AS Q ON Q.runId = R.id
    LEFT OUTER JOIN Setting AS S2 ON Q.scenarioId = S2.id
    LEFT OUTER JOIN Environment AS E ON Q.environmentId = E.id
    LEFT OUTER JOIN RequestHasName AS N ON N.requestId = Q.id
    LEFT OUTER JOIN Host AS H ON R.hostId = H.id
    LEFT OUTER JOIN ActiveRun AS V ON R.id = V.runId
    LEFT OUTER JOIN NonHALRun AS L ON L.runId = R.id
    LEFT OUTER JOIN RequestHasDescription AS D ON D.id = Q.id
    LEFT OUTER JOIN RunProducedOutput AS P ON P.runId = R.id
```

The WHERE clauses defined by a *SQLRunFilter* are accessed via the *getPredicate()* method, which returns a *SQLRunFilter.Predicate* object. An example that more efficiently implements the example of Section B.3.2.2 might be:

```
List<AlgorithmRun> runs = dataManager.getRuns(
  new SQLRunFilter() {
    private static final Predicate p = new Predicate();
    static {
      p.setWhere("coalesce(Q.measuredCpuTime, R.measuredCpuTime) <= 1.01 AND coalesce(Q.
          measuredCpuTime, R.measuredCpuTime) >= 0.99");
    }
    public Predicate getPredicate() {
      return p;
    }
    public boolean contains(DatabaseAlgorithmRun r) {
      return Math.abs(r.getMeasuredCpuTimeIfStarted() - 1.0) <= 0.01;
    }
  });
```

Note that since this filter implements *Filter*$\langle$*DatabaseAlgorithmRun*$\rangle$, it can also be used with non-SQL data managers (though its additional efficiency would presumably be lost). Finally, note that *SQLRunFilters* sufficient to construct most common queries are included in HAL 1.1.

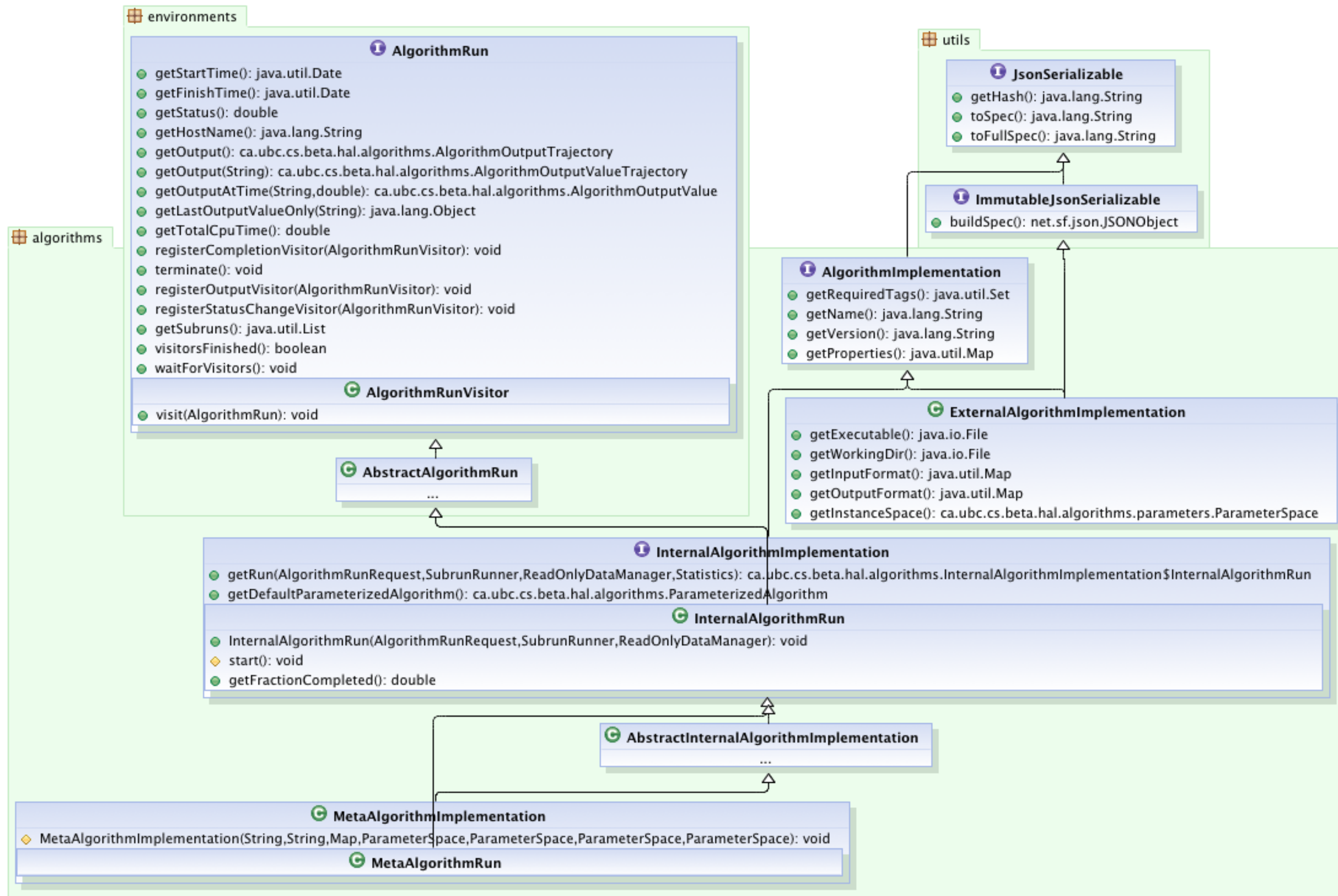

Figure B.2: Class diagram illustrating *AlgorithmImplementations* and *AlgorithmRuns*.

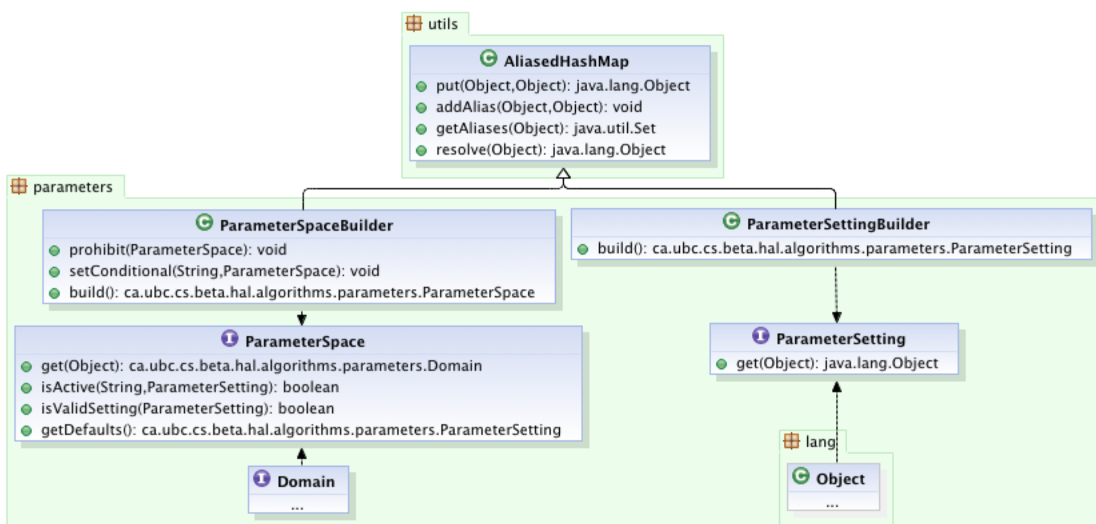

Figure B.3: Class diagram illustrating *ParameterSpaces* and *ParameterSettings*. A *ParameterSpace* is map between *String* parameter names and *Domains* of valid values; a *ParameterSetting* maps between names and specific *Object* values. As both are immutable, they are constructed using associated builder objects.

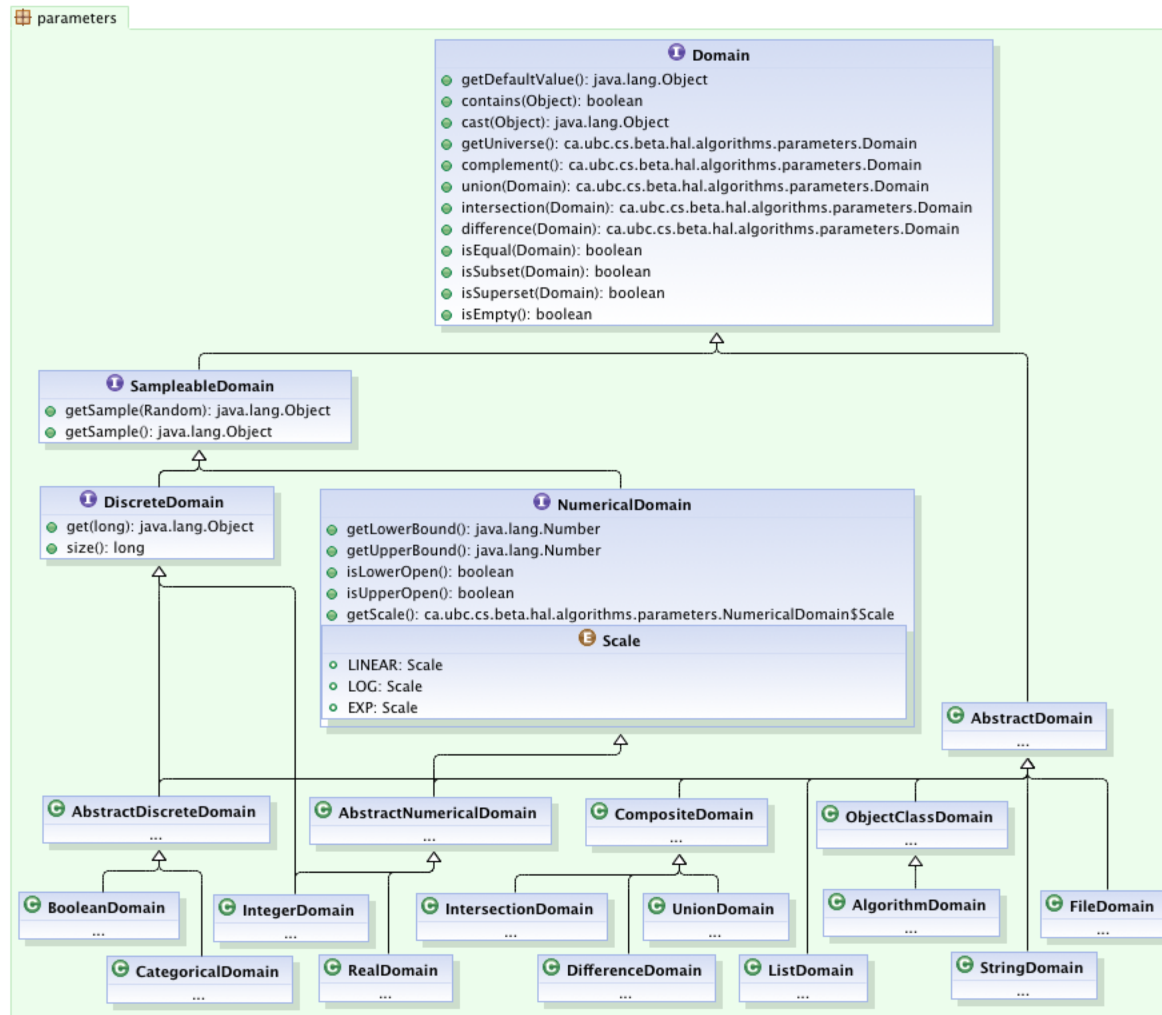

Figure B.4: Class diagram illustrating parameter *Domains* in HAL 1.1. Note in particular the added functionality associated with *SampleableDomain*, *NumericalDomain*, and *DiscreteDomain* sub-interfaces. Also note the *CompositeDomains*, used to implement union, intersection, and difference operations.

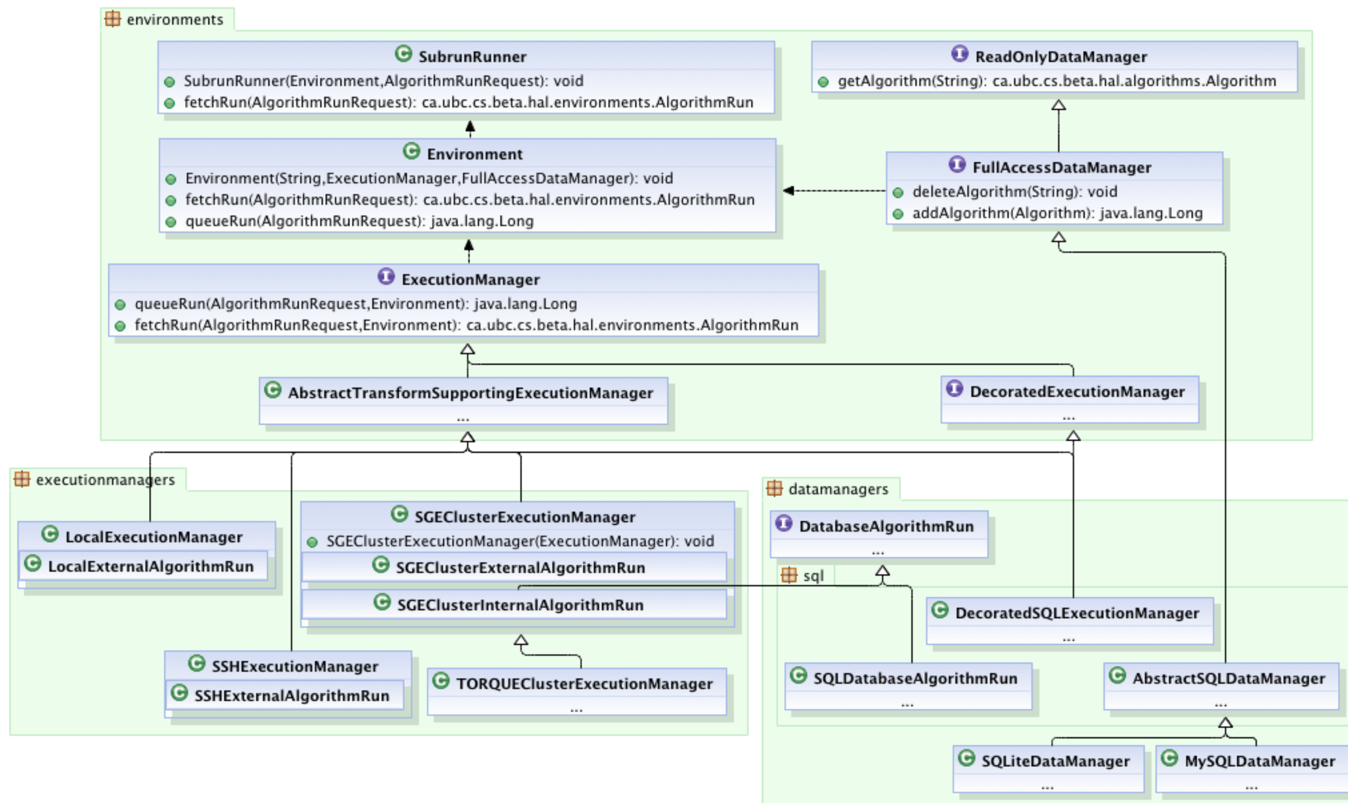

Figure B.5: Class diagram illustrating *Environments* in HAL 1.1. Note that an *Environment* is constructed with a *FullAccessDataManager* and several *ExecutionManagers*.

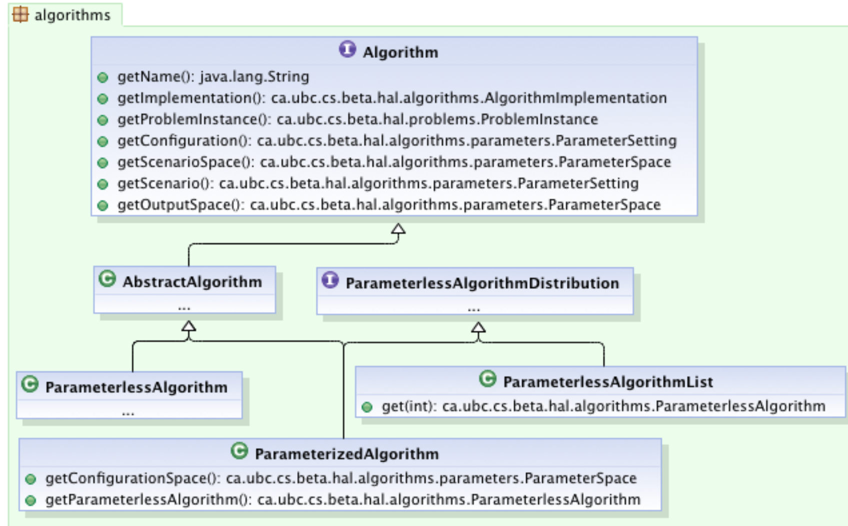

Figure B.6: Class diagram illustrating *Algorithms* in HAL 1.1.

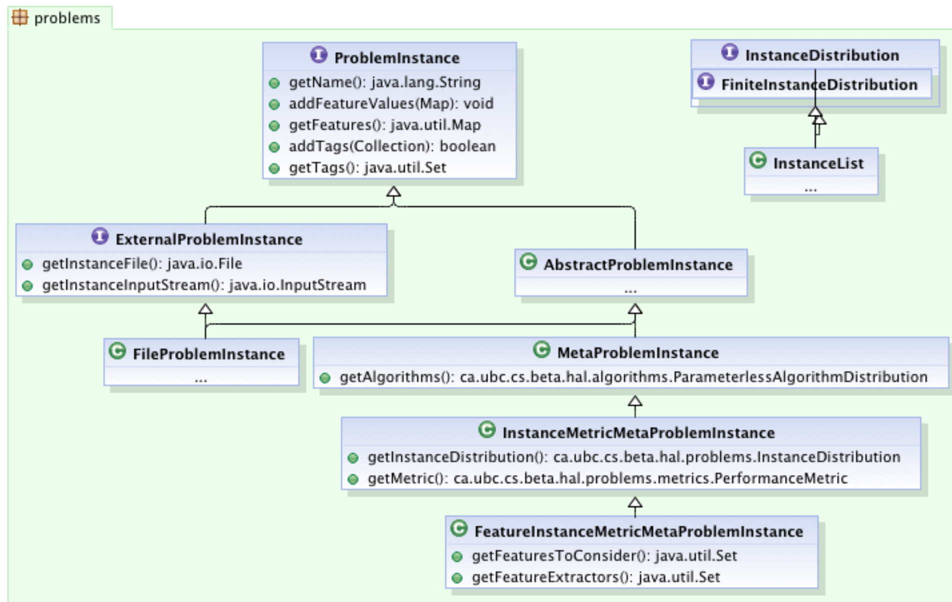

Figure B.7: Class diagram illustrating *ProblemInstances* in HAL 1.1.

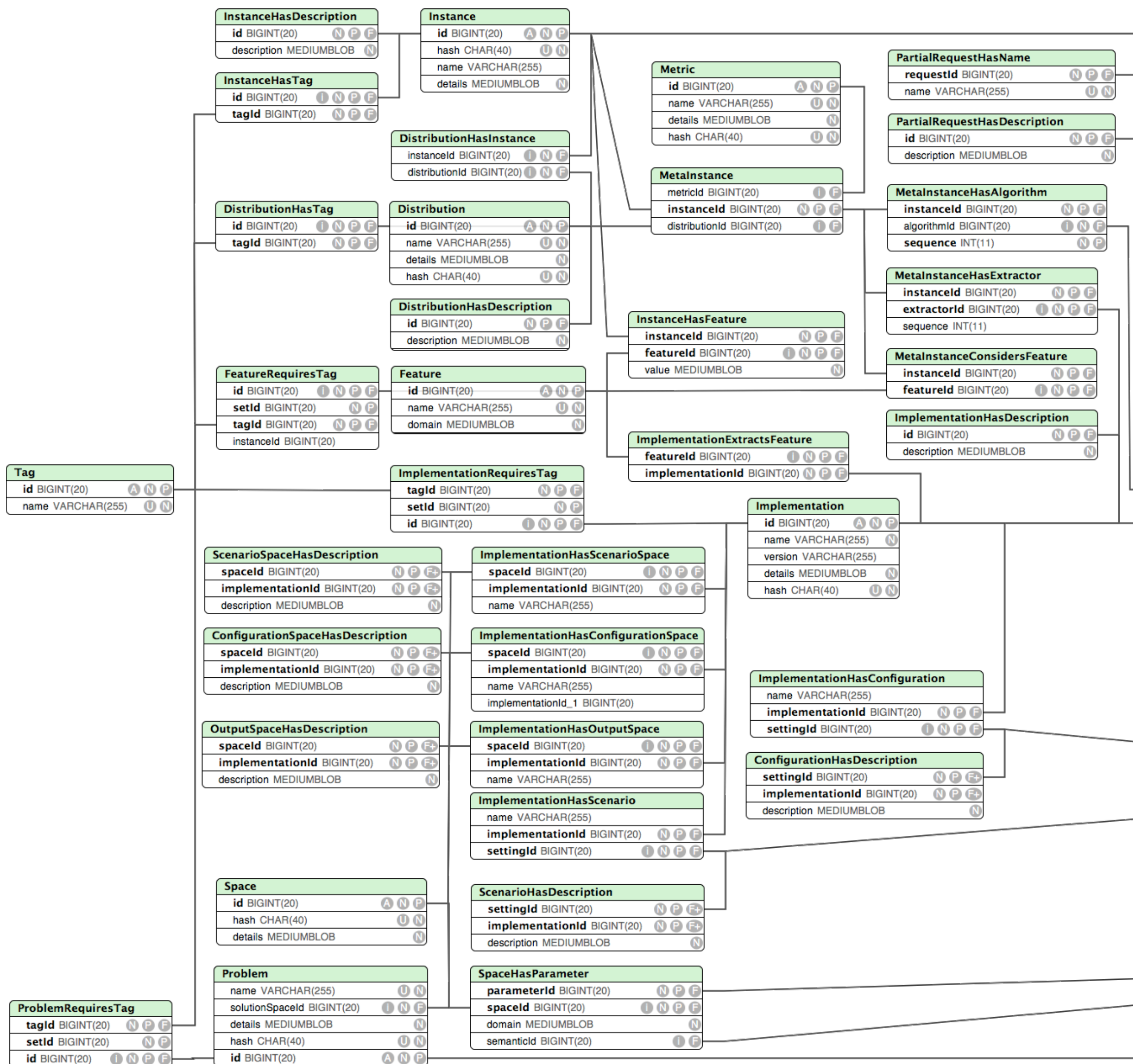

Figure B.8: Table schema used by SQL data managers (continued next page...)

Column Annotation Key	
i	indexed
N	not null
P	primary key
F	foreign key
U	unique
A	auto-increment

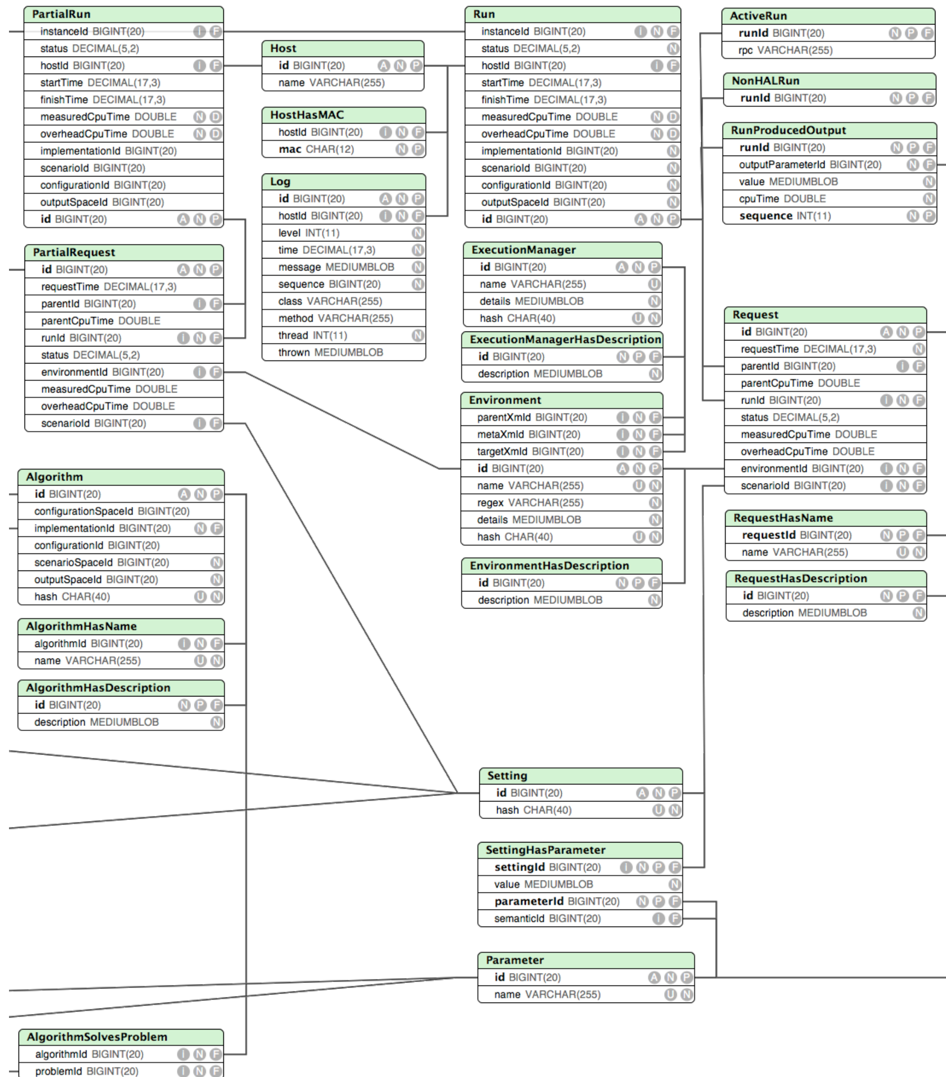

Figure B.8: Table schema used by SQL data managers. (continued)

Column Annotation Key

i	indexed
N	not null
P	primary key
F	foreign key
U	unique
A	auto-increment