

**SCHEDULELAB: AN ECLIPSE/OSGI BASED  
PLATFORM FOR EMPIRICAL ANALYSIS OF  
STOCHASTIC LOCAL SEARCH ALGORITHMS  
SOLVING RESOURCE SCHEDULING PROBLEMS**

by

Hussein Vastani

B.C.S, University of Pune, India, 1998

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE  
in the School  
of  
Computing Science

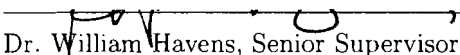
© Hussein Vastani 2008  
SIMON FRASER UNIVERSITY  
Spring 2008

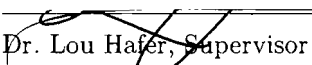
All rights reserved. This work may not be  
reproduced in whole or in part, by photocopy  
or other means, without the permission of the author.

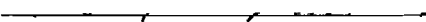
## APPROVAL

**Name:** Hussein Vastani  
**Degree:** Master of Science  
**Title of project:** ScheduleLab: An Eclipse/OSGi based platform for empirical analysis of Stochastic Local Search algorithms solving Resource Scheduling problems

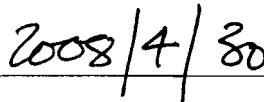
**Examining Committee:** Dr. Ze-Nian Li  
Chair

  
Dr. William Havens, Senior Supervisor

  
Dr. Lou Hafer, Supervisor

  
Dr. Ted Kirkpatrick, SFU Examiner

**Date Approved:**

  
\_\_\_\_\_



SIMON FRASER UNIVERSITY  
LIBRARY

## Declaration of Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website <[www.lib.sfu.ca](http://www.lib.sfu.ca)> at: <<http://ir.lib.sfu.ca/handle/1892/112>>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library  
Burnaby, BC, Canada

# Abstract

There is a shortage of software development tools that support researchers in academia and industry alike in experimentation for performance evaluation of resource scheduling algorithms based on stochastic local search (SLS) techniques. Given their stochastic nature researchers rely on empirical techniques for performance analysis of SLS algorithms. This work contributes an effort to develop such a tool, called ScheduleLab, based on the Eclipse/OSGi platform. The class of SLS algorithms is expansive, so we focus our efforts on SLS algorithms solving resource scheduling problems to control the scope of the work. The tool is non-invasive to the developer's code base and extensible. The tool focuses on supporting problem instance generation and providing an experimentation harness for performance evaluation of such algorithms. We illustrate the utility of the tool with experimentation on algorithms that solve the job shop scheduling problem.

# Acknowledgments

I would like to thank my senior supervisor Dr Bill Havens for his patience and generous support in seeing me through my MSc Project. Bill is a thoughtful, pragmatic teacher. Also my supervisor Dr Lou Hafer for his time and thoughts and my examiner Dr Ted Kirkpatrick for making time to examine my project report. I wanted to thank Computing Science advisor Margo Leight and Dr Fred Popowich, current Director of the CS Graduate Program, for their valuable advice and assistance in helping me return to finish my MSc at SFU. I want to acknowledge the general culture of supportiveness and the pursuit of knowledge and improvement at the CS graduate school that I've had another opportunity to avail of. I feel that I need more of it. Hopefully I can come back to school in the near future.

I would like to thank Dr Morten Irgens, Florissa Abreu and Junas Adhikary for their unwavering encouragement, support and friendship and for giving me an opportunity to work in a supportive and inspiring environment at Actenum Corporation. I wanted to thank Tom Carchrae and Peter Harvey of Actenum for important discussions and supporting contributions in this project.

My sisters, Salima and Shamina, have been key to my returning back to SFU to finish my degree. I want to thank Salima for pushing me to go see Margo Leight and to both of them for seeing me through the important last stretch of my final semester. I wanted to thank my parents for their support, love and sacrifices over the years. Finally I wanted to thank Salima (Isani) for her love and support, for inspiring me to engage positively with my work and for renewing my energy to build for the future.

# Contents

<b>Approval</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>iv</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Programs</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Contribution . . . . .	2
1.3 Report outline . . . . .	2
<b>2 Literature Review</b>	<b>4</b>
2.1 Empirical Analysis of SLS algorithms . . . . .	4
2.2 Generating Problem Instances . . . . .	8
2.3 Related work . . . . .	10
<b>3 System Architecture &amp; Design</b>	<b>13</b>
3.1 Desirable features . . . . .	13
3.1.1 Generating Problem Instances . . . . .	13
3.1.2 Support for Empirical Performance Analysis . . . . .	14
3.1.3 Software Engineering and Development considerations . . . . .	14

3.2	ScheduleLab . . . . .	15
3.2.1	Why Eclipse and Java? . . . . .	16
3.3	ScheduleLab Architecture . . . . .	18
3.3.1	User scenario . . . . .	19
3.4	Leveraging the Eclipse platform . . . . .	20
3.5	ScheduleLab core components . . . . .	24
3.5.1	Instance generator engine . . . . .	24
3.5.2	Sampling Engine . . . . .	27
3.5.3	Analysis and visualisation engines . . . . .	29
3.5.4	Persistence engine . . . . .	30
3.6	ScheduleLab extension points . . . . .	31
3.6.1	Problem Type extension point . . . . .	31
3.6.2	Problem Instance generator extension point . . . . .	33
3.6.3	Data collector extension point . . . . .	34
3.6.4	Data analysers and visualisers . . . . .	35
3.7	ScheduleLab User Interface . . . . .	37
3.8	Summary . . . . .	39
<b>4</b>	<b>Operation &amp; Evaluation</b>	<b>50</b>
4.1	Setup & Methodology . . . . .	51
4.1.1	Job Shop Scheduling Problem . . . . .	51
4.1.2	Algorithms . . . . .	52
4.1.3	JSP Problem Type . . . . .	53
4.1.4	Problem Instance Generation . . . . .	54
4.1.5	Experiment Session . . . . .	57
4.2	Analysis . . . . .	58
4.3	Experiment results . . . . .	58
4.3.1	Random instance results . . . . .	59
4.4	Structured instance results . . . . .	59
4.5	Discussion . . . . .	60
4.6	Summary . . . . .	62

<b>5 Conclusion</b>	<b>64</b>
5.1 Project Summary . . . . .	64
5.2 Contributions . . . . .	65
5.3 Future Work . . . . .	66
<b>A ScheduleLab Descriptor files</b>	<b>67</b>
A.1 JSP Problem Instance structure descriptor published by a problem type plugin	67
A.2 Problem instance type structure descriptor <i>targStruct.xml</i> from instance generator . . . . .	68
A.3 Problem instance type structure descriptor from target solver . . . . .	69
A.4 ScheduleLab project descriptor . . . . .	70
A.5 Instance ensemble descriptor file <i>ensemble.xml</i> . . . . .	70
A.6 Experiment session descriptor file <i>session.xml</i> . . . . .	70
A.7 Analyser descriptor file <i>analyser.xml</i> . . . . .	71
A.8 Visualiser descriptor file <i>visualiser.xml</i> . . . . .	71
A.9 Sample ScheduleLab instance generator plugin parameter descriptor <i>generator.xml</i> . . . . .	72
A.10 Sample ScheduleLab data-collector descriptor for <i>solution trace</i> data collector <i>dataCollector.xml</i> . . . . .	73
<b>B ScheduleLab extension points</b>	<b>75</b>
B.1 Java interface for ScheduleLab instance generators . . . . .	75
B.2 Java interface for ScheduleLab problem type extension point . . . . .	76
B.3 Java interface for ScheduleLab data collector . . . . .	76
B.4 Java interface for ScheduleLab data analyser . . . . .	77
B.5 Java interface for ScheduleLab data visualiser . . . . .	77
<b>C Output files</b>	<b>79</b>
C.1 Snippet of problem instance file generated by Taillard Uniform Generator for JSP . . . . .	79
C.2 Snippet of output produced by solution trace data collector . . . . .	81
C.3 Snippet of output produced by <i>QRTD</i> , <i>SQD</i> data analyser . . . . .	81
<b>Bibliography</b>	<b>84</b>



# List of Figures

3.1	ScheduleLab's contributed extensions to Eclipse workbench UI . . . . .	23
3.2	ScheduleLab internal structural relationships . . . . .	26
3.3	Sample ScheduleLab project structure . . . . .	36
3.4	New Projects wizard showing ScheduleLab Project wizard entry . . . . .	40
3.5	New Projects wizard showing ScheduleLab Project wizard entry . . . . .	41
3.6	New ScheduleLab Project wizard . . . . .	42
3.7	ScheduleLab Editor . . . . .	43
3.8	New ScheduleLab Problem Instance Ensemble wizard . . . . .	44
3.9	Instance Ensembles Master Detail view . . . . .	45
3.10	Experiment Sessions Master Detail view . . . . .	46
3.11	Running Experiment session run-sets . . . . .	47
3.12	Running Analyser on experiment session data . . . . .	48
3.13	Session Analysis data Visualiser master-detail view . . . . .	49
4.1	Combined SQDs for sample random problem instance . . . . .	60
4.2	Combined QRTD for sample random problem instance . . . . .	61
4.3	Combined SQDs for sample structured problem instance . . . . .	61
4.4	Combined QRTD for sample structured problem instance . . . . .	62

# List of Programs

3.1	Pseudo-code showing usage of Java <i>dynamic proxies</i> to register event handlers with solvers . . . . .	35
4.1	JSP problem instance type descriptor <i>targStructure.xml</i> . . . . .	55
4.2	Internal instance object skeleton for JSP problem-type . . . . .	56

# Chapter 1

## Introduction

### 1.1 Motivation

Local search and systematic search are contrasting search paradigms used to solve hard combinatorial problems. Systematic search algorithms scan the search space for a given problem instance systematically, such that if there is a solution to the problem then it will be found and if no solution is found that means that there is no solution for the problem instance. This is referred to as the *completeness* property of systematic search. Local search algorithms start at a point in the search space and move to a neighbouring point in the search space in search of the solution, using knowledge local to the neighbourhood of the previous search position. *Stochastic Local Search* algorithms are characterised by the use of randomization in the decisions to move through the search space.

Stochastic Local Search (SLS) algorithms are widely applied to solve academic as well as industrial decision and optimisation problems. SLS algorithms are typically applied to find in reasonable time best approximations to the optimal solutions of many NP-Hard problems where complete and optimal techniques would not be able to find solutions in a reasonable amount of time. Practitioners are faced with the issues of selecting the most appropriate algorithm from a set of algorithms for the task at hand or to gain deeper insights into the behaviour of algorithms. Mathematical analytical techniques would be the preferred choice [11, c.4] for the analysis of SLS algorithms. However mathematical analysis of SLS algorithms is hard given their inherently non-deterministic behaviour. Instead practitioners must base their understanding of the performance characteristics of a given SLS algorithm,

on its own and relative to other algorithms, on a chosen portfolio of empirical techniques. Over time, experts in the field have identified several best practices in this regard. There is also acknowledgement that this is an active research area [2]. The prevalence of real world problems that can be solved using SLS algorithms and the increasing recognition of the importance of optimisation in strategic and operational decision making means that industrial practitioners need to be supported with comprehensive tools that can enable them to efficiently analyse SLS algorithms. Currently there is a paucity of such tools for industrial and academic practitioners, leading to duplication of effort in this regard. We see this application area as having rich potential for continuous industrial-academic collaboration given the active synergy between industry and academia in the area of solving hard optimisation problems.

## 1.2 Contribution

This work contributes to addressing the need for tools for academic and industrial practitioners applying or studying SLS algorithms. We note that there is a growing focus on rigorous empirical analysis of SLS algorithms that is not matched by availability of tools to support efficient and effective application of important empirical analysis techniques. We present a proof-of-concept tool called ScheduleLab intended to fill this need. ScheduleLab is intended to be useful to both industrial as well as academic researchers by providing an extensible empirical analysis toolbox for researchers. We restrict our focus to Resource Scheduling SLS algorithms to control the scope of our work. We demonstrate problem instance generation and basic empirical analysis of the performance of two algorithms that solve the Job Shop Scheduling Problem. We discuss how to extend ScheduleLab to incorporate more advanced analysis and experimentation tools as needed.

## 1.3 Report outline

In chapter 2 we present a literature review looking at any tools that support empirical research. We also explore fundamental empirical research concepts and practises in the field that can help us identify features for design of a comprehensive tool. In chapter 3 we focus on the system architecture and design of our tool, ScheduleLab. We summarise what we think are the desirable features in an empirical analysis tool and we present ScheduleLab

a comprehensive tool that is based on the Eclipse IDE and written in the Java programming language. In chapter 4 we present an operational evaluation of ScheduleLab by doing preliminary empirical analysis on two SLS algorithms that solve the Job Shop Scheduling problem. We define the problem and walk through the steps of extending ScheduleLab to add the necessary capabilities to do the experimentation and discuss the results obtained. In chapter 5 we make concluding remarks and discuss thoughts about future work.

## Chapter 2

# Literature Review

ScheduleLab is a software tool that supports empirical analysis of resource scheduling SLS algorithms. In this chapter we discuss some fundamental concepts in the empirical analysis of SLS algorithms from the literature and explore related work.

### 2.1 Empirical Analysis of SLS algorithms

There are two ways to study the performance of algorithms [9, 11]: a theoretical approach and an empirical approach. An example of the theoretical approach is to use deductive mathematics to derive worst-case and/or average case analysis of algorithm behaviour. The empirical approach relies on computational experiments that eventually arrive at a predictive model that relates causal factors (such as problem instance characteristics and algorithm parameters) to algorithm behaviour. Theoretical analysis is considered to be a formal scientific technique, while empirical analysis unfairly suffers from the image of being less scientific or more of an art than a science. However, as compared to empirical analysis, theoretical analysis is usually significantly more laborious and less prolific, and can have limited practical applicability even when theoretical analysis results are available for a given algorithm. For instance, Simulated Annealing (SA) is theoretically proven to converge to the global optimum solution under certain conditions that includes an infinitesimally slow cooling schedule [8], which is not practical. Worst-case complexity results are asymptotic and represent a worst-case that may seldom occur. Average-case complexity analysis available for simple algorithms, are often based on problem instances sampled from random distributions that

are unlikely to be encountered in practise. Given the non-deterministic nature of SLS algorithms and the diversity of complex SLS techniques in the literature, current theoretical analysis techniques unfortunately fall short of enabling the researcher to gain practical and accessible insight into algorithmic behaviour. Therefore empirical analysis is the common methodology applied by SLS algorithm researchers. Further, there is a need to have at least a more principled approach to study algorithm performance using empirical analysis.

Hoos et al.[11] offers a comprehensive survey of techniques for empirical analysis of SLS algorithms, covering two categories of SLS algorithms: SLS algorithms that solve decision problems and those that solve optimisation problems. Our focus in this project is resource scheduling problems that are optimisation problems that require minimising/maximising some objective function. Hoos et al. [11] adopts the more general terminology of Las Vegas Algorithms (LVAs) in that SLS algorithms are a special case of LVAs. An LVA:

- Returns a correct solution when it finds one.
- Its run-time is a random variable.

Further, an Optimisation Las Vegas Algorithm (OLVA), is additionally characterised by solution quality that is a random variable. Examples of LVAs include SLS algorithms solving decision problems such as graph-colouring and satisfiability (SAT) problems. Examples of OLVAs include SLS algorithms solving optimisation problems such as travelling salesman problem (TSP), vehicle routing problem (VRP) and job-shop scheduling problem (JSP). Thus, due to their stochastic nature the run-time of SLS algorithms even on a single problem instance can vary across a sampling of independent runs. We could plot solution run-time (y-axis) for individual sample runs (x-axis) for a given problem instance. But researchers prefer to think of the run-time performance of an SLS algorithm as a cumulative probability distribution function called Run-Time Distribution (RTD) [11, c.4] giving a smoother and more informative graph of the run-time behaviour of the algorithm. The RTD is a function  $rtd(t)$  that defines the probability  $P_s$  of the run-time  $RT$  of algorithm  $A$  of successfully finding a solution for instance  $\pi$  as:

$$rtd(t) = P_s(RT_{A,\pi} \leq t) \quad (2.1)$$

In addition to being characterised by run-time performance, SLS algorithms solving resource scheduling problems are solving optimisation problems in that they are also characterised

by solution quality as represented by an objective function that must be minimised or maximised. In the travelling salesman problem (TSP) [11, c.8] the objective is to minimise the total edge weights of the Hamiltonian cycle going through all the nodes (cities). In the job-shop scheduling problem (JSP), usually we concern ourselves with minimising the makespan (maximum completion time of all jobs). Thus the performance of resource scheduling SLS algorithms is characterised by both run-time as well as solution quality in the form of a bivariate probability distribution over these two variables. Of course a bivariate probability distribution plot gives the researcher a visually complete perspective of the empirically determined performance characteristics of a given optimisation SLS algorithm. But researchers prefer to work with the following marginal univariate probability distributions derived from the bivariate distribution (reproduced from [11]) :

- *Qualified Run-time distribution*: If  $rtd(t, q)$  is the bivariate *RTD* of optimisation SLS algorithm  $A$  on instance  $\pi$  then for any solution quality  $q'$ , the qualified run-time distribution (*QRTD*) of  $A$  on  $\pi$  given  $q'$  is defined by the distribution function:

$$qrtd_{q'}(t) := rtd(t, q') = P_s(RT_{A,\pi} \leq t, SQ_{A,\pi} \leq q') \quad (2.2)$$

where  $RT$  is the algorithm's run-time and  $SQ$  is the solution quality.

- *Solution Quality Distribution*: If  $rtd(t, q)$  is the bivariate *RTD* of optimisation SLS algorithm  $A$  on instance  $\pi$  then for any run-time  $t'$ , the solution quality distribution (*SQD*) of  $A$  on  $\pi$  for  $t'$  is defined by the distribution function:

$$sqd_{t'}(q) := rtd(t', q) = P_s(RT_{A,\pi} \leq t', SQ_{A,\pi} \leq q) \quad (2.3)$$

We think that the preference for working with univariate probability distributions derived from the bivariate distribution of an optimisation SLS algorithm is because researchers prefer to apply common empirical analysis techniques for decision-variant SLS algorithms and optimisation-variant SLS algorithms. In optimisation-variant SLS algorithms, *QRTDs* are useful for characterising the run-time ability of an SLS algorithm to find the optimal or near-optimal solution quality for a given instance. Similarly *SQDs* are useful to characterise the solution quality performance of an algorithm in a given time-limit.

*QRTDs* and *SQDs* for every instance are the fundamental performance observations researchers can make about a given SLS algorithm solving (optimisation) resource scheduling



problems. As noted by [11, c.4] we can obtain both *QRTDs* as well as *SQDs* by collecting the solution traces of each of  $k$  independent runs of optimisation SLS algorithm  $A$  on instance  $\pi$ . A solution trace is the pair  $(RT, SQ)$  that is reported every time the algorithm finds an improved solution. For  $k$  independent runs, let  $sq(t, j)$  represent the solution reached by run  $j$  by time  $t$ . Then the empirical *QRTD* of  $A$  on  $\pi$  is defined as the following probability:  $P_s(RT \leq t, SQ \leq q) := \#\{j | sq(t, j) \leq q\} / k$ . In addition to being able to obtain distributions and summary statistics for the performance of optimisation SLS algorithm  $A$  on an instance  $\pi$  researchers also need to characterise the performance of  $A$  on an ensemble of instances. Further, researchers need to compare the performance of algorithms both on single instances as well as on an ensemble of instances. This entails obtaining descriptive statistics like mean, quantiles, quantile ratios, performance variation coefficient and performing statistical tests based on the *QRTDs* and *SQDs* of the algorithms. During characterising algorithm performance, researchers may need to qualify *QRTDs* with the value of other factors such problem instance size or algorithm tuning parameters [14].

In the above distributions solution quality is measured as relative solution quality. For a minimisation problem, relative solution quality is defined as  $(q/q^*) - 1$ , and for a maximisation problem it is  $(q^*/q) - 1$ , where  $q$  is solution quality achieved by the algorithm and  $q^*$  is the optimal solution quality. When the optimal solution quality is not known, then a tight lower bound (in the case of minimisation) can be used. For a minimisation problem like traveling salesman problem (TSP) the optimal solution quality would be the least cost hamiltonian cycle and for a JSP it is usually the minimum makespan (maximum completion time of all jobs). Thus relative solution quality is the percentage deviation of the solution quality found from the optimal solution value. The closer to 0 the better.

As noted above, *QRTDs* and *SQDs* are the fundamental empirical observations of the performance of an optimisation SLS algorithm on a given instance. They can form the basis for further visual and statistical analysis. Basic statistics such as mean, median, quantiles and standard deviation can be obtained for each *QRTD* and *SQD*. A solution quality distribution over Time (*SQT*), shows the development of a statistic of solution quality over time. In effect an *SQT* represents the plot of a *SQD* (solution quality distribution for time  $t$ ) statistic such as mean or median solution quality taken from a series of *SQDs* corresponding to an entire time interval (*SQD* for  $t_1$ , *SQD* for  $t_2$ , etc). Quantiles are preferred

statistics over means because of their inherent stability. Combinations of *SQTs* are useful for illustrating trade-offs between run-time and solution quality for an entire series of *SQDs*. The orthogonal notion of *SQT* is a qualified run-time distribution statistics dependent on solution quality (*RTQ*). Here, we plot a statistic of run-time performance taken from a series of *QRTDs* corresponding to an entire solution quality interval (*QRTD* for  $q_1$ , *QRTD* for  $q_2$ , etc.)

While measuring run-time, one can measure operation counts (constant time) in addition to cpu-time. In fact when the algorithm is run on different machines with different run-times it is necessary to measure operation counts for comparability. For SLS algorithms a local search move can be used as a unit operation count. When operation counts are used instead of CPU seconds, run-time distribution is referred to as run-length distribution (RLD) [11][c.4].

## 2.2 Generating Problem Instances

Researchers characterise the behaviour of a given SLS algorithm or a set of competing algorithms by running them against one or more problem instances. For example, given two SLS algorithms *A* and *B* that solve Job Shop Problems (JSP) we may want to ask what is the expected run-time (in CPU seconds) of each algorithm when solving JSP instances to optimality. Ideally we want our algorithms to be robust, that is our algorithms should be able to find optimal solutions to any problem instance, however difficult. The issue of using relevant and representative problem instances is an important issue to address in the empirical analysis of heuristic algorithms. Hooker [10] laments a tendency in the heuristic algorithms literature for "competitive testing". By competitive testing he means the practise of over-fitting a new algorithm implementation in order to perform favourably against an existing state of the art algorithm, solving a fixed set of benchmark problem instances. From a research perspective this is a fruitless exercise because it yields no new insights into why the new algorithm performs well (or doesn't) on the benchmark instances. Further, from a practical standpoint it is necessary to ask if the chosen ensemble of benchmark instances is representative enough. If the algorithm performs favourably on one ensemble of benchmark instances can we expect it to perform well on a different ensemble of instances? A

more appropriate endeavour would be to characterise the performance of the new algorithm with respect to characteristics of the problem instances. For example, how does problem size affect performance, or how does the combination of problem size and algorithm parameter values affect algorithm performance. Hooker [9] promotes the practise of controlled experimentation using sound statistical techniques to arrive at a predictive model that can explain and predict why a given algorithm performs well or poorly against specific problem characteristics. It is generally believed in the literature that random synthetic instances are usually more difficult to solve than structured problem instances found in real world situations. This is because researchers can design algorithms that exploit the structure of the problem instances. This means that testing algorithms solely on random instances may say nothing about how the algorithm would perform in a practical setting. However it is not always the case that an structured problem instances are easier to solve than random instances, as noted by [18]. In general it is safe to say that testing algorithm performance against difficult random instances alone may not be sufficient. In fact in practical applications, it maybe sufficient to use an algorithm that is not a stellar performer on random instances because the typical instances are much easier to solve.

Whitley et al. [21, 20] acknowledge that "horse-race" competitive testing has serious disadvantages but in practise faced with an optimisation problem class having specific characteristics, researchers and practitioners still need to recommend one algorithm over others. So we still need to be able to do comparative evaluations of algorithms' performance. Whitley emphasises specifically the need to consider testing on structured problem instances as well as the traditional unstructured or random problem instances. Comparative evaluation of algorithms should be qualified by the characteristics of test problems.

There a numerous examples in the literature of methods to generate random as well structured problem instances of a particular problem type. For example for JSP, Taillard [17] proposes a simple technique to generate random JSP instances of any size in which job operation durations are sampled from a random uniform distribution and machine ordering for a job is a random permutation. Watson [18] presents two techniques to create structured JSP problem instances: machine correlated instances have the duration of a job operation sampled from a gaussian distribution specific to its machine; job correlated instances have the duration sampled from a gaussian distribution specific to the operation's job. We are not

aware of any software tools or frameworks in the literature that support generating problem instances. Some authors publish all or a sample of their generated benchmark instances [17] while others simply describe the problem generating code that maybe available by personally contacting the authors. Even when the instances are published they are usually in a non-standard format and the users are expected to write their own code to read/write these instances. As a result there is a lot of duplication of effort.

In ScheduleLab we support the development and sharing of custom parameterized problem instance generators thereby encouraging researchers to consider testing on both structured as well as unstructured instances. Further, users can share and import problem instance ensembles. The instances are saved in a consistent format and ScheduleLab can readily read/write these instances.

## 2.3 Related work

To the best of our knowledge there is a paucity of comprehensive software tools that support researchers to do empirical analysis of SLS algorithms. This is noted by Hoos et al. [11][Epilogue, pp. 533-534] as an important area for further work. The one comprehensive software tool found in the literature is EasyAnalyzer [7].

EasyAnalyzer is an extensible framework written in C++ to support empirical analysis on SLS algorithms. It is independent of the underlying solver or algorithm implementation. It covers a comprehensive array of empirical analysis techniques for SLS algorithms:

- Search Space analysis: how the topography of the search space affects performance.
- Runtime analysis: Run-time distribution (RTD), Run-length distribution (RLD), Solution Quality Distribution (SQD).
- Comparative analysis: of solvers, and of parameter configurations for a single solver.

EasyAnalyzer aims to make SLS practitioners more productive by providing scaffolding based on an inversion of control design to support the empirical analysis of SLS algorithms. By inversion of control we mean that the framework effectively has a "don't call us, we'll

call you” policy in that the researcher simply provides the framework specific pieces of functional implementation. The framework consists of a top-level layer of *Analyzers* that are responsible for coordinating execution and delegating (making call-backs) to the specialized functionality provided by the researcher when appropriate. An *Analyzer* implements the control logic for a particular analysis task (search space analysis, run-time behaviour analysis, comparative analysis). The researcher is expected to provide a possibly re-usable integration layer between *Analyzers* and the underlying solver or algorithm implementation. This integration layer provides hooks into the underlying solver in order to get the data for the various *Analyzers*. The integration layer includes providing implementations for:

- *StateManagerAdapter*: it provides methods to enumerate and sample the search space independently of the neighbourhood function and to calculate the cost function value on a given state.
- *NeighbourhoodExplorerAdapter*: it provides methods to enumerate the neighbourhood of a given search space point and evaluate the cost function value.
- *SolverAdapter*: it provides methods to execute a complete run returning running time and cost function values for the trajectory of search states explored.

The researcher states which analysis is needed by specifying the appropriate command line parameters and supplying his implementation of the solver(s) that implements the integration layer mentioned above. Unfortunately, its not clear from the description of EasyAnalyzer [7] if the researcher is required to provide an implementation for all three of the above solver specific components even if he is only interested in performing, say, run-time behaviour analysis. That would force the user to write unnecessary code, taking away some of the benefits of using a scaffolding such as EasyAnalyzer that is meant to minimise the amount of user written code for the task.

EasyAnalyzer offers comprehensive empirical analysis features but it also induces a tight coupling between the user code and EasyAnalyzer. This can be a concern in industrial scenarios where the optimisation code is integrated as part of a much larger system and introducing a compile-time dependency on EasyAnalyzer may not be practical as it may further complicate the software build process. ScheduleLab offers an alternate approach in which we focus on providing compile-time decoupling between user code and ScheduleLab.

This means that the user's algorithm code is not expected to extend any abstract classes or implement any interfaces in order to integrate with ScheduleLab. Instead ScheduleLab relies on user specified meta-data to integrate with the solver at run-time using the Java platform's run-time introspection capabilities [13]. Unlike ScheduleLab, EasyAnalyzer has no support for problem instance generation and by delegating the monotonous task of reading/writing problem instances to the underlying user code, it does not address the issue of duplication of effort in this respect.

ScheduleLab currently lacks the breadth of concrete empirical analysis features offered by EasyAnalyzer, but as explained in the next chapter, ScheduleLab's architecture supports extensions that could provide more advanced analysis features. As an alternative to command-line run EasyAnalyzer, ScheduleLab is a Java IDE (Eclipse [3]) based tool that is also an extensible inversion of control scaffolding. We believe that implementing ScheduleLab as a Java IDE based tool enhances usability. Unlike EasyAnalyzer, ScheduleLab does not require the user to implement a tight integration layer for every solver library used so that the scaffolding can extract data from the solver for analysis. EasyAnalyzer seems to assume that the solver library is the entry-point to the algorithm solving the problem instance. We think that this is seldom the case. Mostly, the solver library (EasyLocal++, ConstraintWorks, COMET, etc) is used behind the scenes by a wrapper component that is the entry point to the algorithm implementation code. This wrapper component is responsible for taking a problem domain specific instance object, doing any pre-processing, setting up the underlying solver and then running the solver. In a sense the wrapper component is a black-box providing solving capability using an underlying solver library behind the scenes. Any empirical analysis tool would need to integrate with such a solving component without direct integration with the underlying solver library. ScheduleLab does not expect to integrate directly with the underlying solver library. Instead ScheduleLab requires the user's solving component to exhibit behaviour that ScheduleLab can integrate with at run-time. In particular for run-time behaviour analysis ScheduleLab expects the user's code to generate events on solution improvements and to provide methods to register event listeners, query the run-time, operation count and solution quality. The user is expected to declare in the meta-data the signature of the methods and the event listener interface. ScheduleLab relies on the Java reflection API to integrate with the solving component using the user specified meta-data.

## Chapter 3

# System Architecture & Design

### 3.1 Desirable features

Before we present the architecture and design of ScheduleLab, it would be useful to summarize what we think are the desirable features of a tool that supports empirical analysis of resource scheduling SLS algorithms.

#### 3.1.1 Generating Problem Instances

As explained in section 2.2 in order to test for robustness we need to test our algorithms against problem instances with varying characteristics including random and structured problem instances. Further researchers should be able to re-use problem instance generators and to share benchmark instances without having to duplicate the code to read/write problem instances.

An empirical analysis tool should support generating new random as well as structured problem instances. Researchers should be able to write custom instance generators as well as share generators. It should be possible to write parameterized generators with support for validation of input provided by the user. It should be possible to publish instance ensembles and use ensembles generated by others in a seamless way. The tool should free researchers from duplicating effort in writing code to read/write problem instances.

### 3.1.2 Support for Empirical Performance Analysis

We saw in section 2.1 that the fundamental observations that a researcher can make about the performance of an SLS algorithm solving resource scheduling problems are *QRTDs* (qualified run-time distributions) and *SQDs* (qualified solution quality distributions). These can form the basis for further visual and statistical analysis. The tool should support a workflow that includes:

- create or import problem instance ensembles
- integrate algorithms to conduct experiments on
- specify sample size and algorithm parameters
- collect solution traces for every combination of an algorithm and problem instance.
- analyse the solution traces to yield *QRTDs*, *SQDs*, *SQTs* and basic descriptive statistics such as mean, quantiles and standard deviation.
- provide support to do statistical tests to determine if statistics (mean, quantiles) of the distributions of two algorithms are different.
- provide visualisation support to view distributions in semi-log or log-log plots if desired.

### 3.1.3 Software Engineering and Development considerations

SLS researchers include both academic as well as industrial users. From a user's point of view a tool for empirical analysis of resource scheduling SLS algorithms must be easy to use and it should leverage the user's existing development environment. We believe that graphical user interface (GUI) based development environments are much easier to use and enjoy wide acceptance amongst academic as well as industrial developers, as compared to command line development environments. Further, forcing the user to learn a new programming language is not a practical expectation at least of industrial users. From the point of view of developing the tool itself, re-inventing the wheel in terms of creating a new integrated development environment (IDE) from scratch that is dedicated to empirical analysis would be undesirable. It would also force the user to work with separate IDEs that offer different functionality, operating on shared code. In other words, the tool should integrate



with popular software development contexts so as to minimise the complexity of learning and using the tool.

In industrial projects the algorithm implementations are usually a part of a software component that fits into a larger system of components. The empirical analysis tool should be easy to integrate with in that the tool should have minimal or no compile-time impact on the user's algorithm implementation code. For example if the tool is a library that exposes its application programming interface (API) [5] and expects the user to extend certain base classes in order to use the tool to do empirical analysis then that is undesirable because it introduces a compile time dependency between the empirical analysis tool library and the entire code base of the user. This may not be acceptable in some development projects and would introduce additional complexity in the software build, test, release process.

The empirical analysis tool should support re-usability and minimise duplication of effort in the practise of empirical analysis. When researcher  $R_1$  publishes benchmark JSP problem instances, and researcher  $R_2$  wants to do empirical analysis using those instances or create more instances based on the original instance distributions then  $R_2$  without any help from  $R_1$  must re-implement the code to read and generate instances, implement  $R_1$ 's algorithm, etc. This would be mitigated if  $R_1$  could share his code, data and analysis in a format that  $R_2$  can seamlessly consume and extend, knowing that they're both working on the same class of resource scheduling SLS problems. It should be possible for users to contribute new re-usable extensions to the tool such as problem instance generators, new problem type capabilities, new experimentation, data analysis and visualization capabilities.

## 3.2 ScheduleLab

We now introduce ScheduleLab, our proof-of-concept tool for empirical analysis of resource scheduling SLS algorithms. In order to control the scope of our work we focus on a subset of the features introduced in Section 3.1. These features will be elaborated upon in later sections. ScheduleLab is a software development tool to assist the user to perform empirical analysis on one or more resource scheduling SLS algorithm implementations. Features supported by ScheduleLab are, support for:

- defining new resource scheduling SLS problem types, such as job shop scheduling

problem, vehicle routing problem with time window constraints, etc.

- instance Ensemble generation.
- performing sample runs of one or more algorithms on one or more instances in order to collect solution traces for data analysis.
- data analysis to obtain *RTQs* and *SQDs*.
- visualizing algorithm performance distributions.
- customizing and contributing new capabilities in each of the above areas.
- Minimal impact on target user code. ScheduleLab uses meta-data to integrate with the target code at run-time and imposes no compile time dependencies on the target user code.

ScheduleLab is implemented in the Java programming language as a plugin of the Eclipse IDE [3].

### 3.2.1 Why Eclipse and Java?

Traditionally SLS algorithms in the literature tend to be implemented on C/C++ because of the sheer speed of natively compiled code as compared to interpreted execution. However, C/C++ has some disadvantages such as the developer is responsible for memory management (which can be a tedious and error prone exercise) and the compiled and linked program is not generally portable. The usual argument against using an interpreted language such as Java for SLS algorithms is that Java is slow. It is true that Java is slower than C/C++ however there are some important advantages in using Java over C/C++. One can argue that Java's automatic memory management, platform independence (compiled Java bytecode is mostly portable), proliferation of open source Java software libraries and development tools, its ubiquity as a programming language both in industry as well as academia and vibrant developer community means that the average developer will be more productive writing software in Java to solve practical enterprise related problems. Further with advances in hardware performance and affordability as well as in just-in-time (JIT) compilers for the Java Virtual Machine (JVM), the performance difference between Java and C/C++ code has diminished, albeit far from completely. Java particularly suffers from a relatively slow

startup phase as the JVM loads the large platform libraries into memory. However the benefits of the JIT compiler become apparent as the JVM is kept running for longer periods. Thus we believe that in real world applications that are either server-side or client-side Java applications where the SLS algorithm is part of a larger system, implementing the algorithm in Java is a reasonable choice. Further, it is possible to interface Java with C/C++ using the Java Native Interface (JNI) bridge, effectively accessing C/C++ code with a thin Java wrapper.

IBM's Eclipse IDE ([www.eclipse.org](http://www.eclipse.org)) is an open-source software development platform that is written in Java and based on an implementation of the OSGi runtime. It is the most popular IDE in the Java development space, circa 2008. Even though Eclipse is written in Java it supports writing software in a wide array of languages including C/C++, JavaScript, Perl, PHP and of course Java. IBM intended Eclipse to be an extensible development platform to which developers could add new capabilities dynamically at run-time as platform plugins. IBM chose to implement the Eclipse plugin kernel to comply with the OSGi specification. OSGi (a legacy acronym that stands for Open Services Gateway initiative) is a module system specification for Java ([www.osgi.org](http://www.osgi.org)), that emerged out of the space of Java for embedded devices. It allows OSGi modules called bundles or plugins to dynamically be part of an in-JVM network of services that can be consumed by other services or objects. Essentially each plugin either extends a base class or implements an interface and the OSGi runtime then instantiates the plugin when there is a need for that service. Services can be added or removed dynamically in an OSGi runtime. Eclipse itself provides a powerful environment to write plugins for the Eclipse platform.

Eclipse is a mature extensible development platform with an expansive user community. Implementing ScheduleLab as an Eclipse/OSGi plugin means that we can build on the existing extensive user interface infrastructure provided by Eclipse and that we can integrate with the user's preferred software development environment. This would contribute significantly to usability. Further, Eclipse allows plugins to be extensible themselves by declaring appropriate plugin extension-points. Other developers can contribute extensions to ScheduleLab's extension-points. Thus we can achieve our goal of making ScheduleLab itself be extensible.

### 3.3 ScheduleLab Architecture

ScheduleLab broadly comprises of two parts: the core ScheduleLab experimentation framework and the ScheduleLab extension-points. The core ScheduleLab experimentation framework comprises of the following capabilities that are internal to the tool:

- Instance generator engine: given the appropriate parameters – problem type, instance generator, relevant parameters – the instance generator engine coordinates instance ensemble generation.
- Sampling engine: it is responsible for coordinating the running of experiments given instance ensemble, target solvers, problem type, data collectors, relevant algorithm parameters.
- Analysis and visualisation engine: it is responsible for coordinating data analysers to analyse the data and visualises to view results.
- Persistence engine: it is responsible for the efficient disk storage and retrieval of objects like problem instances, solution traces, analysis results, etc. in Extensible Markup Language (XML) format.

ScheduleLab defines the following primary extension-points for extensibility.

- Problem type extension-point: Allows new problem type capabilities to be added to ScheduleLab. Each problem type capability defines the resource scheduling SLS problem class it supports (e.g. JSP), declares the expected structure of a problem instance of this class and provides a transformer service to transform between (bi-directional) any external problem instance type to an internal problem instance type of this problem class. We explain in later sections the details of this feature.
- Problem instance generator: Adds one or more new instance generators for a given problem type to ScheduleLab. Each instance generator must also declare how its internal representation of an instance object corresponds to the instance structure expected by the problem type plugin.
- Data collector: Adds new data collection capabilities to ScheduleLab. The basic data collector knows how to connect to a solver and collect solution traces of a given sample

run. Another data collector may be focussed on collecting data to study search space topology.

- **Data analyzer:** Adds new data analysis capabilities to ScheduleLab. Declares the data collector it is compatible with and processes data collected from sample runs for visualization and/or statistical analysis.
- **Data visualizer:** Adds new data visualization capabilities to ScheduleLab. Declares the data analyzers it is compatible with and provides graphical views of analyzed or raw data.

In later sections we elaborate on the above components and extension points of ScheduleLab.

### 3.3.1 User scenario

The following is the user scenario supported by ScheduleLab

- The user creates a new ScheduleLab project in Eclipse, specifying the name and location on disk and the problem type of interest. The user adds the Eclipse projects that have the target algorithms as project references to the new ScheduleLab project.
- From the project the user can choose to add a new problem set ensemble by choosing an instance generator and specifying parameters such as number of instances, size of the problem and any generator specific parameters.
- The user can then create a new experiment session, specifying the target algorithms and their run-time meta-data, the intended data collector and one or more run-sets. A run-set is a configuration of algorithms, instances and algorithm parameters. The user can then launch the experiment session and ScheduleLab will execute each of the specified run-sets.
- The user can select an action to analyze the data collected in the previous step.
- The user can select an action to visualize the results of data analysis.

### 3.4 Leveraging the Eclipse platform

ScheduleLab is implemented as an extensible Eclipse plugin to be used by Java developers. Eclipse allows plugin developers to contribute new functionality to the Eclipse IDE such as (see [4] for a comprehensive study of Eclipse plugin development):

- **Projects:** Eclipse allows the user to create projects (top-level folders) in the Eclipse workspace. A project is of a particular type. The most common project in eclipse is a Java project. A Java project allows developers to create packages and Java source files in the project. The project has a build-path (Java classpath) that can be configured through the user-interface by the user and can include libraries (.jar files internal and external to the project) as well as other Java projects in the Eclipse workspace. Plugin developers can contribute new project types to Eclipse. A project of a particular type, say Java project, is tagged as having a corresponding project *nature*. A Java project is tagged as having a Java project nature. Various user interface (UI) components in the Eclipse environment customise their behaviour in response to the presence of a particular project nature. For example, when the user selects a Java project and brings up the *Properties* dialog, the dialog shows the *Build Path* properties page that allows the user to set the build-path for the project. But doing the same for a plain project does not show the *Build Path* properties page.
- **Editors:** The notion of editors in Eclipse is more general than the usual text editor available in any operating system. An Eclipse editor allows the user to change the contents of an underlying workspace resource (usually a file) but its UI may take a form different from the typical window that allows the user to edit text. For example an Eclipse editor for an XML file could be a master-detail UI, showing the XML elements as a tree, on the left, forming the master view. When the user selects an element in the tree then the detail view on the right, shows a form with text fields that allow the user to edit XML element attribute values. Eclipse provides some base classes for writing custom editors. Overall the plugin developer is responsible for implementing the complete UI functionality (UI design, editor actions and model-view synchronization). Eclipse allows the plugin developer to associate the editor to a specific file type (.xml, .slab) or a specific file name so that when the user double-clicks on such a file, Eclipse automatically instantiates the plugin developer's editor to edit

the file.

- Views: Eclipse views are simply arbitrary UI components that conventionally are used to provide a visual representation for some underlying workspace resource. Views can use the entire workspace as their underlying resource (as in the *Navigator* view that displays a file system view of the workspace projects) or only the currently selected or active workspace resource (such as the *Outline* view that displays the structure of the Java source file currently being edited). Plugin developers can contribute new views to the Eclipse IDE, that can be opened by user action or programmatically.
- Wizards: Eclipse offers a wizard user interface framework and allows a plug-in developer to contribute a wizard of a pre-defined category. A wizard is a dialog window that pops up on specific user actions, such as create a new project or export a workspace resource into a specific format. It guides the user through a series of steps to elicit the required data input for the task being requested. For example, a plug-in developer can contribute a new project creation wizard by declaring a contribution to the `org.eclipse.ui.newWizard` extension point and providing an implementation of the interface `org.eclipse.ui.INewWizard` and possibly extending the base class `org.eclipse.jface.wizard.Wizard`.

Our ScheduleLab plugin contributes a new ScheduleLab project type as well as a new integrated editor that provides the main UI for ScheduleLab and a new ScheduleLab project wizard. Fig. 3.1 shows the UI contributions made by ScheduleLab to the Eclipse workbench (Eclipse IDE user interface). Contributing the editor means declaring the contribution to the extension point `org.eclipse.ui.editors` in the ScheduleLab `plugin.xml` plugin descriptor using the PDE, and providing an implementation of the interface `org.eclipse.ui.IEditorPart`. More details of the ScheduleLab UI are presented in section 3.7. The editor effectively gives the user a single entry point that allows the user to edit a set of meta-data files that are created inside the user's ScheduleLab project. Meta-data files store information that includes the project name, the project's problem type, problem instance ensemble descriptions and experiment session descriptions. The meta-data files are usually in XML format and are meant to be transparent to the user. The user only interacts through the ScheduleLab editor UI which supports a high-level workflow as outlined in section 3.3.1. The user's actions, choices and results are saved as files in the user's ScheduleLab project. We will see more details of the ScheduleLab project and editor functionality

later in the chapter. Like other projects in the Eclipse workspace, a user will be able to export a ScheduleLab project to a source control repository like CVS or Subversion using the excellent plugins available for this purpose in Eclipse.

A ScheduleLab project contains the following top-level folders and files:

1. *sLabProject.xml*: A sample ScheduleLab descriptor file is show in section A.4. A given ScheduleLab project can have only one problem type (e.g. JSP, Vehicle Routing with Time Windows [11]).
2. *ensembles* folder: The *ensembles* folder contains sub-folders that correspond to problem instance ensembles for the project's problem type. Details are provided in section 3.5.1.
3. *sessions* folder: The *sessions* folder contains sub-folders that correspond to experiment sessions. Each experiment session includes setting up algorithms, a data-collector and run-sets. The session sub-folder contains the experiment raw-data and output from analysers. Details are provided in section 3.5.

Eclipse has an extensible plugin architecture in which a plugin can declare extension points that other developers can contribute to using the robust Eclipse plugin development environment (PDE) that is part of the IDE. In fact all of Eclipse is built as plugin extensions around a core OSGi kernel called Equinox [6]. ScheduleLab defines a set of extension points (detailed later) such as problem instance generators, data analysers and visualizers that other developers can write and share with other ScheduleLab users promoting re-use and minimising duplication of effort. The latter can install these extension plugins using the standard ways of deploying plugins in eclipse [4, 6]. It is important to keep in mind that an Eclipse plugin or plugin extension when activated is a single instance of that plugin in the Eclipse OSGi run-time. Typically the activated plugin provides services (e.g. compiling, editing, viewing Java source files) to process Eclipse workspace related resources such as files and projects. Therefore the plugin code is usually stateless, but Eclipse itself does not require that design convention.



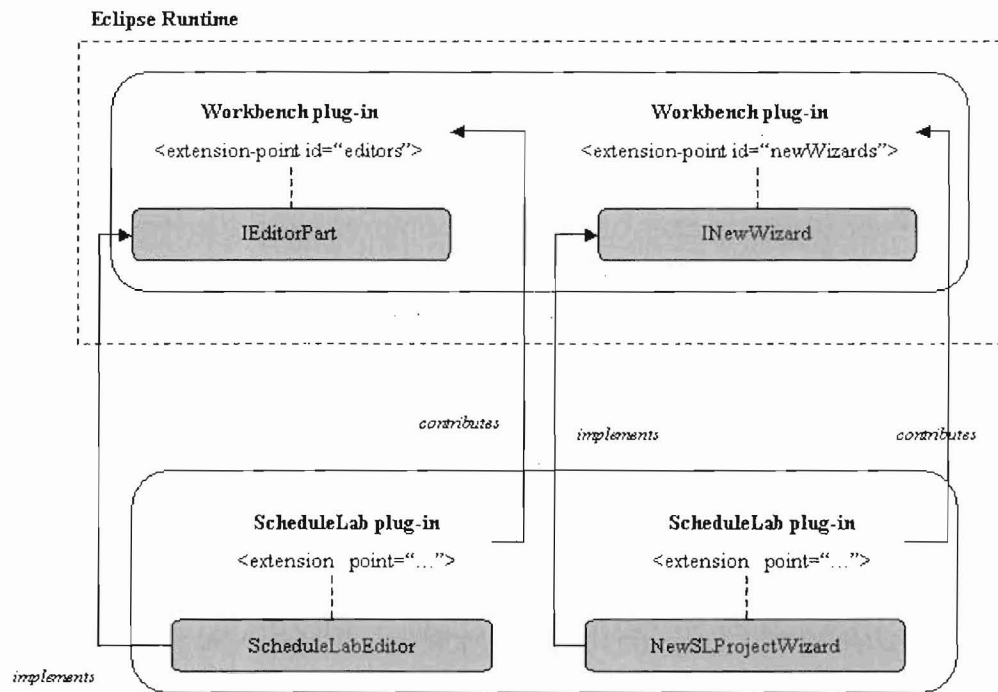


Figure 3.1: ScheduleLab's contributed extensions to Eclipse workbench UI

## 3.5 ScheduleLab core components

ScheduleLab core components are internal components that take as input the meta-data obtained from the user and create or update artifacts in the ScheduleLab project. The meta-data includes descriptors for the project, problem instance generation, solvers and experiment sessions. The artifacts produced by the core components include experiment raw-data, problem instances and analysis data. The core components are described below:

### 3.5.1 Instance generator engine

The instance generator engine is responsible for coordinating the creation of new problem instances according to the information supplied by the user. The instance generator engine requires as input:

- the location of the user's ScheduleLab project in the Eclipse workspace,
- the unique name of the problem instance ensemble specified by the user. This must correspond to a sub-folder of that name in the project's *ensembles* top-level folder. The sub-folder must contain an *ensemble.xml* file that contains all the information necessary for the instance generator engine to instantiate an instance generator and produce new instances.

See fig. 3.2 showing the instance generator engine's dependencies. The engine needs an instance generator implementation to generate new instances in memory and a persistence engine (section 3.5.4) to save the generated instances to disk. A sample *ensemble.xml* file is shown in section A.5. The *ensemble.xml* file is generated and edited by the ScheduleLab editor, described later in the chapter, that obtains all the necessary information from the user. The editor allows the user to select an action to launch the instance generator engine for a specific ensemble descriptor.

The instance generator engine queries the Eclipse plugin registry [4][c.11] to get access to the instance generator plugin extension having the specified *generatorId*. An instance generator extension implements an interface listed in section B.1. The instance generator engine is only interested in invoking the `IInstanceGenerator.createInstance(..)` method repeatedly to generate instances by passing it a `Map` of parameter values from the *ensemble.xml* file. The problem instance returned by the instance generator is assigned a

global unique id (GUID), transformed appropriately (explained below) and saved in the ensemble sub-folder using the persistence engine.

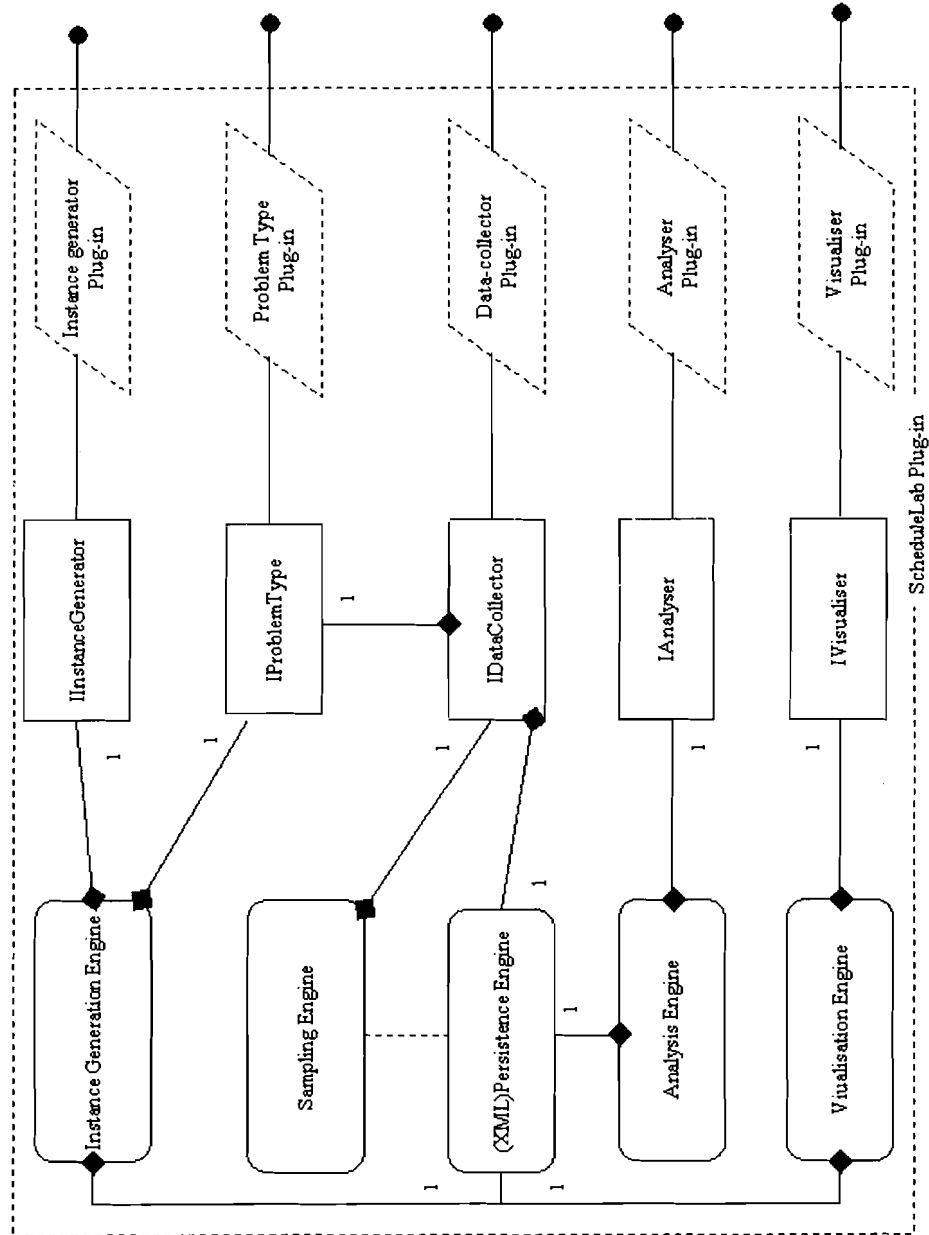


Figure 3.2: ScheduleLab internal structural relationships

Note that the instance generator extension plugin returns its own problem instance type that is external to ScheduleLab effectively providing a level of decoupling. ScheduleLab allows the user's solver code and instance generator plugins to define their own respective problem instance type (a Java object hierarchy representing a problem instance and all its constituents). Thus, a user's solver code solving JSP problem instances can define a problem instance type (`hvastani.msc.jsp.solver1.JSPInstance`) that is different from that used by a random JSP problem instance generator:

(`hvastani.msc.generators.jsp.taillard.Schedule`). The cornerstone for this decoupling to work is a problem instance transformer service offered by the problem-type extension plugins. Problem-type extension plugins are described in later sections, but they essentially define a new problem-type capability (e.g. JSP, VRPTW) for ScheduleLab to support empirical analysis for algorithms of that problem type. The problem-type plugin's transformer service can do bi-directional transformation between an external problem instance type and the problem-type plugin's internal problem instance type, in a transparent way from the perspective of the ScheduleLab core components. It is the problem-type plugin's internal problem instances that are read/written to the user's ScheduleLab project ensemble folders. This decoupling is particularly useful for the algorithm code because it frees it from any dependencies on ScheduleLab. However, the external problem instance type must share the same *structure* as the problem-type plugin's internal problem instance type for the transformation to be feasible. The details of the transformation service are addressed in later sections.

### 3.5.2 Sampling Engine

The sampling engine is responsible for coordinating an experimentation session. An experimentation session involves running one or more algorithms against one or more ensembles of problem instances. The sampling engine requires the following as input:

- the location of the user's ScheduleLab project in the Eclipse workspace,
- the unique name of the experimentation session specified by the user. This must correspond to a sub-folder of that same name in the project's *sessions* top-level folder. The sub-folder must contain *session.xml* and *dataCollector.xml* files that contain all the information necessary for the sampling engine to instantiate and run a data collector, algorithms and problem instances.

As shown in fig. 3.2, the sampling engine uses a `IDataCollector` to run the specified experiment session. The `IDataCollector` in turn needs the persistence engine (section 3.5.4 and the correct implementation of `IProblemType`. The problem type allows the data collector to transparently transform problem instances stored in the `ScheduleLab` project using an internal representation, into a representation expected by the user's algorithm implementation. The data collector extension point is described in detail later in this chapter. A data collector plugin encapsulates the control logic to execute a particular data sampling procedure corresponding to a specific kind of performance analysis of SLS algorithms. For example, a `solution trace` data collector plugin is capable of taking multiple samples of an algorithm's solution trace while solving a single problem instance. A `search space analysis` data collector plugin would be capable of mapping the search space topography corresponding to an algorithm solving one or more problem instances.

A sample `session.xml` file is listed in section A.6. The contents of the `dataCollector.xml` file are specific to a given data collector plugin and the details are described in later sections (e.g., for a `solution trace` data collector, the `dataCollector.xml` contains meta-data about what methods to invoke on the user's solver code to start running the solver, description of the expected problem instance type and the solver's event publishing interface). Both the files are populated with values obtained from the user through the `ScheduleLab` editor. The editor allows the user to select an action to launch the sampling engine to run an experimentation session. The `session.xml` file specifies the Eclipse plugin id of the data collector plugin selected by the user and one or more run-sets. A run-set is a named combination of algorithms, algorithm parameters and instance ensemble (present in the project). The sampling engine queries the Eclipse plugin registry to get access to the data collector plugin matching the specified id. For every run-set specified in `session.xml` the sampling engine invokes the data collector. The data collector runs each run-set algorithm with its parameter values on each of the instances belonging to the instance ensemble. Every data collector plugin implements a Java interface listed in section B.3. The sampling engine must supply the parameters required by the `IDataCollector.start(..)` method. Given the project location the sampling engine determines the associated problem type. The `IRunSet` and `IDataCollectorDescriptor` are object wrappers for a given session run-set and the `dataCollector.xml` file respectively. The data collector obtains the necessary meta-data information for each solver specified in the run-set from the supplied `IDataCollectorDescriptor`.

The output of the data collector is saved in a sub-folder called simply, *data* inside the specified session folder. A further sub-folder is created for output data corresponding to each run-set, named with the run-set id.

### 3.5.3 Analysis and visualisation engines

The analysis and visualisation engines are responsible for coordinating running user specified analysers on the output of a compatible data collector and running user specified visualisers on the output of compatible analysers, respectively. The engines require the following as input:

- the location of the user's ScheduleLab project in the Eclipse workspace,
- the unique name of the experimentation session specified by the user. This must correspond to a sub-folder of that same name in the project's *sessions* top-level folder. The sub-folder must contain *session.xml* file and a *data* sub-folder containing the output of the session's data collector for each session run-set.
- for analysis tasks, the unique name of an analysis task specified by the user. This must correspond to a sub-folder of that same name in the *analysis* sub-folder of the session, that contains an *analyser.xml* descriptor file (section A.7).
- for visualisation tasks, the unique name of an analysis task corresponding to a sub-folder by that name under the *analysis* sub-folder of the session, containing the output of a compatible analysis task for each run-set.

Fig. 3.2 shows that both the analysis and visualisation engines need access to a persistence engine in order to read the compatible input data for processing. The analysis engine inputs the data into an implementation of *IAnalyser* and saves the output data to disk using the persistence engine. The visualisation engine inputs the data into an implementation of *IVisualiser* that in turn may for example save graph image files in the project. A ScheduleLab analyser extension plugin declares the data-collectors it is compatible with, that is, the data-collectors whose output it can analyse. For example, a *SQD* and *QRTD* analyser is only compatible with a *solution trace* data-collector. Similarly, a ScheduleLab visualiser

plugin declares the analysers it is compatible with.

The ScheduleLab editor allows the user to select and launch an analyser that is compatible with the experimentation session's data collector. Similarly, the user can launch a visualiser that is compatible with the output of a previously run analyser. When the user launches an analyser the result is a sub-folder in the *analysis* folder of the session, with a user specified name, that contains a descriptor file called *analyser.xml* and the output of the analysis task. A sample descriptor file is show in section A.7.

#### 3.5.4 Persistence engine

The ScheduleLab plugin generally saves data in Extensible Markup Language (XML) format. Java objects like problem instances, solution trace data and analysis output data need to be stored and retrieved from the user's ScheduleLab project, in XML format. The actual XML schema that these files are represented in is irrelevant. On the other hand, ScheduleLab related descriptor files (*sLabProject.xml*, *ensemble.xml*, *session.xml*, etc.) are XML files with a predefined structure, that may be directly edited by the user and we need to programmatically read the XML element and attribute values throughout ScheduleLab. ScheduleLab has a XML persistence engine that it exposes to other core components and any dependent plugins as shown in fig. 3.2. The persistence engine provides services such as transparently storing and retrieving Java objects from XML files, as well as reading and writing descriptor files. Behind the scenes the persistence engine relies on two third-party libraries:

- XStream: Given a Java class that has been annotated with XStream annotations, XStream can transparently and efficiently save and retrieve Java objects from XML files. The XML files are saved in an XStream specific schema and are in a sense internal to ScheduleLab. In later sections we see how ScheduleLab plugins use XStream annotations to save and retrieve their Java objects.

(<http://xstream.codehaus.org/>)

- Jakarta Commons Configuration: Given an XML configuration file that can be edited in a text editor, Commons Configuration allows us to programmatically read and edit XML element and attribute values.

(<http://commons.apache.org/configuration/>)



## 3.6 ScheduleLab extension points

### 3.6.1 Problem Type extension point

Problem type plugins allow ScheduleLab to support new resource scheduling problem instances. Problem type plugins act as a bridge between ScheduleLab's core components, the user's algorithm code and problem instance generators. As shown in figure 3.2, a problem type plugin contributes an implementation of `IProblemType` that in turn is used by the sampling engine (passed as input to a data-collector) as well as the instance generator engine. ScheduleLab allows user's algorithm code and problem instance generators to expect their own problem instance object types. But it is the problem type plugin that provides the internal translation service.

The problem type plugin has the following responsibilities:

- It specifies the expected structure of the problem instances for this type of resource scheduling problem.
- It provides a service to transform between an arbitrary target problem instance type object that complies with the expected instance structure into the plugin's internal problem instance representation.

The decoupling between ScheduleLab, instance generators and user algorithm code is desirable because it minimizes the amount of compile-time coupling and effort needed for a user to use ScheduleLab. The user's algorithm implementation may use its own object hierarchy representing a problem instance. Also the user's problem instance may need to hold additional application specific data that may be irrelevant to ScheduleLab. Discarding application specific Java objects for classes that are compatible with ScheduleLab is tedious and undesirable.

In order for this decoupling to work, it must be feasible for the problem type plugin to transform between external and internal problem instance representations. If two arbitrary objects are structurally the same, differing only in naming or in concrete types that have the same superclass (`java.util.ArrayList` and `java.lang.Set` are both `java.util.Collection`), then it would be feasible to transform between them. For example, in the case of JSP problem instances, we can say that the problem instance should have an array or a collection of resource objects and job objects. Each of these can define

additional properties and further associations with other domain object types. If two JSP problem instance types comply with the above structure, they can be transformed into each other.

The basis for this mechanism is the Java Reflection API. Through the Reflection API, the Java platform provides a programmer the ability to introspect and instantiate any arbitrary Java class. Introspection includes listing the fields and methods of a class as well as being able to invoke `public` visibility methods on the object, with the appropriate parameters. The expected XML meta-data (empty values) is specified by the problem type plugin and the meta-data values are specified by the user or instance generator plugin developer, once only. See section A.2 for an example of a meta-data file for a JSP problem-type plugin. The meta-data file must be specified by the user or a problem instance generator plugin developer. The meta-data file captures all the necessary information that the problem type plugin will need to transform between the target problem instance to the plugin's internal problem instance representation. The meta-data file also describes the expected structure of the target problem instance. In the given example, the target problem instance is expected to be an object that has a field that provides a (string) description of the instance. The target instance must have a collection or an array of objects representing resources. Each resource must have a field for a (long integer) id and a field for the name of the resource. Similarly, the target instance must have a collection or an array of objects representing JSP jobs, and each job must have a duration field, a resource assignment field and a start time field. The values of the XML elements of the meta-data file correspond to actual field names or fully-qualified Java class names that the problem type plugin uses with the Java Reflection API to provide a bi-directional problem instance transformation service between external and internal problem instance representations.

The problem type plugin developer is expected to implement the interface listed in section B.2. ScheduleLab's core components register the external instance meta-data with the problem type before invoking methods to do transformations. Finally, the problem type plugin developer is expected to publish supplementary documentation explaining the expected structure from the target problem instance, in addition to an empty XML meta-data file.

### 3.6.2 Problem Instance generator extension point

Instance generators are contributed to ScheduleLab as Eclipse plugins that are extensions of ScheduleLab's problem instance generator extension point. Figure 3.2 shows that an instance generator plugin contributes one or more implementations of `IInstanceGenerator` that are used by the instance generator engine to generate new instances in memory. Instance generators are specific to a ScheduleLab problem type that they must declare. They implement an interface given in section B.1. The key method is `IInstanceGenerator.createInstance(...)` that accepts a map of parameters needed by the generator and returns a new instance object. The generator plugin must describe its parameters in a file named *generator.xml* in its directory. See section A.9 for a sample listing of *generator.xml*. This file will describe each parameter that the generator accepts. For each parameter the following is required:

- name
- type of the parameter: int, long, double, string
- validation rule: expressed as a java boolean expression that can refer to other parameters by name. e.g.  $((minDuration > 0) \text{and} (maxDuration > 0))$

Internally, ScheduleLab uses an expression evaluation library called Jakarta Commons Jexl to evaluate validation expressions (with parameter references) [1]. ScheduleLab accepts and validates the parameter inputs from the user and when the user gives the command to generate instances, it invokes the `createInstance(...)` method of the plugin with a `Map` of the user specified parameters. The implementation of the `createInstance(...)` method would typically create an empty problem instance object and populate it according to a specific algorithm. For example, a plugin implementing Taillard's [17] random JSP generating algorithm would create the user specified jobs and machines and for each job operation, sample a random uniform distribution to assign durations.

An instance generator must specify the meta-data for its own instance object hierarchy so that ScheduleLab can transform instances generated by the generator to an internal format for disk storage. ScheduleLab will transform the instance generator's instance object into the internal problem instance type of the corresponding problem type plugin, before

saving the generated instance using the persistence engine.

### 3.6.3 Data collector extension point

The data collector extension point represents the heart of ScheduleLab's experimentation capabilities and hence is the most complex plugin type to implement. It implements the `IDataCollector` interface listed in sec B.3. The data collector is meant to encapsulate the control logic to initialise solvers, pass them problem instances as input, perform sample runs and extract the appropriate data necessary for a particular type of experimental analysis. Run-time and solution quality performance analysis [11][c.4] and search space topography analysis [11][c.5] are two kinds of experimental analysis for SLS algorithms. As an example, we discuss the former. Figure 3.2 shows that a data collector plug-in contributes an implementation of `IDataCollector` that in turn is used by the sampling engine to execute an experiment session.

A data collector for run-time and solution quality performance accepts a `IDataCollectorDescriptor` object that wraps a descriptor file like in section A.10. This descriptor is specified by the user through the ScheduleLab Editor. The descriptor provides the data collector plugin with the information necessary to instantiate and integrate at run-time with the run-set solvers. Using Java Reflection API the plugin can instantiate the solver and know how to query methods on the solver. But more importantly the data collector plugin must register appropriate listener objects with each solver at run-time to be notified of solution trace events. This is achieved using *dynamic proxies* available as part of the Java platform. Dynamic proxies allow us to create a proxy object implementing any interfaces, such that we can assign it a handler object and then any method calls on the proxy will be delegated to the handler object. Thus, we can create dynamic proxies for the target solver code's event handler interface specified in the *dataCollector.xml* file and register those as event-handlers with the solver. A pseudo-code example to illustrate this is shown in program 3.1.

When the data collector is notified of a solution improvement event, it invokes methods on the solver to collect the sample point data: time elapsed, operations count and metric value (solution quality). The sample points are collected in memory. Once all the samples

```

//we'll create a dynamic proxy for SolverEventHandler
//define our proxy handler
InvocationHandler handler = new InvocationHandler(){
    //any method call on the proxy gets delegated to this method
    public Object invoke(Object proxy, Method method, Object[] args){
        //depending on method name, do something
        //if method name is "notifyImprovement" then get trace data
        //if method name is "notifyStop" then sample run has ended.
    }
};

//now create proxy and register with instantiated solver object
Class lstnrClass = Class.forName(..);
Object l = Proxy.newProxyInstance(lstnrClass.getClassLoader(),
    new Class[] { lstnrClass },
    handler);

//register our dynamic proxy with the solver using reflection api
...

```

Program 3.1: Pseudo-code showing usage of Java *dynamic proxies* to register event handlers with solvers

are obtained, the data is saved to disk using the persistence engine (see section C.2).

### 3.6.4 Data analysers and visualisers

As shown in figure 3.2 a data analyser plug-in must provide an implementation of the ScheduleLab interface `IDataAnalyser` listed in section B.4. A data analyser plugin declares which data collectors it is compatible with by giving an array of the unique plugin-ids of the compatible data collectors. The ScheduleLab editor only displays those data analysers to the user that are compatible with the active experiment session's data collector. The user must specify any parameters implemented in the same way as that for problem instance generators. When the data analyser is launched it must save the output of the analysis in XML format using the supplied persistence engine object, in the folder specified by the Data analysis engine.

The QRTD, SQD data analyser plugin can extract *QRTDs* and *SQDs* from the solution trace data produced by the solution trace data collector.

A data visualiser plugin must implement the ScheduleLab interface `IDataVisualiser` listed in section B.5. A data visualiser declares which data analysers it is compatible with through the implemented interface's methods. The ScheduleLab editor allows the user to select a visualiser that is compatible with the analysers used in the active experiment session. The QRTD, SQD data visualiser plugin generates combined QRTDs and SQDs graph files for the session's algorithms, corresponding to each run-set.

```
-SampleProject
  |_sLabProject.xml
  |_ensembles
  |   |_taillard_10x50
  |     |   |_ensemble.xml
  |     |   |_1.xml
  |     |   |_2.xml ...
  |     |_watson_machCorr_10x50
  |       |_ensemble.xml ...
  |_sessions
  |   |_cworks_jsp
  |     |_session.xml
  |     |_dataCollector.xml
  |     |_data
  |     | |_taillard_10x50
  |     | | |_1.xml ...
  |     | | |_watson_machCorr_10x50
  |     | | |_1.xml
  |     | | |_2.xml ...
  |     |_analysis
  |       |_sqd_qrtd
  |         |_analyser.xml
  |         |_taillard_10x50
  |         | |_1
  |         | | |_qrtd
  |         | | | |_solver1_1_10%.xml ...
  |         | | | |_solver1_1_200%.xml
  |         | | | |_sqd
  |         | | | | |_solver1_1_5s.xml ...
  |         | | | | |_solver1_1_180s.xml
  |         | | | |_2 ...
  |         | |_watson_machCorr_10x50
  |         | | |_1 ...
```

Figure 3.3: Sample ScheduleLab project structure

Figure 3.3 shows a skeletal view of the ScheduleLab project structure after an analysis task. The output of a visualiser is specific to a plugin and will be described in later sections. Usually a visualiser plugin contributes new Eclipse views or editors to graphically display the analysis output. At the top level the project has a *sLabProject.xml* descriptor file, an *ensembles* folder containing generated problem instance ensembles (one sub-folder for each ensemble) and a *sessions* folder containing experiment session related files. Each ensemble sub-folder must contain an *ensemble.xml* descriptor file and may contain the generated instance files stored in XML format. The *sessions* folder contains a sub-folder for every experiment session created in the project. A session sub-folder must contain *session.xml* and *dataCollector.xml* descriptor files. If the experiment session has been run then the session sub-folder must contain a *data* folder containing sample raw-data corresponding to each ensemble instance of a run-set. If any analysers have been run then the sessions folder must contain an *analysis* folder containing a sub-folder for each analysis task performed.

### 3.7 ScheduleLab User Interface

In this section we present the ScheduleLab user interface. As mentioned earlier, the ScheduleLab plug-in contributes a new-project wizard and a custom editor to the Eclipse workbench UI. Thus, when the user selects the menu `New > Project...`, he is offered a new project category for creating a ScheduleLab project. This is shown in the screen-shot in figures 3.4 and 3.5. When the user selects to create a new ScheduleLab project the ScheduleLab *new project wizard* is instantiated by Eclipse and displayed to the user, as shown in figure 3.6.

Once the user specifies all the fields in the wizard, the wizard's control logic creates a new ScheduleLab project and opens the ScheduleLab editor for this newly created project, as shown in figure 3.7. The ScheduleLab editor is declaratively associated to be the editor for the following file types: *sLabProject.xml*, *ensemble.xml*, *session.xml*, *dataCollector.xml*, *analyser.xml*, *visualiser.xml*. Thus, when the user double-clicks on any of these files in a ScheduleLab project, the editor is opened. The ScheduleLab editor is a multi-page editor in that it has multiple pages or tabs as seen in 3.7. The editor is a single editor that transparently updates all the meta-data files in the ScheduleLab project. But the role of

the ScheduleLab editor is to effectively support an experimentation workflow of generating problem instances, setting up experiment sessions, running experiment sessions to collect raw-data, analysing and visualising raw data.

Figure 3.8 shows the *Ensembles* page of the ScheduleLab editor. The user can click on the *Add* button to bring up a wizard to create a new problem instance ensemble. Once the user completes the wizard, the wizard's control logic creates a new *ensemble.xml* file in the ScheduleLab project and updates the editor's view. Figure 3.9 shows two things: the ensembles master-detail view (in the background), and the user having launched the instance generator engine for the ensemble *taillard\_10x50*. The user can edit the parameter values and save the editor to update the underlying *ensemble.xml* file. The instance generator engine is launched by selecting the ensemble in the master view and clicking the *Launch* button. This opens a progress dialog that gives a status of the instance generator engine's progress. Also, the user can choose to run this job in the background by clicking on the *Run in Background* button, so as to continue doing other tasks in the Eclipse workbench.

Figure 3.10 shows the result of the previous step in the *Package Explorer* view on the left with newly generated problem instances in the *ensemble* folder. The figure also shows the *Experiment Sessions* editor page master-detail view. The master view is a tree with all the experiment sessions of this project as top-level nodes. Each session node has exactly one *Data collector* node and one *Run-sets* node. The user can add exactly one data-collector by selecting the *Data collector* node clicking on *Add* to bring up a wizard that asks the user to select an data collector implementation. The screen-shot shows that the user has selected a *solution trace* data collector. Similarly, one or more run-sets can be added under the *Run-sets* node. Clicking on any top-level node or leaf-node activates the corresponding details page in the details section on the right. The screen-shot shows the details view of the selected run-set. Figure 3.11 shows the user launching the sampling engine to run the selected experiment session. This is done by selected a top-level session node and clicking on the *Launch* button. This action opens a progress dialog similar to that for the *Ensembles* editor page.

Figure 3.12 shows the master-detail view on the *Session Data Analysis* editor page in the background with the user having launched the selected analysis task. The user can add



new analysis tasks by clicking on the *Add* button and edit the parameters in the details view. Finally, figure 3.13 shows the master-detail view of the *Session Data Visualisation* editor page with similar interaction features.

### 3.8 Summary

In this chapter we described the architecture and design of ScheduleLab implemented as an extensible Eclipse plug-in. We described the working of the core components of ScheduleLab, the primary extension points and the ScheduleLab user interface within the Eclipse IDE context. ScheduleLab is an effective, extensible tool for empirical analysis of resource scheduling SLS algorithms that facilitates re-usability.

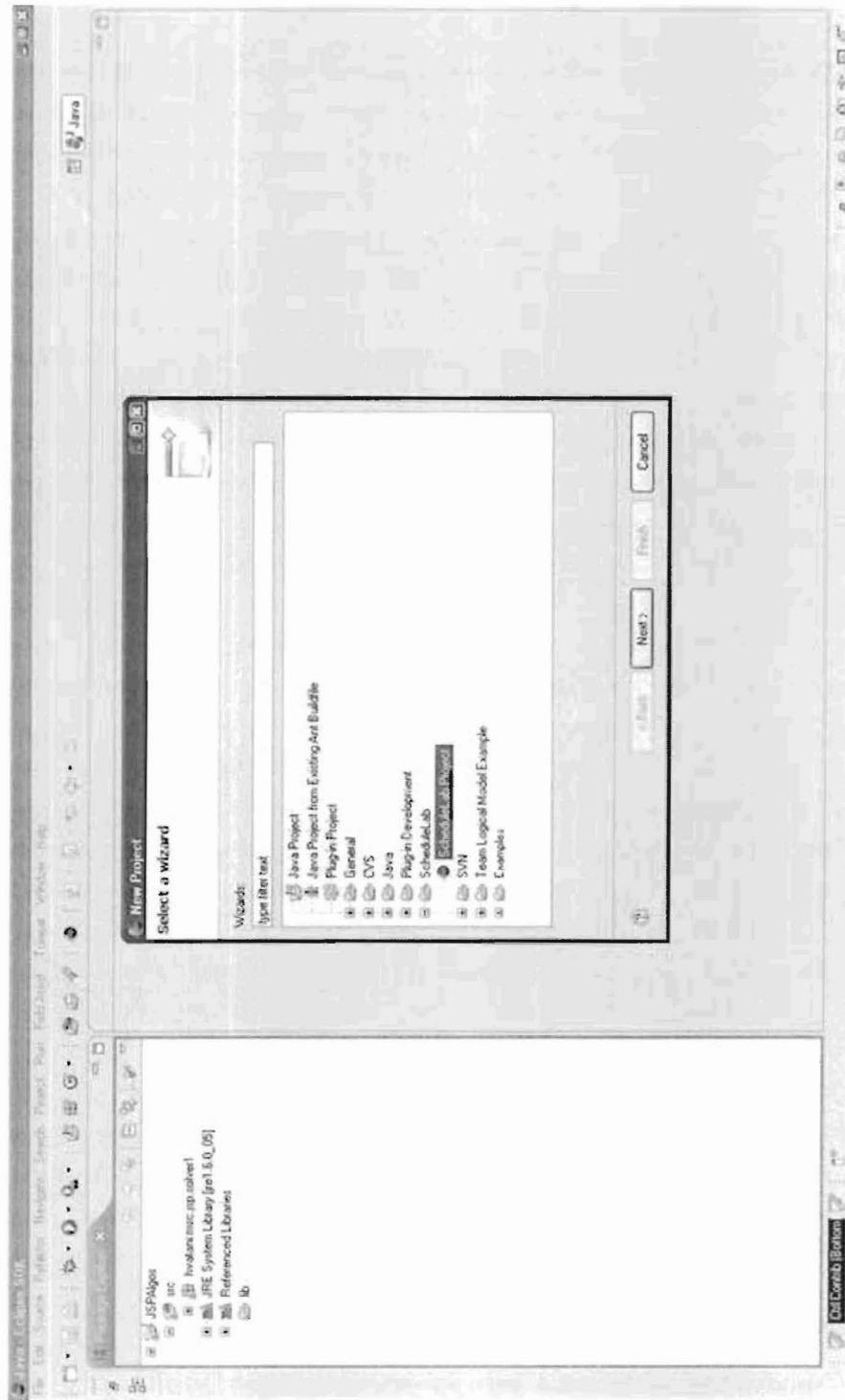


Figure 3.4: New Projects wizard showing ScheduleLab Project wizard entry

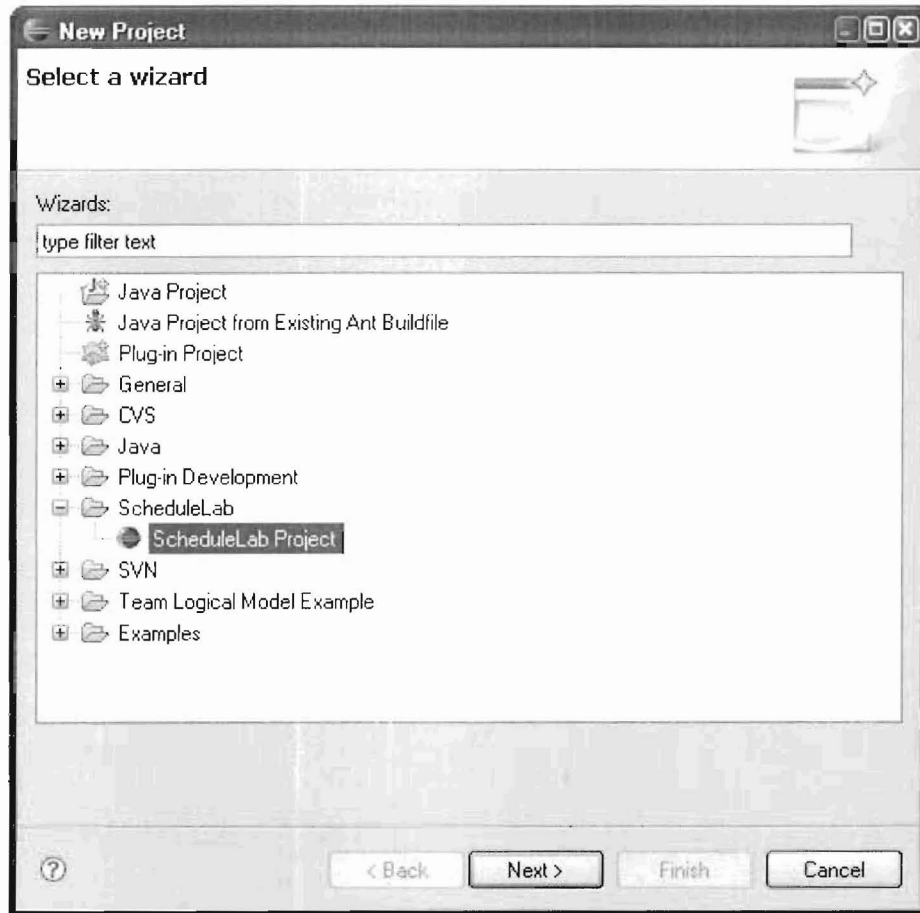


Figure 3.5: New Projects wizard showing ScheduleLab Project wizard entry

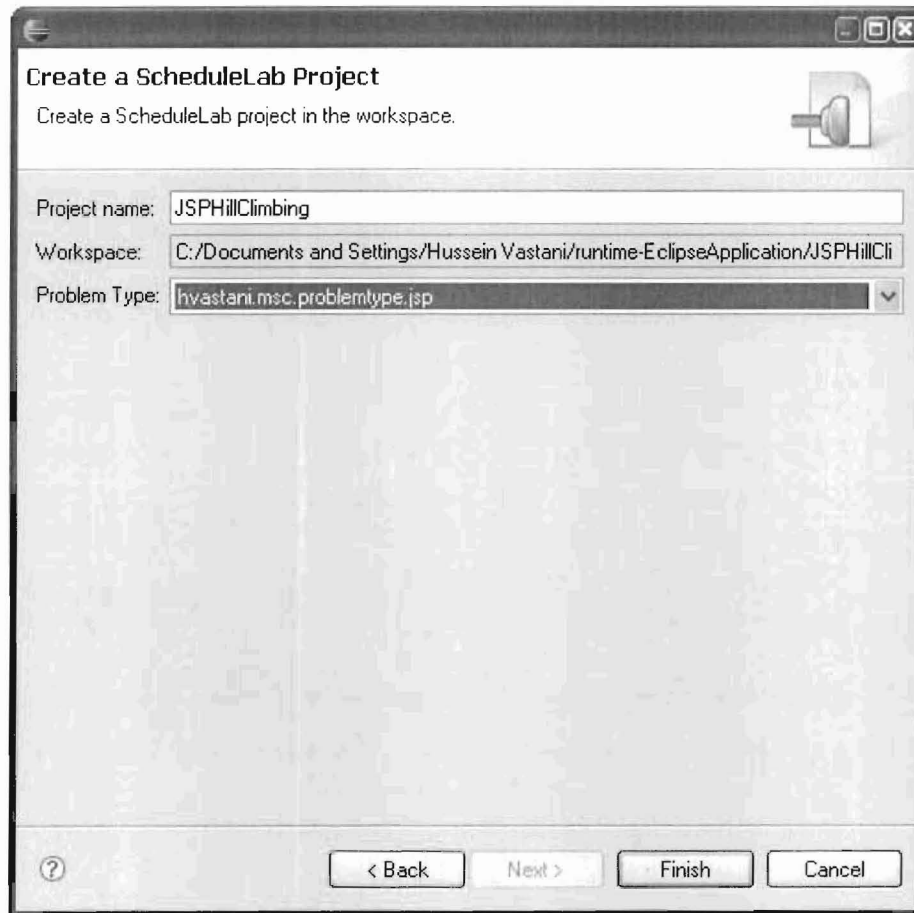


Figure 3.6: New ScheduleLab Project wizard

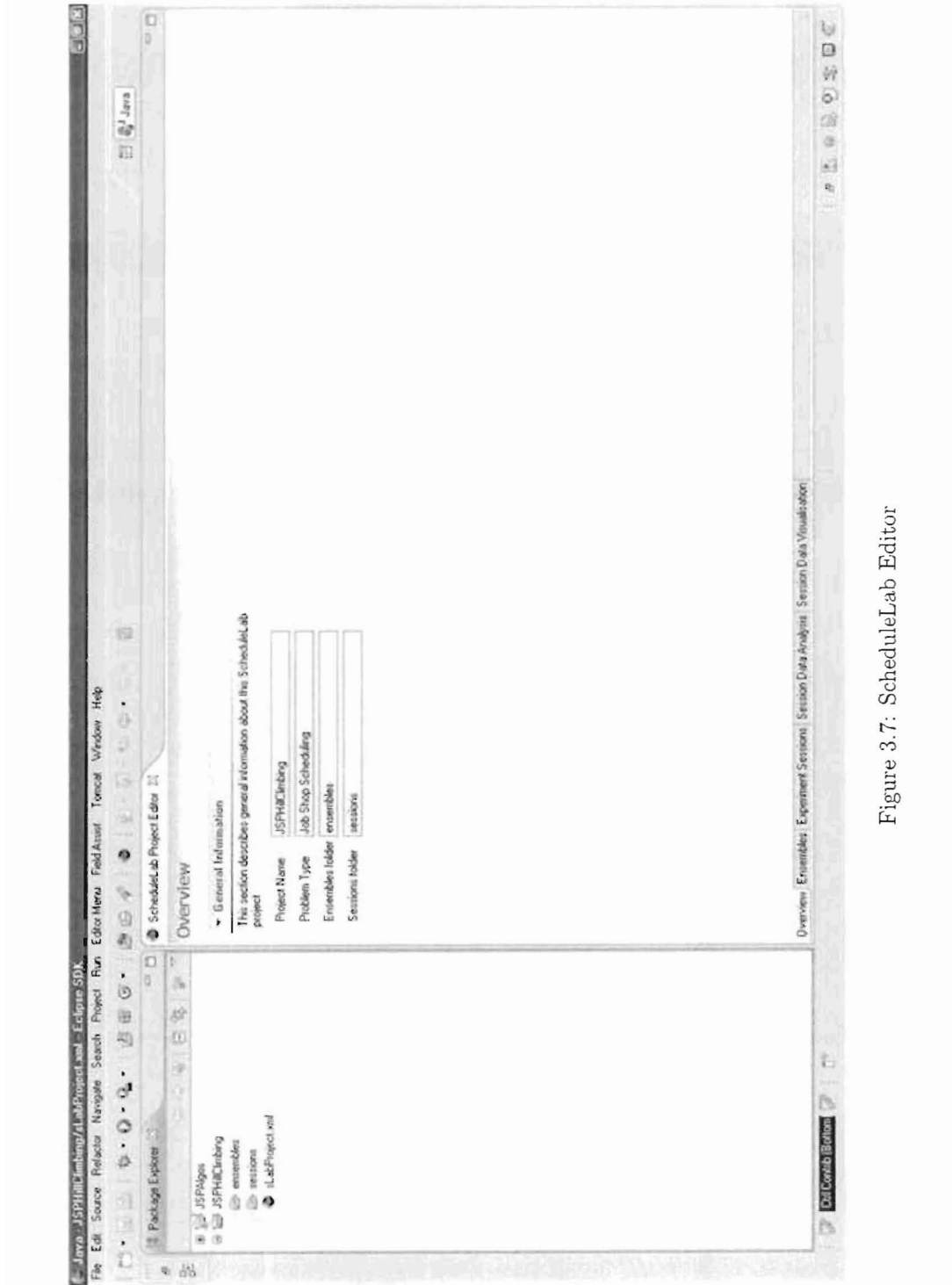


Figure 3.7: ScheduleLab Editor

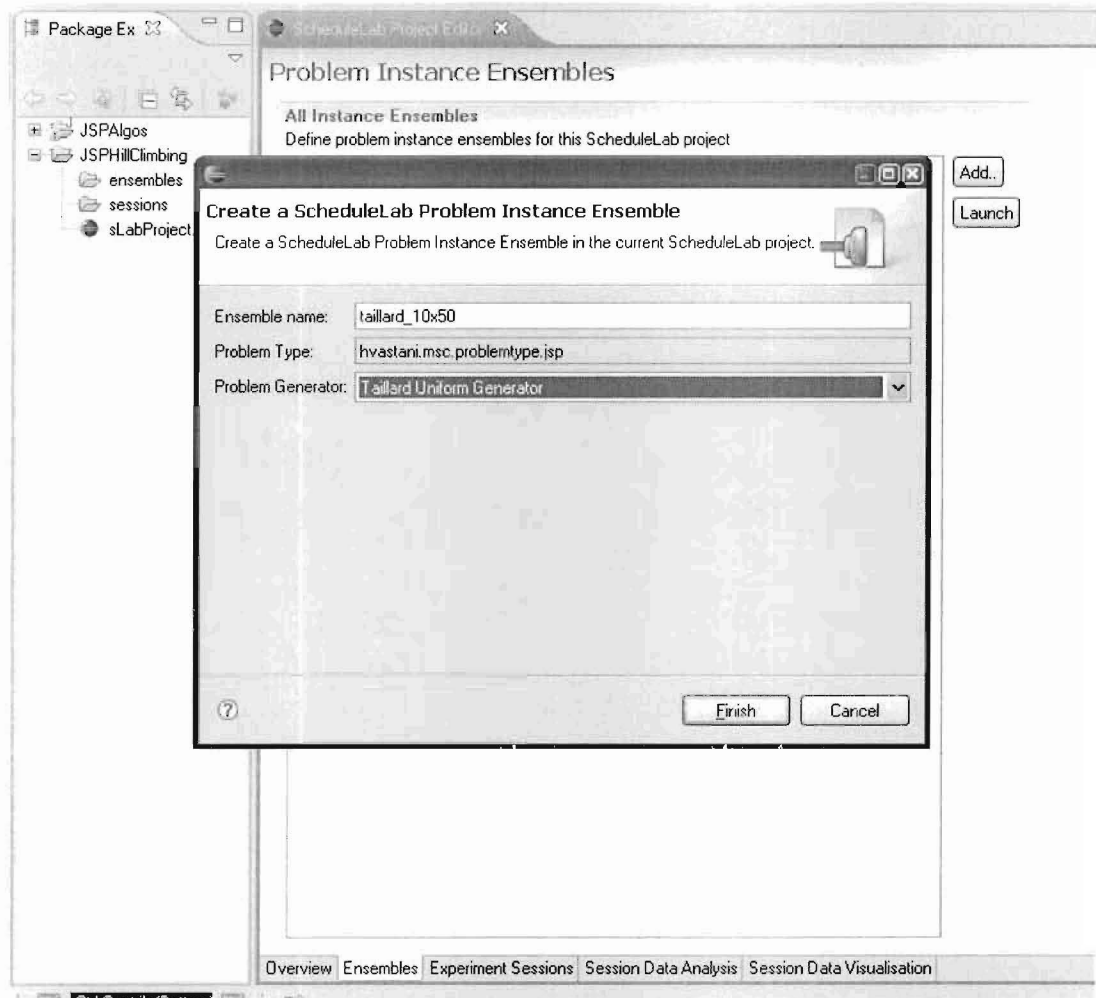


Figure 3.8: New ScheduleLab Problem Instance Ensemble wizard

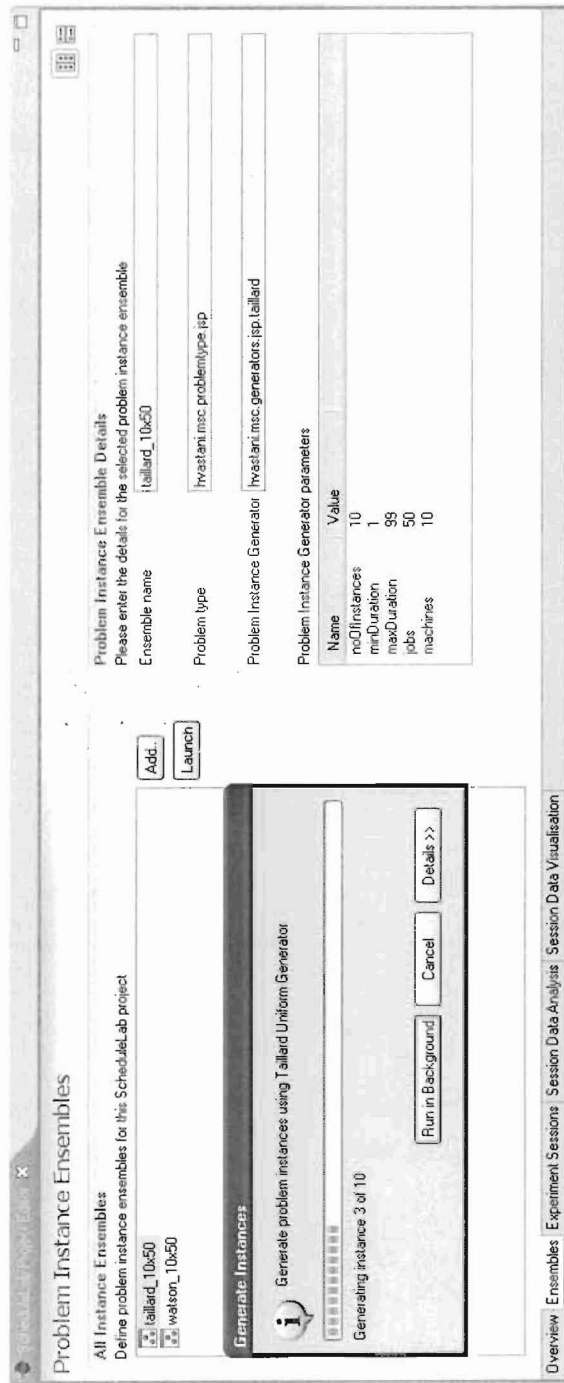


Figure 3.9: Instance Ensembles Master Detail view

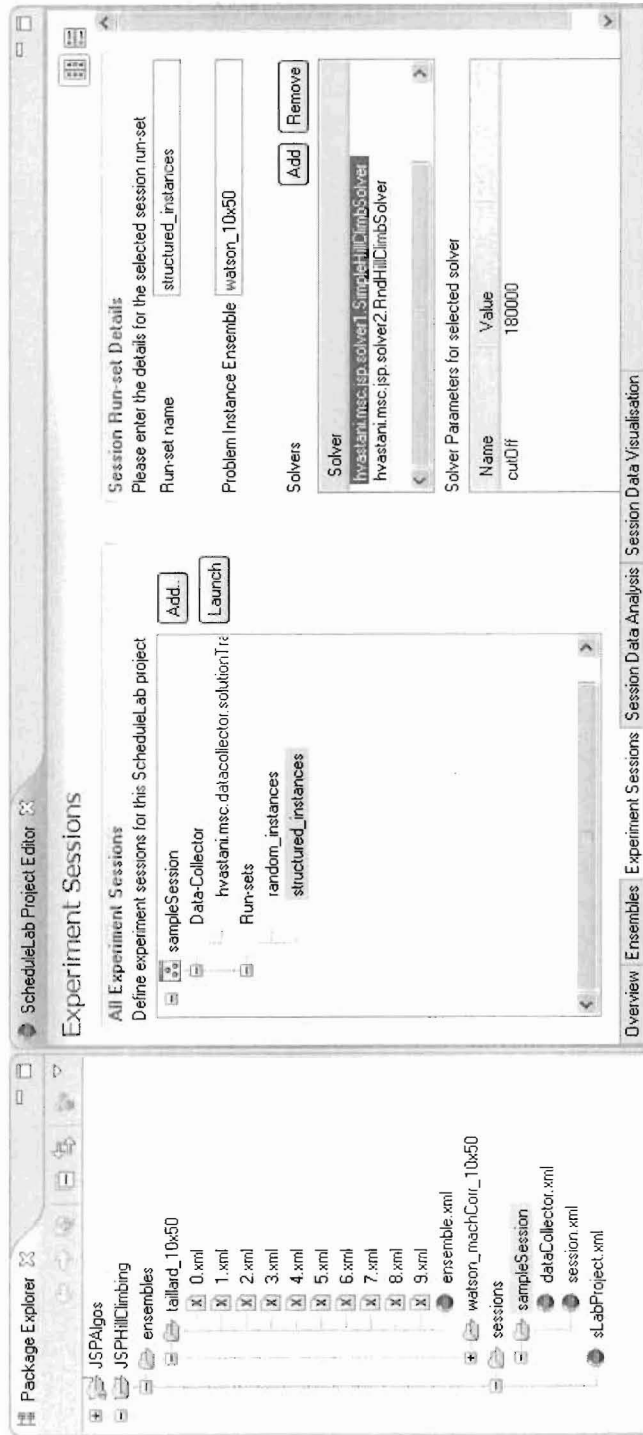


Figure 3.10: Experiment Sessions Master Detail view



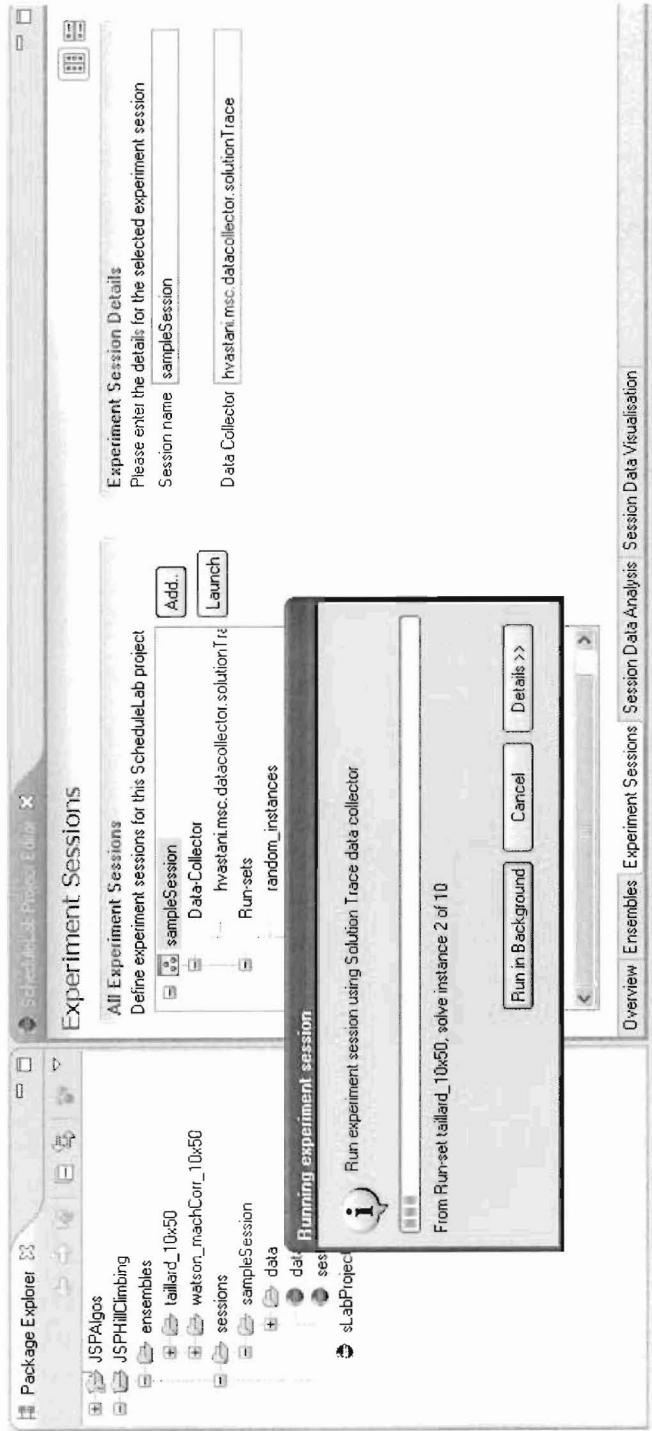


Figure 3.11: Running Experiment session run-sets

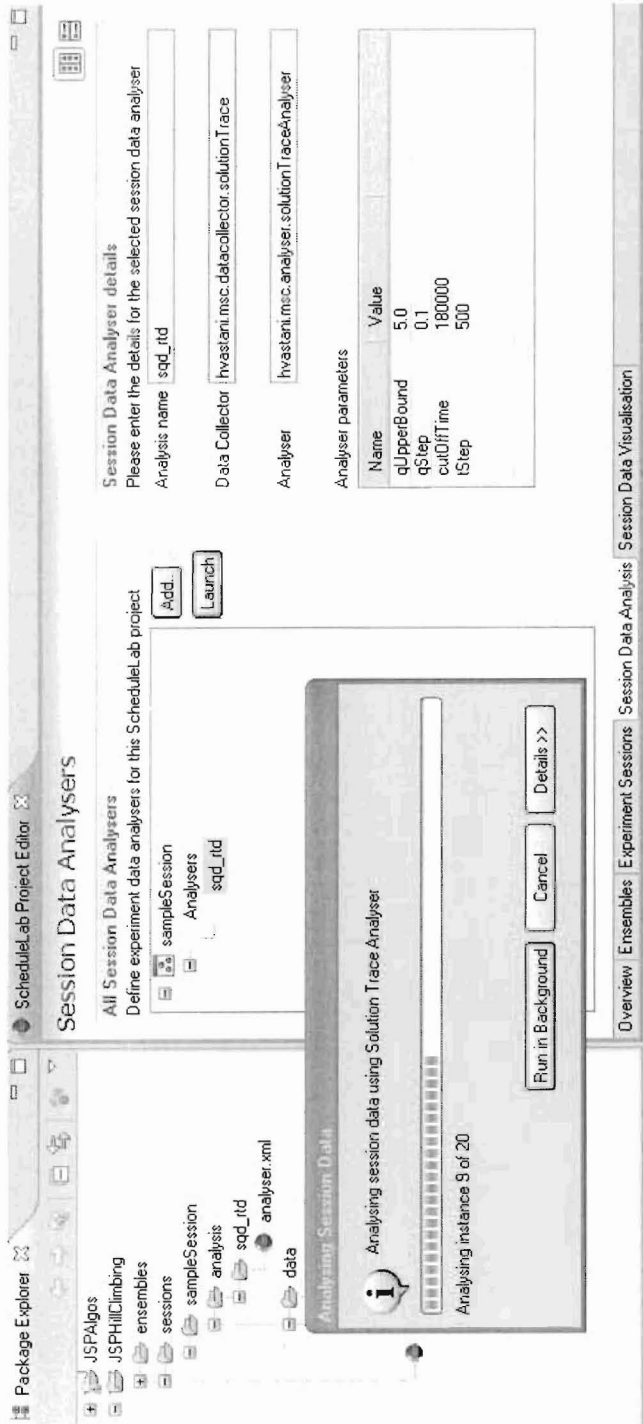


Figure 3.12: Running Analyser on experiment session data

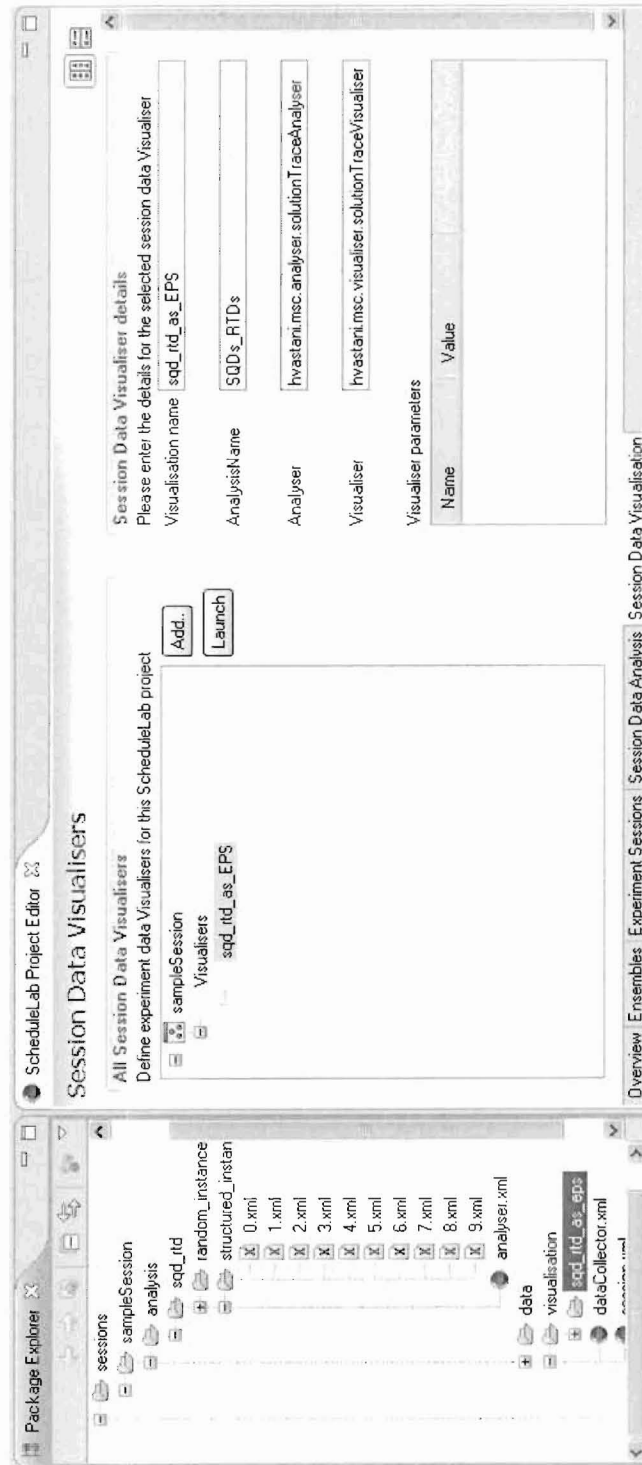


Figure 3.13: Session Analysis data Visualiser master-detail view

## Chapter 4

# Operation & Evaluation

ScheduleLab is a proof-of-concept tool based on the Eclipse platform [3] that aims to provide comprehensive support for doing empirical analysis of SLS algorithms solving resource scheduling problems. ScheduleLab allows users to extend and contribute functionality to the tool while facilitating experimentation on algorithm implementations in a loosely coupled manner. There is a need for increased research effort in tool support for empirical analysis of SLS algorithms and we see ScheduleLab as a contribution in that direction. ScheduleLab is not a standalone software library for experimentation on SLS algorithms like EasyAnalyzer [5], that must be run from the command-line. Instead ScheduleLab integrates with a feature-rich integrated development environment (IDE) with the aim of enhancing the tool's usability. We also believe that software developers increasingly expect that new development tools and products integrate with their favourite IDE that they are familiar with instead of expecting the developer to learn to use a new piece of software in conjunction with their IDE.

ScheduleLab aims to make the experimentation process efficient for researchers by minimizing duplication of effort through the re-use of plugins contributed by other users. ScheduleLab is an extensible Eclipse IDE plugin targeted at researchers programming in Java (note that it is possible to integrate with C/C++ code from Java). So users who want to contribute new functionality to ScheduleLab are expected to know how to use the Eclipse Plugin Development Environment (PDE) [4, 6, 3] to write Eclipse extension plugins for ScheduleLab. However, deep knowledge of the working of the Eclipse platform is not required. Eclipse plugin extension points are essentially Java interfaces and Eclipse plugins that extend an extension point, basically implement interfaces represented by the extension

point. Thus, ScheduleLab contributors would have to contribute one or more classes implementing one or more ScheduleLab interfaces (see chapter 3). Eclipse in turn manages the lifecycle of the plugin, namely its initialisation, registration in a central plugin registry and finally disposal of the plugin. ScheduleLab's core components delegate control to plugins that extend ScheduleLab's extension points. These plugins are discovered through the Eclipse plugin registry.

In this chapter, we present an operational evaluation of ScheduleLab by using ScheduleLab to do preliminary empirical analysis of two SLS algorithms solving Job Shop Scheduling problems. The intention here is not to present a case study of effective empirical research, but to illustrate that ScheduleLab can be used to effectively pursue that endeavor. As part of this exercise we wrote plugins to add new functionality to ScheduleLab, described in later sections, in order to conduct our experiments. Also, we only discuss the effort involved by the developer to achieve his goals in terms of programming or configuration work involved. To see a discussion of the ScheduleLab user interface, please refer to section 3.7.

## 4.1 Setup & Methodology

In this section we describe that we started with the choice of a resource scheduling problem domain and two candidate algorithms implementations and we went through the necessary steps to use ScheduleLab to do preliminary empirical analysis. We discuss the effort involved in using ScheduleLab for this exercise. The experiment was run on a machine having a 1.2GHz Xeon processor with 1.7GB RAM on an Ubuntu Linux operating system with Sun Java 6.

### 4.1.1 Job Shop Scheduling Problem

The JSP is characterized by  $m$  machines or resources and  $n$  jobs. Each job is executed exactly once on each machine in a pre-determined way. The execution of a job  $i$  ( $1 \leq i \leq n$ ) on a machine  $j$  ( $1 \leq j \leq m$ ) is called an operation or task. Thus each job  $i$  is comprised of  $m$  operations and the pre-determined permutation of machines  $\pi$  that each job is routed through implies precedence constraints between the operations of that job. Operation  $o_{ij}$  cannot initiate processing until  $o_{i,j-1}$  has completed. For our study, the JSP

objective function is to minimize the makespan, that is, the maximum job completion time of the entire schedule. Please refer to [19, 11] for a thorough treatment of JSP and related scheduling problems.

We selected the JSP for our study because of its familiarity. The JSP is one of the most studied resource scheduling problems in the literature. At the same time this prototypical scheduling problem is NP-Hard and continues to pose a constant challenge to researchers [12].

### 4.1.2 Algorithms

We obtained two functional, yet preliminary algorithm implementations that can solve JSPs. The two algorithms were supplied by researchers at Actenum Corporation, Vancouver, Canada ([www.actenum.com](http://www.actenum.com)) and are based on Actenum's ScheduleWorks scheduling library. For our purposes we treat each algorithm as a black box, with the following brief descriptions:

- **Simple Hill-Climbing:** A simple hill-climbing algorithm that does a first-improving local search on a neighbourhood that re-inserts a single activity at a time, using a slack heuristic. The activities are ordered by the slack heuristic. This heuristic looks at the earliest and latest possible start times for each activity, and gives a higher heuristic value to activities with the least slack. For example, it assigns the largest heuristic value to any activity that is on the critical path. The neighbourhood systematically iterates through the insertion points of the candidate activity.
- **Random Hill-climbing:** A hill-climbing algorithm that does a first-improving local search on a random single activity re-insertion neighbourhood. Each local move is a random walk to find the first improving solution. The neighbourhood chooses an activity and an insertion point randomly.

Both algorithms are first-improvement hill-climbing algorithms and so we can expect them to be prone to get stuck in local minima sooner or later. However the random hill-climbing algorithm probably has a better chance of avoiding local minima because it randomly chooses both the activity as well as its re-insertion point.

Each algorithm is implemented as a component that uses ScheduleWorks behind the scenes. The algorithm component is the entry point to the solving functionality and it

accepts its own problem instance type as input. Further, each algorithm component can register solver event listeners that can receive solver events. This interface is described in later sections.

### 4.1.3 JSP Problem Type

In order to do experimentation on JSP problems we need to ensure that ScheduleLab has a JSP problem type capability. Without a JSP problem type, ScheduleLab would not let us create a ScheduleLab project. For a user, this means writing a JSP problem type extension plugin or installing one written by some other user.

For this evaluation a new ScheduleLab problem type was implemented to support JSP algorithms and instances. This would be a one-time effort to implement in the form of an Eclipse plugin extension that can be published for other researchers to download into their respective Eclipse installation [6].

Recall from Section 3.6.1 that implementing a new problem type extension plugin entails:

- Implementing an *internal* problem instance object hierarchy that will be used for saving JSP problem instances to disk as XML files.
- Specifying the structure of a JSP problem instance (in XML) that problem generators as well as algorithm target code must comply with.
- Implementing a bi-directional transformation service that will take an arbitrary external problem instance object representing a JSP instance that complies with a structure prescribed by our problem type plugin.

Section A.1 lists the problem instance structure meta-data XML file that we publish as part of our problem type extension plugin. Note that the tags have no values. Our problem-type extension plugin publishes this in its documentation. Problem generator contributors and users are expected to use this empty meta-data file as a template and fill in the appropriate values. The hierarchy of the XML tags in the meta-data indicates the expected structure of the problem instance. Thus, an *instance* has a *description* and a *type* as fields. The *type* is expected to be a fully qualified Java class name. Further, the *instance* must have *resources* and *jobs* as child lists (Java collection or array). Note that all the

XML element value types are character or string. The values essentially represent the name of a field that can be accessed via public `getters/setters`. A descriptor file specified by a JSP problem instance generator for our JSP problem-type is listed in program 4.1:

Our problem type's internal problem instance object looks like the listing in program 4.2. The Java annotations are XStream annotations required by the ScheduleLab persistence engine to transparently save/retrieve our JSP problem instance. These annotations are not required for our problem type plugin to function. They are included here in order to customize how XStream represents our problem instance object in XML. Here the `@XStreamImplicit` annotation tells XStream not to surround the `machine` XML elements with an `arrayList` parent element, but to simply list out `machine` elements representing each JSP machine object, contiguously in the XML file without any surrounding parent XML element.

Finally, we implemented the bi-directional object transformer service in our JSP problem type, using an open-source framework called Morph [15] that provides a scaffolding for the task of transforming between objects and requires the library user to implement any custom fine grained object-to-object transformers if necessary that Morph then delegates to, while doing coarse grained object-to-object transformations. See section B.2 for the listing of the Java interfaces of ScheduleLab's problem-type. Note that Morph provides a higher layer of abstraction above the Java Reflection API [13] that minimizes the amount of code we need to write to do object transformations. The choice of using third-party libraries like Morph is internal to the plugin development. ScheduleLab does not impose such choices.

#### 4.1.4 Problem Instance Generation

We decided to do preliminary analysis of our two algorithms by running them against two ensembles – one of unstructured (random) and the other of structured instances. In all we generated 20 instances, 10 instances in each ensemble. Each of the instances has 10 machines and 50 jobs and the activity durations are in the interval [1,99].

Since this is a first time use, we have to implement the problem instance generators ourselves. In order to generate the unstructured instances we wrote an instance generator in ScheduleLab that implements the technique from Taillard's ORLIB [17]. Basically the duration of each activity is sampled from a uniform distribution over [1,99] and then each



```

<?xml version="1.0" encoding="UTF-8"?>
<instance-defn>
  <instance>
    <type>hvastani.msc.jsp.randomNbrHood.Schedule</type>
    <desc>description</desc>
    <resources>
      <name>machines</name>
      <resource>
        <type>hvastani.msc.jsp.randomNbrHood.Resource</type>
        <id>id</id>
        <name>name</name>
      </resource>
    </resources>
    <jobs>
      <name>jobs</name>
      <job>
        <type>hvastani.msc.jsp.randomNbrHood.Job</type>
        <id>id</id>
        <name>name</name>
        <tasks>
          <name>activities</name>
          <task>
            <type>hvastani.msc.jsp.randomNbrHood.Activity</type>
            <id>id</id>
            <duration>duration</duration>
            <startTime>startTime</startTime>
            <assignment>assignment</assignment>
          </task>
        </tasks>
      </job>
    </jobs>
  </instance>
</instance-defn>

```

Program 4.1: JSP problem instance type descriptor *targStructure.xml*

```

@XStream(alias="instance")
public class Instance {

    private String description;

    @XStreamImplicit(itemName="machine")
    private ArrayList<Machine> machines;

    @XStreamImplicit(itemName="job")
    private ArrayList<Job> jobs;

    //getters, setters...
    ....
}

```

Program 4.2: Internal instance object skeleton for JSP problem-type

job's operations go through a random permutation of the machines.

We also wrote a structured instance generator that implements the technique presented in [18] to generate machine correlated JSP instances with no workflows imposed on machine ordering for jobs. In machine correlated JSPs the duration of every activity going through machine  $m_i$  is correlated in the sense that it is sampled from the same Gaussian distribution. Thus each machine has its own Gaussian distribution. The further apart the various distributions are the less overlap in the durations of job operations across machines implying more structure in the problem. The parameter  $\alpha \in [0, 1]$ , controls the degree of Gaussian distribution overlap. Smaller values of  $\alpha$  mean greater overlap and less structure. A workflow imposes a pre-determined partition on the set of machines that each job must go through yielding an ordering of machine subsets such that the job must finish processing on the machines in subset  $i$  before initiating processing on subset  $(i + 1)$ . *Wk1* means no partitioning, *Wk2* means that machines are partitioned into 2 subsets and *Wkm* means  $m$  subsets.

We used  $\alpha = 0.5$  to generate our instances. The other parameters were  $(\sigma_{lb}, \sigma_{ub}, \mu_{lb})$  that were used to select  $(\sigma, \mu)$  for each machine's Gaussian distribution. We used  $\sigma_{lb} = 1$

and  $\sigma_{ub} = 20$  and  $\mu_{lb} = 30$  for the entire ensemble.

See section B.1 for a listing of the Java interface for an instance generator extension plugin and the generator descriptor file for the Taillard technique instance generator. As required by ScheduleLab, our generator extension plugin has a descriptor file named *generators.xml* (section A.9). The descriptor file declares any input parameters that the generator will need with optional validation expressions. At run-time, ScheduleLab only needs to parse *generators.xml* in order to acquire all validated parameters from the user before invoking the instance generator through the instance generator Java interface. The validation expression must be a Java boolean expression that can refer to other generator properties by name as shown in the listing. Internally, ScheduleLab uses an expression evaluation library called Jakarta Commons Jexl to evaluate validation expressions (with parameter references) [1]. Section C.1 for a snippet of the random problem instance stored in XML format.

#### 4.1.5 Experiment Session

We set up our experiment session by choosing both our algorithms as targets and selecting a data collector. We selected the `solution trace` data collector that required us to supply the target algorithm meta-data that would allow the data collector to integrate with each algorithm component to pass problem instances and receive instances and to collect solution trace data.

See section A.3 for a listing of our target instance structure and section A.10 for our data collector descriptor file. Note that the descriptor specifies the methods that ScheduleLab's data collector should call on the solver in order to obtain the solution trace tuple for a given solution improvement event. We specified our cutoff time to be 180 seconds and our data collector to be the `solution trace` data collector.

Our experiment entailed collecting solution trace data from 50 independent sample runs of a run-set. We had two run-sets: each run-set involved running both algorithms against one instance ensemble (random, structured) generated previously. We also specified that the sample size should be 50 meaning there will be 50 independent runs for each problem instance solved by each algorithm.

The contents of the solver descriptor file are specific to the data collector and specify

how the `solution trace data collector` can connect to the solver to execute a sample run and collect solution trace tuples at every improving solution (see section 3.6.3).

Once we ran the experiment session, we could fire an `analyze` action to get the `SQDs` and `RTDs` out of our raw solution traces. See section C.2 for a snippet of the solution trace raw-data for a given instance produced by the `solution trace data collector`.

## 4.2 Analysis

We used `ScheduleLab` to delegate analysis to the `QRTD and SQD analyzer` that is compatible with the output of the `solution trace data collector`. The analyzer accepts the solution traces that are the output of the `solution trace data collector` and extracts qualified run-time distributions and solution quality distributions. For the `SQDs` we must specify either the optimal solution quality for every instance or a lower bound. This is necessary for comparability of algorithm performance over ensembles of instances. We specified Taillard's lower bound (maximum of the maximum total job duration or the maximum machine job duration) for every instance [17]. In order to generate qualified run-time distributions, the analyzer requires an upper-bound on the solution quality and a step value. We specified 3.0 and 0.1 respectively. The analyzer then generated *QRTDs* corresponding to solution quality going from 0% to 300% in 10% increments. For the *SQDs* the analyzer takes as input the cut-off time and a step value to generate *SQDs* qualified by run-time. We specified 180 seconds and 1 second. The output of the analysis is saved to disk in XML format and can be inputted to a compatible visualizer to graph the *QRTDs*, *SQDs* and other statistics. See section C.3 for a snippet of the output produced by the `QRTD_SQD analyser`.

We selected the `QRTD and SQD visualizer` that takes the *QRTD* and *SQD* analysis for a given instance belonging to an ensemble and generates combined *QRTD* and *SQD* graphs showing the performance of both algorithms.

## 4.3 Experiment results

For discussion, we look at a sampling of *QRTDs* and *SQDs* for one instance from each ensemble. For each instance (random, structured) we have the combined *SQDs* for both

algorithms corresponding to 25% and 75% of the cut-off time. We also have combined algorithm *QRTDs* corresponding to a solution quality of 5.0.

Fig. 4.1 shows the combined *SQDs* for the random instance and fig. 4.2 shows the combined *QRTD* for the random instance. Similarly, Fig. 4.3 shows the combined *SQDs* for the structured instance and fig. 4.4 shows the combined *QRTD* for the structured instance.

### 4.3.1 Random instance results

As was expected, both the algorithms perform quite poorly, as seen in the *SQDs* in fig. 4.1, assuming Taillard's lower bound is tight. It's clear from the *SQDs* in fig. 4.1 that the random hill-climbing algorithm is able to find better quality solutions faster than the simple hill-climbing algorithm, on this random instance. In the case of the random hill-climbing algorithm, it does find better solutions when given more time, as seen in the shift to the left of the distribution mass. But this is barely the case for the simple hill-climbing algorithm. In fact the simple hill-climbing algorithm shows very little variability in its distribution and seems to relatively stagnate into a consistent local minima even with more run-time, as highlighted by the increasing gap between the two distributions in each combined *SQD*.

The *QRTD* in fig. 4.2 confirms the relatively superior performance of the random hill-climbing algorithm over the simple hill-climbing algorithm.

## 4.4 Structured instance results

According to the *SQDs* in fig. 4.3, the performance of both algorithms on the structured instance is similar to that on random instance. That is also true of their relative performance. This is interesting because we had expected both algorithms to perform much better on the structured instance than on the random instance, based on the belief in the literature [12] that structured instances are usually easier to solve. The random hill-climbing algorithm seems to exhibit lesser variability on the structured instance than on the random instance. Though for the simple hill-climbing algorithm there is barely any variability just like on the random instance.

The *QRTD* in fig. 4.4 suggests that both algorithms consistently reach a solution quality of 5.0 much earlier for the structured instance than for the random instance.

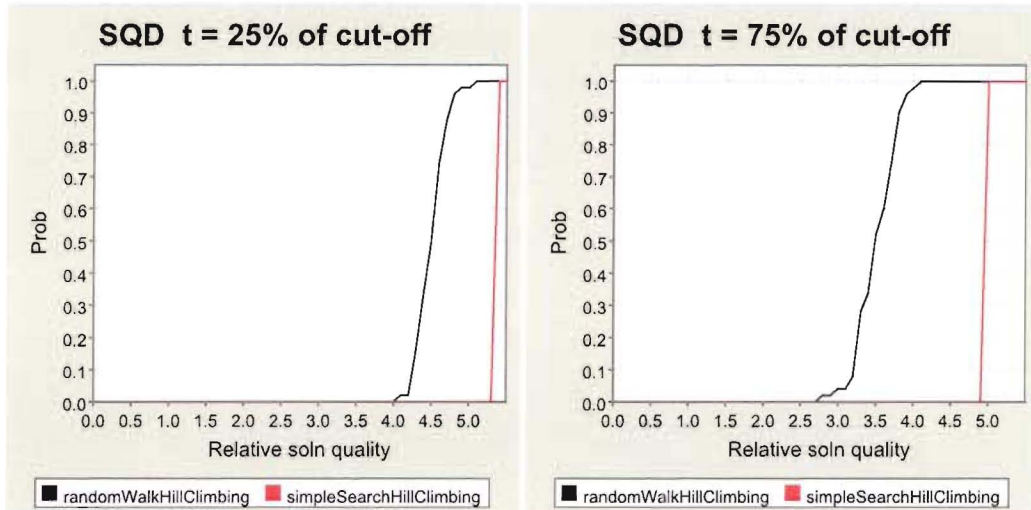


Figure 4.1: Combined SQDs for sample random problem instance

## 4.5 Discussion

Does ScheduleLab minimize the effort needed to do empirical analysis on SLS algorithms solving resource scheduling problems? When the user is only re-using ScheduleLab components and plugins to do experimentation tasks (create new instances, configure target algorithm code, configure an experiment session, execute run-sets and analyze and visualize results) the amount of effort is minimal in that the user does not need to write any code to enhance ScheduleLab and can focus completely on the experimentation part. However, the more new plugins the user needs to write in a given situation to customize ScheduleLab, the more the effort appears to be greater than that for using a library like EasyAnalyzer. This is because, the plugin contributor has to not only implement a ScheduleLab interface, but also declare meta-data in XML files in order to support the decoupling feature of ScheduleLab. We think that this is an acceptable cost to pay for the more important decoupling between ScheduleLab and the user's algorithm code. The user is free to design his algorithm code as he wants, albeit his problem instance type and solver invocation methods comply with the

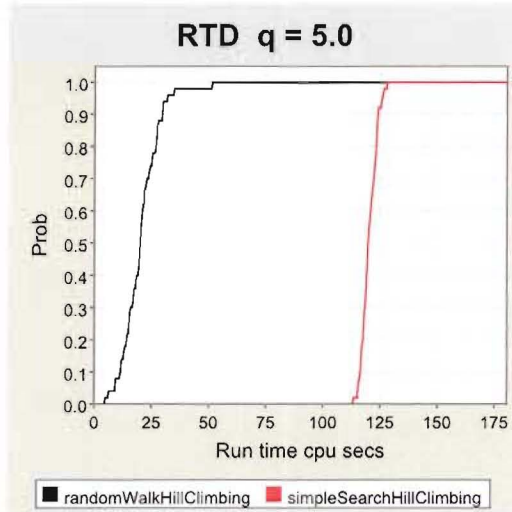


Figure 4.2: Combined QRTD for sample random problem instance

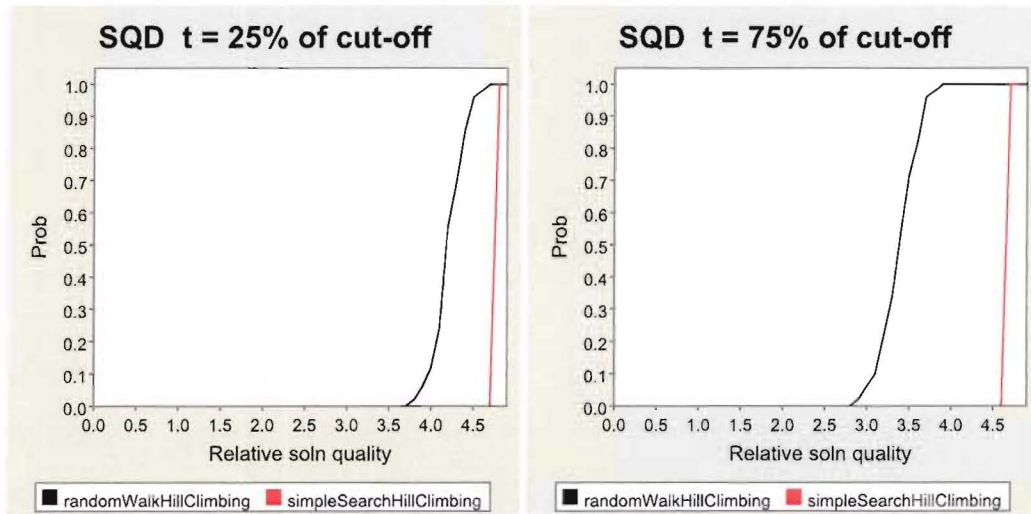


Figure 4.3: Combined SQDs for sample structured problem instance

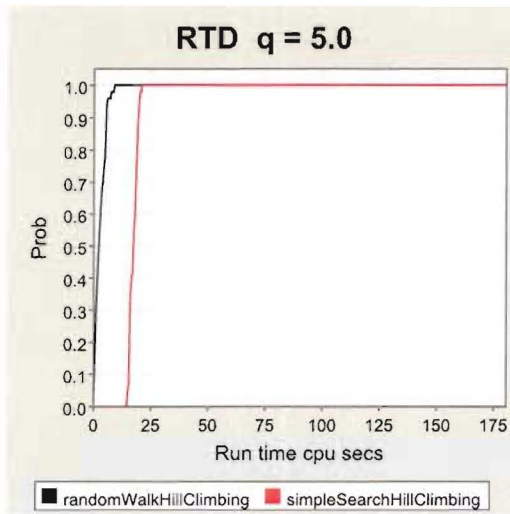


Figure 4.4: Combined QRTD for sample structured problem instance

structural requirements of a ScheduleLab problem type and data collector.

ScheduleLab delegates the responsibility for the correctness and stability of plugin code to the plugin developer. For example, the problem instance generator contributor specifies validation rules for its input parameters in the *generator.xml* file. ScheduleLab does not prevent the developer from specifying a divide-by-zero expression, for instance. Further, this project does not address any ScheduleLab specific design time tool support for ScheduleLab plugin developers to catch and diagnose errors in their plugin code or plugin configurations at design time. A plug-in developer could also shutdown the JVM by calling `System.exit()`. We acknowledge this issue, but note that our focus has been on creating a proof-of-concept usable and extensible tool for empirical analysis. This is an area that needs further investigation.

## 4.6 Summary

We discussed how we used ScheduleLab to perform preliminary empirical analysis on two hill-climbing algorithms solving JSP instances. We discussed how we did a one-time implementation of the JSP problem-type extension plugin. We decided to do our experimentation



on two (small) ensembles of instances, one generated by a ScheduleLab instance generator that implemented Taillard's random JSP technique and the other by Watson's structured machine correlated JSP technique. We used two algorithms supplied by researchers from Actenum Corporation, Vancouver, BC, one a simple first improvement hill-climbing algorithm, and the other a randomized first improvement hill-climbing algorithm. Both algorithms use a one activity re-insertion neighbourhood. The randomized algorithm selects an activity and its insertion point randomly to perform a first improvement move, while the simple hill-climbing algorithm selects an activity based on a slack heuristic and then systematically searches its insertion points to perform a first-improvement move. We presented and discussed the exported graphical output of the qualified run-time distributions and solution quality distributions of both these algorithms on the instances. Note that the experiment setup can be easily shared and replicated by another ScheduleLab user. We have illustrated how ScheduleLab as an integrated tool can be used for empirical analysis of resource scheduling SLS algorithms.

## Chapter 5

# Conclusion

In this report, we have presented ScheduleLab, a proof-of-concept Eclipse/OSGi based tool that allows industrial as well as academic researchers alike to do empirical analysis of stochastic local search algorithms that solve resource scheduling problems. ScheduleLab supports flexible problem instance ensemble generation and publishing for re-use. It also supports experimentation to study the performance characteristics of one or more algorithms solving one or more instance ensembles. ScheduleLab also provides a set of extension points for developers to contribute new capabilities and extensions to ScheduleLab. We illustrated the operation of ScheduleLab by implementing a Job Shop Scheduling problem-type capability in ScheduleLab and did preliminary empirical analysis of the performance characteristics of two local search algorithms solving JSPs.

In section 5.1 we summarize the approach taken by this work. Section 5.2 we review our contributions and finally section 5.3 we discuss future work.

### 5.1 Project Summary

Chapter 2 presented a literature review with two goals: one to identify any previous work done and two, to discuss issues and practices in empirical analysis of SLS algorithms in the literature. We looked at EasyAnalyzer, a C++ command-line driven library that is designed to integrate with cooperating local search solver implementations. However any algorithms must implement or extend EasyAnalyzer's base classes which imposes a compile time dependency on EasyAnalyzer in the target code. Further EasyAnalyzer has no support

for problem instance generation and analysis visualization and is not part of a mainstream Integrated Development Environment. We also looked at the notion that because of the non-deterministic nature of SLS algorithms, their run-time performance is characterized by a bi-variate probability distribution  $rtd(t, q)$  where  $t$  is the run-time and  $q$  is solution quality found upto time  $t$ .

Chapter 3 delved into the design and architecture of ScheduleLab. ScheduleLab is an extensible and re-usable Eclipse based plugin in that new problem types can be contributed to ScheduleLab as well as new problem instance generators, data collectors, data analyser and visualizers. ScheduleLab enforces decoupling with the user's target source code in that it has no compile time impact on the target source code. The integration of ScheduleLab and the target source code is based on a meta-data layer defined in XML configuration files that is only a run-time integration and not a compile time dependency.

Chapter 4 presented an operational evaluation of ScheduleLab by using ScheduleLab to do preliminary empirical analysis on 2 SLS algorithms that solve JSP problems. We generated a pair of instance ensembles comprising an equal number of structured as well as unstructured problem instances. We performed sample runs on each ensemble and analysed the solution trace data to extract *QRTDs* and *SQDs* for the experiment setup. We illustrated that it is possible to use ScheduleLab as an effective tool for empirical analysis.

## 5.2 Contributions

We believe there is a need to focus on effective tool support for empirical analysis of SLS algorithms in a way that supports both industrial as well as academic researchers and is based on a mainstream software platform for acceptance and sustainability. We believe that ScheduleLab is a contribution in that direction. As a tool for empirical analysis ScheduleLab:

- is extensible in that it allows researchers to develop and share their own extensions to ScheduleLab to add new problem types that can be solved using resource scheduling SLS algorithms, new problem instance generators for a given problem type, new data collectors for different experimental data collection from algorithms, new data analyzers for analyzing collected data and data visualizers for visualizing distributions and
-

statistics.

- ScheduleLab does not impose compile time dependencies on user's target code, but instead relies on a XML based meta-data layer that describes to ScheduleLab how to integrate with the user's target code at run-time (using Java Reflection API or its abstractions).

Thus, we believe that ScheduleLab is a viable alternative to comprehensive standalone libraries like EasyAnalyzer to do empirical analysis of SLS algorithms solving resource scheduling problems.

### 5.3 Future Work

We would like to pursue enhancing ScheduleLab in the following directions going forward:

- Adding more problem-types and other extensions as well as looking into expanding ScheduleLab to deal with other classes of SLS algorithms.
- Research requirements like finding the optimal tuning parameters for SLS algorithms like simulated annealing, tabu search and ant colony optimisation requires more sophisticated tool support for design of experiments methodology because we have several additional factor levels to vary in addition to just instance ensembles and algorithms. Supporting the work flow for design of experiments methodology would be an important and interesting direction to pursue.
- In ScheduleLab we assume that the user is interested in the metrics that are being published in the solution trace by the algorithm during sample runs. But this may not be the case in general. A user may want to specify custom metrics derived from the current solution. In ScheduleLab we could have the solver provide a method that ScheduleLab could invoke to get the instance when a solution improvement is reported. User specified code or scripts could then query this instance and generate the solution trace tuple. This assumes that measuring run-time is paused during publishing and processing a solution improvement event.

## Appendix A

# ScheduleLab Descriptor files

### A.1 JSP Problem Instance structure descriptor published by a problem type plugin

```
<?xml version="1.0" encoding="UTF-8"?>
<instance-defn>
  <instance><!-- No tag values. Users must fill in the values.
  <type></type>
  <desc></desc>
  <resources>
    <name></name>
    <resource>
      <type></type>
      <id></id>
      <name></name>
    </resource>
  </resources>
  <jobs>
    <name></name>
    <job>
      <type></type>
      <id></id>
      <name></name>
      <tasks>
        <name></name>
        <task>
```

```

        <type></type>
        <id></id>
        <duration></duration>
        <startTime></startTime>
        <assignment></assignment>
    </task>
</tasks>
</job>
</jobs> -->
</instance>
</instance-defn>

```

## A.2 Problem instance type structure descriptor *targStruct.xml* from instance generator

```

<?xml version="1.0" encoding="UTF-8"?>
<instance-defn>
  <instance>
    <type>hvastani.msc.jsp.taillard.Schedule</type>
    <desc>description</desc>
    <resources>
      <name>resources</name>
      <resource>
        <type>hvastani.msc.jsp.taillard.Resource</type>
        <id>id</id>
        <name>name</name>
      </resource>
    </resources>
    <jobs>
      <name>jobs</name>
      <job>
        <type>hvastani.msc.jsp.taillard.Job</type>
        <id>id</id>
        <name>name</name>
        <tasks>
          <name>activities</name>
          <task>

```

```

        <type>hvastani.msc.jsp.taillard.Activity</type>
        <id>id</id>
        <duration>duration</duration>
        <startTime>startTime</startTime>
        <assignment>assignment</assignment>
    </task>
</tasks>
</job>
</jobs>
</instance>
</instance-defn>

```

### A.3 Problem instance type structure descriptor from target solver

```

<?xml version="1.0" encoding="UTF-8"?>
<instance-defn>
  <instance>
    <type>hvastani.msc.jsp.solver1.Instance</type>
    <desc>name</desc>
    <resources>
      <name>resources</name>
      <resource>
        <type>hvastani.msc.jsp.solver1.Machine</type>
        <id>id</id>
        <name>name</name>
      </resource>
    </resources>
    <jobs>
      <name>jobs</name>
      <job>
        <type>hvastani.msc.jsp.solver1.Job</type>
        <id>id</id>
        <name>name</name>
        <tasks>
          <name>activities</name>
          <task>
            <type>hvastani.msc.jsp.solver1.Task</type>
            <id>id</id>

```

```

        <duration>duration</duration>
        <startTime>startTime</startTime>
        <assignment>assignment</assignment>
    </task>
</tasks>
</job>
</jobs>
</instance>
</instance-defn>

```

#### A.4 ScheduleLab project descriptor

```

<?xml version="1.0" encoding="UTF-8"?>
<sLabProject>
  <projName>sampleProject</projName>
  <problemTypeId>hvastani.msc.problemtypes.jsp</problemTypeId>
  <problemTypeName>Job Shop Scheduling</problemTypeName>
  <ensemblesFolder>ensembles</ensemblesFolder>
  <sessionsFolder>sessions</sessionsFolder>
</sLabProject>

```

#### A.5 Instance ensemble descriptor file *ensemble.xml*

```

<?xml version="1.0" encoding="UTF-8"?>
<ensemble>
  <problemTypeId>hvastani.msc.problemtypes.jsp</problemTypeId>
  <generatorId>hvastani.msc.generators.jsp.taillard</generatorId>
  <parameter name="nOfInstances" type="long" value="10"/>
  <parameter name="minDuration" type="long" value="1"/>
  <parameter name="maxDuration" type="long" value="99"/>
  <parameter name="jobs" type="long" value="50"/>
  <parameter name="machines" type="long" value="10"/>
</ensemble>

```

#### A.6 Experiment session descriptor file *session.xml*

```

<?xml version="1.0" encoding="UTF-8"?>
<session>
  <dataCollectorId>hvastani.msc.dataCollectors.solutionTrace</dataCollectorId>

```



```

<run-set>
  <id>random_instances</id>
  <ensembleId>taillard_10x50</ensembleId>
    <solver>
      <id>hvastani.msc.jsp.solver1.SimpleHillClimbSolver</id>
      <param name="cutOff" type="long" value="180000"/>
    </solver>
    <solver>
      <id>hvastani.msc.jsp.solver2.RndHillClimbSolver</id>
      <param name="cutOff" type="long" value="180000"/>
    </solver>
  </run-set>
  <run-set>
    <id>structured_instances</id>
    <ensembleId>watson_machineCorr_10x50</ensembleId>
      <solver>
        <id>hvastani.msc.jsp.solver1.SimpleHillClimbSolver</id>
        <param name="cutOff" type="long" value="180000"/>
      </solver>
      <solver>
        <id>hvastani.msc.jsp.solver2.RndHillClimbSolver</id>
        <param name="cutOff" type="long" value="180000"/>
      </solver>
    </run-set>
</session>

```

## A.7 Analyser descriptor file *analyser.xml*

```

<?xml version="1.0" encoding="UTF-8"?>
<analyser>
  <analyserId>hvastani.msc.analyser.sqdQrtd</analyserId>
  <dataCollectorId>hvastani.msc.dataCollectors.solutionTrace</dataCollectorId>
</analyser>

```

## A.8 Visualiser descriptor file *visualiser.xml*

```

<?xml version="1.0" encoding="UTF-8"?>
<visualiser>

```

```

<visualiserId>hvastani.msc.visualiser.sqdQrtd</visualiserId>
<analyserId>hvastani.msc.analyser.sqdQrtd</analyserId>
<analysisName>sqds_qrtds</analysisName> <!-- corresponds to sub-folder in
                                     session's analysis folder -->
</visualiser>

```

## A.9 Sample ScheduleLab instance generator plugin parameter descriptor *generator.xml*

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- describes generator type as well as parameters + validation
      expressions -->
<generators>
<generator>
  <type>hvastani.msc.jsp.generators.TaillardUniformGenerator</type>
  <name><![CDATA[Taillard Random JSP]]></name>
  <description><![CDATA[Implementation of Taillard's random JSP instance
      generator ]]></description>
  <touch-point>
    <name>createInstance</name>
  <return-type>hvastani.msc.jsp.generators.Schedule</return-type>
  <parameters> <!-- passed in as a hash-map with string keys and
      object values -->
    <parameter name="minDuration" type="long" required="true"/>
    <parameter name="maxDuration" type="long" required="true"/>
    <parameter name="seed" type="long" required="false"/>
    <parameter name="jobs" type="long" required="true"/>
    <parameter name="machines" type="long" required="true"/>
  <validations>
    <validation>
      <rule>
        <![CDATA[ (minDuration >= 0) && (maxDuration >= 0)
          && (minDuration <= maxDuration) ]]>
      </rule>
    <error-message>
      <![CDATA[ Minimum and Maximum duration must be
        non-negative and minimum
        duration must be <= maximum duration ]]>
    </error-message>
  </validations>
</generator>
</generators>

```

```

</validation>
<validation>
  <rule>
    <![CDATA[ (jobs > 0) && (machines > 0)]]>
  </rule>
  <error-message>
    <![CDATA[ Number of Jobs and Machines must be
      non-negative ]]>
  </error-message>
</validation>
</validations>
</parameters>
</touch-point>
</generator>
</generators>

```

## A.10 Sample ScheduleLab data-collector descriptor for *solution trace* data collector *dataCollector.xml*

```

<?xml version="1.0" encoding="UTF-8"?>
<dataCollector>
  <noOfSamples>50</noOfSamples>
  <executeParallel>>false</executeParallel> <!-- use threads to
      execute samples -->
  <solvers>
    <solver>
      <solverId>hvastani.msc.jsp.solver1.SimpleHillClimb</solverId>
      <name>Actenum Job Shop Simple Hill Climbing</name>
      <problem-type>hvastani.msc.schedulelab.jsp</problem-type>
      <type>hvastani.msc.jsp.solver1.Solver1</type>
      <listener>
        <type>hvastani.msc.jsp.solver1.SolverEventHandler</type>
        <begin-method>notifyBegin</begin-method>
        <end-method>notifyEnd</end-method>
        <improve-method>notifyImprovement</improve-method>
      </listener>
      <trace>
        <cpu-time>cpuTime</cpu-time>
        <op-count>opCount</op-count>
      </trace>
    </solver>
  </solvers>
</dataCollector>

```

```

        <metric>makespan</metric>
    </trace>
    <solve-method>
        <name>solve</name>
        <instance-type>hvastani.msc.jsp.solver1.Schedule</instance-type>
    </solve-method>
    <stop-method>stopSolve</stop-method>
</solver>
<solver>
    <solverId>hvastani.msc.jsp.solver2.RndNhbrHillClimb</solverId>
    <name>Actenum Job Shop Random Neighbourhood Hill Climbing</name>
    <problem-type>hvastani.msc.schedulelab.jsp</problem-type>
    <type>hvastani.msc.jsp.solver2.Solver2</type>
    <listener>
        <type>hvastani.msc.jsp.solver2.SolverEventHandler</type>
        <begin-method>notifyBegin</begin-method>
        <end-method>notifyEnd</end-method>
        <improve-method>notifyImprovement</improve-method>
    </listener>
    <trace>
        <cpu-time>cpuTime</cpu-time>
        <op-count>opCount</op-count>
        <metric>makespan</metric>
    </trace>
    <solve-method>
        <name>solve</name>
        <instance-type>hvastani.msc.jsp.solver2.JSPInstance</instance-type>
    </solve-method>
    <stop-method>stopSolve</stop-method>
</solver>
</solvers>
</dataCollector>

```

## Appendix B

# ScheduleLab extension points

### B.1 Java interface for ScheduleLab instance generators

```
public interface IInstanceGenerator<T> {  
  
    /** returns unique string id of IProblemType extension that this generator  
    corresponds to */  
    public String getProblemTypeId();  
  
    /** gets description */  
    public String getDescription();  
  
    /** creates and returns an instance of type T */  
    public T createInstance(Map<String, Object> params);  
  
    /** returns inputstream to xml file that has generators config */  
    public InputStream getGeneratorsConfig();  
  
    /** returns inputstream to target structure */  
    public InputStream getTargStructConfig();  
  
}
```

## B.2 Java interface for ScheduleLab problem type extension point

```
public interface IProblemType {
    /** get plugin id */
    public String getProblemTypeId();

    /** gets description */
    public String getDescription();

    /** register the specified target meta-data */
    public void registerInstanceTransformer(Configuration targStructureXML);

    /** converts the specified external object to internal instance type using
    registered meta-data */
    public IInstance convertToInternal(Object external);

    /** convert internal instance to target instance type using corresponding
    registered meta-data*/
    public Object convertToExternal(IInstance internal, Class targetClass);
}
}
```

## B.3 Java interface for ScheduleLab data collector

```
public interface IDataCollector {

    /** gets description */
    public String getDescription();

    /** method to start the data collector */
    public void start(IRunSet runSet,
                    IDataCollectorDescriptor dataCollectXml,
                    IProblemType problemType,
                    IXMLPersistenceEngine xml, File sessionFolder);

    /** method to interrupt and halt data collection */
    public void stop();
}
```

```
}
```

## B.4 Java interface for ScheduleLab data analyser

```
public interface IDataAnalyser {  
  
    /** returns an array of unique string id of IDataCollector extensions that  
    this analyser is compatible with */  
    public String[] getCompatibleDataCollectors();  
  
    /** gets description */  
    public String getDescription();  
  
    /** method to perform analysis */  
    public void doAnalysis(Map<String, Object> params, ISessionDescriptor  
        sessionXml, IAnalysisDescriptor analysisXml, File analysisFolder,  
        IXMLPersistenceEngine xmlEngine);  
  
}
```

## B.5 Java interface for ScheduleLab data visualiser

```
public interface IDataVisualiser {  
  
    /** returns an array of unique string id of IDataAnalyser extensions that  
    this analyser is compatible with */  
    public String[] getCompatibleDataAnalysers();  
  
    /** gets description */  
    public String getDescription();  
  
    /** method to perform visualisation */  
    public void doVisualisation(Map<String, Object> params, ISessionDescriptor  
        sessionXml, IAnalysisDescriptor analysisXml, File analysisFolder,  
        IVisualisationDescriptor visualXml, File visualsFolder,  
        IXMLPersistenceEngine xmlEngine);  
  
}
```

3



## Appendix C

# Output files

### C.1 Snippet of problem instance file generated by Taillard Uniform Generator for JSP

```
<instance probType="hvastani.msc.schedulelab.jsp" guid="25B72197-9120-4064-1DAF-645A3BB053B0">
  <parentEnsemble>Taillard</parentEnsemble>
  <description>Taillard. m = 10, n = 50</description>
  <machine id="0">
    <name>Machine 0</name>
  </machine>
  <machine id="1">
    <name>Machine 1</name>
  </machine>
  <machine id="2">
    <name>Machine 2</name>
  </machine>
  <machine id="3">
    <name>Machine 3</name>
  </machine>
  <machine id="4">
    <name>Machine 4</name>
  </machine>
  <machine id="5">
    <name>Machine 5</name>
  </machine>
  <machine id="6">
    <name>Machine 6</name>
```

```
</machine>
...
<job id="0">
  <name>Job 0</name>
  <task id="1">
    <duration>33</duration>
    <assignment reference="../../../machine[9]" />
  </task>
  <task id="2">
    <duration>60</duration>
    <assignment reference="../../../machine[8]" />
  </task>
  <task id="3">
    <duration>64</duration>
    <assignment reference="../../../machine[6]" />
  </task>
  <task id="4">
    <duration>11</duration>
    <assignment reference="../../../machine" />
  </task>
  <task id="5">
    <duration>81</duration>
    <assignment reference="../../../machine[5]" />
  </task>
  ...
</job>
<job id="1">
  <name>Job 1</name>
  <task id="11">
    <duration>62</duration>
    <assignment reference="../../../machine[5]" />
  </task>
  ...
</job>
</instance>
```

## C.2 Snippet of output produced by solution trace data collector

```

<instance-raw-data>
  <instance-guid>E77EC72F-56A7-03BF-A070-9D55946C96DF</instance-guid>
  <solver-raw-data>
    <solverId>simpleSearchHillClimbing</solverId>
    <metricNames>
      <string>makespan</string>
    </metricNames>
    <sample>
      <sample-run-pt>
        <cpuTime>10188</cpuTime>
        <opCount>2</opCount>
        <metrics>
          <double>18530.0</double>
        </metrics>
      </sample-run-pt>
      <sample-run-pt>
        <cpuTime>21469</cpuTime>
        <opCount>17</opCount>
        <metrics>
          <double>18512.0</double>
        </metrics>
      </sample-run-pt>
      ....
    </sample>
  </solver-raw-data>
</instance-raw-data>

```

## C.3 Snippet of output produced by *QRTD*, *SQD* data analyser

```

<instance-analysis>
  <instance-guid>25B72197-9120-4064-1DAF-645A3BB053B0</instance-guid>
  <solutionBound>2735.0</solutionBound>
  <solver-analysis>
    <solverId>simpleSearchHillClimbing</solverId>

```

```
<rtid>
  <q>0.0</q>
  <median>NaN</median>
  <mean>NaN</mean>
  <__0__75Quantile>NaN</__0__75Quantile>
  <__0__9Quantile>NaN</__0__9Quantile>
  <rtidPt>
    <runTimeOrLength>0.0</runTimeOrLength>
    <prob>0.0</prob>
  </rtidPt>
  <rtidPt>
    <runTimeOrLength>500.0</runTimeOrLength>
    <prob>0.0</prob>
  </rtidPt>
  <rtidPt>
    <runTimeOrLength>1000.0</runTimeOrLength>
    <prob>0.0</prob>
  </rtidPt>
  <rtidPt>
    <runTimeOrLength>1500.0</runTimeOrLength>
    <prob>0.0</prob>
  </rtidPt>
  <rtidPt>
    <runTimeOrLength>2000.0</runTimeOrLength>
    <prob>0.0</prob>
  </rtidPt>
  <rtidPt>
    <runTimeOrLength>2500.0</runTimeOrLength>
    <prob>0.0</prob>
  </rtidPt>
  <rtidPt>
    <runTimeOrLength>3000.0</runTimeOrLength>
    <prob>0.0</prob>
  </rtidPt>
  <rtidPt>
    <runTimeOrLength>3500.0</runTimeOrLength>
    <prob>0.0</prob>
  </rtidPt>
  ...
```

```
<rtPt>
  <runTimeOrLength>180500.0</runTimeOrLength>
  <prob>1.0</prob>
</rtPt>
</rtd>
...
<sqd>
  <rt>110000.0</rt>
  <median>4.395612431444241</median>
  <mean>4.449250084850075</mean>
  <__0__75Quantile>4.892870201096892</__0__75Quantile>
  <__0__9Quantile>5.244826325411335</__0__9Quantile>
  <sqdPt>
    <relSolnError>0.0</relSolnError>
    <prob>0.0</prob>
  </sqdPt>
  <sqdPt>
    <relSolnError>0.1</relSolnError>
    <prob>0.0</prob>
  </sqdPt>
  ...
  <sqdPt>
    <relSolnError>5.499999999999964</relSolnError>
    <prob>1.0</prob>
  </sqdPt>
</sqd>
</solver-analysis>
</instance-analysis>
```

# Bibliography

- [1] Apache Software Foundation. *Jakarta Commons Jexl* (<http://commons.apache.org/jexl/>).
- [2] Jørgen Bang-Jensen, Marco Chiarandini, Yuri Goegebeur, and Bent Jørgensen. Mixed models for the analysis of local search components. In *SLS*, pages 91–105, 2007.
- [3] Wayne Beaton. *Eclipse Platform Technical Overview*. Eclipse Foundation, 2006.
- [4] Jim D’Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java(TM) Developer’s Guide to Eclipse (Paperback)*. Addison-Wesley Professional, May 2003.
- [5] Luca Di Gaspero, Andrea Roli, and Andrea Schaerf. Easyanalyzer: An object-oriented framework for the experimental analysis of stochastic local search algorithms. pages 76–90. 2007.
- [6] Eclipse Foundation. *Eclipse Documentation*. Eclipse Foundation, 2007.
- [7] Luca Di Gaspero, Andrea Roli, and Andrea Schaerf. Easyanalyzer: An object-oriented framework for the experimental analysis of stochastic local search algorithms. In Thomas Stützle, Mauro Birattari, and Holger H. Hoos, editors, *SLS*, volume 4638 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2007.
- [8] Bruce Hajek. Cooling schedules for optimal annealing. *Math. Oper. Res.*, 13(2):311–329, 1988.
- [9] J. N. Hooker. Needed: An Empirical Science of Algorithms. *Operations Research*, 42, 1994.
- [10] J. N. Hooker. Testing heuristics: We have it all wrong. In *Journal of Heuristics*, pages 33–42, 1995.
- [11] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search : Foundations & Applications (The Morgan Kaufmann Series in Artificial Intelligence) (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, September 2004.

- [12] Anant Singh Jain and Sheik Meeran. A multi-level hybrid framework applied to the general flow-shop scheduling problem. *Computers & OR*, 29(13):1873–1901, 2002.
- [13] Sun Microsystems. *Java Reflection API tutorial* (<http://java.sun.com/docs/books/tutorial/reflect/>). Sun Microsystems.
- [14] Enda Ridge and Daniel Kudenko. Tuning the performance of the *MMAS* heuristic. In *SLS*, pages 46–60, 2007.
- [15] Matthew Sgarlata. *Morph reference documentation*. <http://morph.sourceforge.net>, 2006.
- [16] Thomas Stutzle, Mauro Birattari, and Holger H. Hoos, editors. *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics, International Workshop, SLS 2007, Brussels, Belgium, September 6-8, 2007, Proceedings*, volume 4638 of *Lecture Notes in Computer Science*. Springer, 2007.
- [17] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.
- [18] Jean-Paul Watson. *Problem Difficulty and Fitness Landscapes of Structured and Random Job-Shop Problems: What Do Existing Analysis Techniques Really Tell Us?* <http://citeseer.ist.psu.edu/497241.html>.
- [19] Jean-Paul Watson. *Empirical modeling and analysis of local search algorithms for the job-shop scheduling problem*. PhD thesis, Colorado State University, Fort Collins, CO, USA, 2003. Adviser-Darrell Whitley and Adviser-Adele Howe.
- [20] Jean-Paul Watson, Laura Barbulescu, L. Darrell Whitley, and Adele E. Howe. Contrasting structured and random permutation flow-shop scheduling problems: Search-space topology and algorithm performance. *INFORMS Journal on Computing*, 14(2):98–123, 2002.
- [21] D. Whitley, J.P. Watson, A. Howe, and L. Barbulescu. Testing and evaluation and performance of optimization and learning systems. *Keynote Address: Adaptive Computing in Design and Manufacturing*, 2002.