OAT HowTo: High-Level Domain, Problem, and Algorithm Implementation

JASON BROWNLEE
Technical Report 20071218A
Complex Intelligent Systems Laboratory, Centre for Information Technology Research,
Faculty of Information and Communication Technologies, Swinburne University of Technology
Melbourne, Australia
jbrownlee@ict.swin.edu.au
or
jbrownlee AT users.sourceforge.net

I. INTRODUCTION

This document provides a high-level '*how to*' for implementing a new problem domain and new problem and algorithm instances for new and or existing problem domains in the Optimization Algorithm Toolkit (OAT) software package. Sufficient detail is provided such that the new domains, problems, and algorithms are compatible with the excising OAT framework, and existing *Explorer* and *Experimenter* graphical user interfaces (GUI's). This document is suitable for Java developers of moderate skill and assumes a prior selection and knowledge of a desired domain, problem, and or algorithm definitions. It is suggested that this document be read in conjunction with examining the OAT source tree to provide a fuller understanding of the examples provided.

The OAT Software website is `http://optalgtoolkit.sourceforge.net` and the OAT Projects website is `http://sourceforge.net/projects/optalgtoolkit`. This document refers to OAT version 1.4, as of December 2007.

II. OVERVIEW

The OAT software provides both a framework for implementing new problem domains, problem instances, and algorithm instances, as well as base implementations of common domains, problems, and algorithms. The library was designed for extensibility where one may add new problem and or algorithm instances to an implemented problem domain, such as a new function definition for Continuous Function Optimization or binary trap function for Binary Function Optimization. The base implementations are few, thus one may add an entirely new problem domain with new problem and algorithm instances. This document provides a high-level summary of implementing a new problem domain, problem instance, and algorithm instance, indicating the important parts of the source tree and suggested code conventions as needed. OAT contains three primary modalities: an explorer GUI (`com.oat.explorer.*`) for exploratory experimentation and analysis with algorithm and problem instances with visualisation tools, an experimenter GUI (`com.oat.experimenter.*`) for formal experimentation and analysis with statistical tools, and the integration of problems, algorithms, and tools into custom programs. This document focuses on the first two examples, and assumes the third case can be inferred from existing unit tests (`com.oat.junit`), example codes (`com.oat.experimenter.examples`), and this document.

The OAT software and library is open source, and the source code is provided in the download[1] and is also available via remote access to the Concurrent Versions System (CVS)[2]. When extending the OAT framework in anyway way, one may operate directly on the OAT

[1] http://downloads.sourceforge.net/optalgtoolkit
[2] http://sourceforge.net/cvs/?group_id=182624

codebase, or create a new project that uses the OAT codebase as a source library. The first case may be used for personal projects or by developers on the OAT project, the second case may be used for projects to be publically released as plug-ins or extensions to the platform. This second case is preferred for personal projects, and mechanisms exist within the framework such that so-called *OAT plug-ins* are supported in the GUI's. In all cases, the most recent version of the source tree itself provides the definitive documentation and should be examined for examples and inspiration. The philosophy of OAT is to standardise experimental computational intelligence, so we are interested in any contribution you can make to the project toward realising this vision.

III. IMPLEMENTING A PROBLEM DOMAIN

The domain (`com.oat.Domain`) is a construct that provides a point of aggregation for algorithms and problem instances and other aspects related to a run. It exists to provide structured access and tools to such objects to facilitate automation in the GUI and in the batch execution of unit tests and experimentation including algorithms, problem instances, stop conditions and run probes. A run is defined as a the execution of a problem definition with an algorithm definition. The extent of a run execution is defined by a *stop condition*, and extent of the information collected from the run is defined by the *run probes*. Domains are organised under the domains package (`com.oat.domains`) with base examples including Continuous Function Optimization (CFO) (`com.oat.domains.cfo`), the Travelling Salesman Problem (TSP) (`com.oat.domains.tsp`), Protein Structure Prediction (PSP) (`com.oat.domains.psp`), and the Graph Colouring Problem (GCP) (`com.oat.domains.gcp`).

To implement a new problem domain, extend the abstract domain class (`com.oat.Domain`) and implement the require abstract methods. Two important abstract methods are those for preparing lists of algorithms and problems (`loadAlgorithmList` and `loadProblemList`). These methods are used in the GUI's to allow the user to choose among a variety of problem and algorithm instances. For the base domains, these methods use external properties files that define the extent of problem and algorithm instances to prepare for a given domain. For example, the CFO domain uses the files `algorithms.cfo.properties` and `problems.cfo.properties` that define specific fully qualified class names of algorithms and problems to load for the domain[3], in the case of TSP, the problem properties file contains problem definition file names. Alternatively, for smaller domains one may explicitly list the algorithm and problem instances in the domain implementation. Load methods are also used for domain stop conditions and run probes (`loadDomainStopConditions` and `loadDomainRunProbes`), although default implementations are provided with generic implementations of each. These may be overridden and either augmented or replaced with lists of additional domain specific implementations. One may implement a domain and or problem instance specific visualisation (by extending `com.oat.explorer.gui.plot.GenericProblemPlot`) which is displayed in the explorer GUI to provide a indication of the user of the nature of the selected problem instance. To exploit this feature, override the problem plot method and return an instance of the custom plot implementation. Two examples of this feature are CFO that provides a 3D visualisation of selected functions (`com.oat.explorer.domains.cfo.gui.plots.ThreeDimensionalFunctionPlot`), and TSP that provides a 2D visualisation of optimal tours for selected data sets (`com.oat.explorer.domains.tsp.gui.plots.TourPanel`). Finally, another important method for the explorer GUI is for retrieving the domain specific explorer panel. This panel, which must extend a generic explorer panel (`com.oat.explorer.gui.panels.MasterPanel`) provides a workspace in the explorer for working with the domain including the GUI elements for plotting and visualisation. For example the CFO domain provides a customised explorer master panel that provides a real-time 2D plot of samples in the function space

[3] Note that all properties files are organised into the default base package (no package) in the source tree.

(com.oat.explorer.domains.cfo.gui.panels.CFOMasterPanel). A generic explorer master panel is provided that provides a rudimentary interface and generic plots (com.oat.explorer.gui.panels.DefaultMasterPanel).

A properties file is used to maintain a list of all domain classes (domainlist.properties). This list is used by the generic OAT launcher for selecting domains to load into the explorer and by the experimenter to select a domain in which to perform experiments. One may add fully qualified domain class names directly to this master list. Alternatively, one may create a user domain list, which is automatically loaded by the master domain list (userdomainlist.properties). This second case is useful for OAT plug-ins that use the OAT software as a dependant library and want their custom domain to be available in the OAT GUI.

IV. Implementing a Problem Instance

A problem instance (com.oat.Problem) represents a reusable implementation of a problem definition and is responsible for validating and evaluating (assigning an ordinal costing) to problem solutions (com.oat.Solution). Example problem instances include definitions of functions that evaluate vectors of continuous numbers in the CFO domain (com.oat.domains.cfo.problems.*), definitions of functions that evaluate strings of binary numbers and convert vectors of continuous numbers to strings of binary numbers for evaluation in the BFO domain[4] (com.oat.domains.bfo.problems.*), and a generic definition of a TSP problem instance for the TSP domain (com.oat.domains.tsp.TSPProblem) that can be initialised with TSP problem data from file in the TSPLib format[5].

To implement a problem instance extend the abstract problem class and (com.oat.Problem) implement the required methods. The most critical method in this class is the cost method (cost) that provides a pipeline for validating and evaluating solutions, and calls the abstract method that defines the problem specific logic for evaluating solutions (problemSpecificCost). Solutions are intended to be atomic and immutable after creation with regard to the problem specific data, and the solution evaluation assigned is intended to be static. Thus, the default behaviour is for the cost function of the problem to only evaluate and assign a costing to those solutions that are not already evaluated. This functionality may be modified by overriding the generic cost function or the evaluation concerns in the solution class (com.oat.Solution). The costing assigned to problems is expected to be ordinal, thus the problem may be defined as either maximising or minimising cost (isMinimization). Before solutions are evaluated, they are validated to ensure the problem specific data they contain matches the expectation of the problem definition (checkSolutionForSafety). For example, in the case of CFO, a generic parent CFO problem instance (com.oat.domains.cfo.CFOProblem) provides functionality for ensuring a submitted solution contains the required number of continuous variables and that each variable is within the defined bounds.

Both problem instances and algorithm instances may have user-modifiable parameters. For effective use of such parameters in the explorer and experimenter GUI, parameters are expected to have well-formed and symmetric accessor and mutator methods (‘get’ and ‘set’). This also applies to boolean parameters which must use the ‘get’ and ‘set’ convention instead of the ‘is’ JavaBean convention. This is because of the automated discovery of parameters used in algorithm and problem instance configuration in the GUI, and the automated externalisation mechanisms for saving configurations to file in the experimenter GUI (see the generic implementation provided of the com.oat.Populator interface). By default, problem

[4] This conversion allows algorithms that generate solutions as vectors of real numbers to operate on problems from the BFO domain that expect strings of binary digits (ones and zeros).
[5] See http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

instances are not user configurable (the `isUserConfigurable` method and the `com.oat.Configurable` interface), if user parameters are added to the problem instance, the user configurable method must be overridden to return true. An example of this is the generic CFO problem ancestor that provides a configurable number of dimensions and method of converting vector solutions to binary strings[6].

When problems are executed in the run pipeline (see `com.oat.AlgorithmExecutor`) a generic contract for problems is required involving validation (a generic implementation of `validateConfiguration` is provided that calls the abstract method `validateConfigurationInternal`), initialisation (`initialiseBeforeRun`), and cleanup (`cleanupAfterRun`). The validation method should validate the problem definition including any user parameters. The initialisation and cleanup provide facility for the allocation and de-allocation of resources needed when a problem instance is being used in a run. Problem instances are reusable, thus initialisation is required to ensure a given instance is ready for use. For example in the case of the generic problem definition in the TSP domain (`com.oat.domains.tsp.TSPProblem`) the initialisation method is used to load specific TSP data from file the first time it is called.

Once a problem instance is defined, it must be acknowledge by the domain to be used within the OAT GUI's. If the problem instance is for an existing domain, it may be added to the domains problem list, otherwise it may be included in the problem domains problem-loading method.

V. IMPLEMENTING AN ALGORITHM INSTANCE

An algorithm instance (`com.oat.Algorithm`) represents a reusable strategy for generating solutions to be validated and evaluated by problem instance of a specific problem domain. Examples include a series of evolutionary algorithms and hill climbers for the BFO domain (`com.oat.domains.bfo.algorithms.*`) that provide generic and reusable strategies for generating binary strings which may be suitable for BFO problems as well as problem instance of other domains to which binary strings can be converted to viable solutions (such as vectors of continuous variables for the CFO domain).

To implement an algorithm instance you must extend the abstract algorithm class (`com.oat.Algorithm`) and implement all abstract methods. An algorithm strategy is executed in the presence of a problem instance, thus the most important method of an algorithm is the execution method (`executeAndWait`) that calls the internal abstract version that must provide the specific strategy for generating solutions (`internalExecuteAlgorithm`). As is the case with problem instances, algorithm instances can have user configurable parameters that must use well-formed and symmetrical accessor and mutator ('get' and 'set') methods to be usable within the OAT GUI's. Also as in the case of problem instances, algorithm instances are subjected to a contact by the run pipeline (see `com.oat.AlgorithmExecutor`) of validation (`validateConfiguration`), initialisation (`initialiseBeforeRun`), and cleanup (`cleanupAfterRun`). Importantly, both the initialisation and cleanup methods provide a reference to the problem instance the algorithm is and was executed against. This allows an algorithm strategy to adjust any automatic configurations based on problem specific information. For example, the initialisation method is used by many population-based strategies for the creation of an initial population of solutions.

An algorithms execute method (`executeAndWait`) manages the extent of the execution of the strategy including the management of what generated solutions to evaluate and when to evaluate them (by calling the `cost` method on the problem instance). A common pattern for computational intelligence strategies especially those that employ a memory to iterative

[6] This conversion allows algorithms that generate binary string solutions from the BFO domain to function of continuous function problem instances of the CFO domain.

improve sampling (such as a population of samples) is to execute in epochs. A generic epoch algorithm strategy is provided that may provide a basis for a given algorithm (`com.oat.EpochAlgorithm`). The epoch algorithm manages the execution of an algorithm strategy with a sample solution-based memory including evaluation of generated samples, providing abstract methods for the initialisation (`internalInitialiseBeforeRun`), repeated generation of samples called epochs (`internalExecuteEpoch`), and any post epoch cleanup (`internalPostEvaluation`). For example, the Ant Colony Optimisation (ACO) algorithms instances for the TSP problem domain (`com.oat.domains.tsp.algorithms.aco`) exploit the post epoch evaluation method to integrate the knowledge obtained through the evaluation of samples to update an internal quality map of tour components called a pheromone map used for the probabilistic step-wise construction of new solutions.

A first generic strategy for any newly created problem domain is the *random strategy* (*random search* or *random algorithm*). Given the simplicity of the approach, a generic implementation is provided (`com.oat.algorithms.GenericRandomSearchAlgorithm`) that may be specialised for the generation of random samples for any domain (via the `generateRandomSolution` method). Finally, given that many optimization strategies have accepted heuristics for the configuration of user parameters based on available information from a problem instance, a generic mechanism is provided for the automatic configuration of algorithm instances (implement the `AutomaticallyConfigurableAlgorithm` interface). An automated configuration method (`automaticallyConfigure`) is called by the run pipeline (see `com.oat.AlgorithmExecutor`) providing a reference to the problem instance such that an algorithm instance can automatically determine configuration parameters. The use of automatic configuration is provided in the explorer GUI via a user selectable checkbox. The ACO approaches for the TSP domain (`com.oat.domains.tsp.algorithms.aco`) as well as the evolutionary algorithms for the BFO domain (`com.oat.domains.bfo.algorithms.evolution`) exploit this mechanism to configure their parameters based on best practice.

VI. Summary

This work provided high-level information on how to implement a new problem domain and new problem and algorithm instances for new and or existing problem domains. It highlighted the main ancestor classes that must be extended in each case, a summary of related code and naming conventions, and some reasoning behind related framework design decisions.

This *how-to* demonstrated the primarily method for extensibility in the OAT software as the extension of existing ancestor classes. The examples provided only covered a small but important area of the framework. Not included in this discussion were examples for stop conditions (`com.oat.StopCondition` and `com.oat.stopcondition.*),` run probes (`com.oat.RunProbe` and `com.oat.probes.*`), explorer visualisations (`com.oat.explorer.gui.plot.GenericProblemPlot`), or experimenter statistical tests (`com.oat.experimenter.stats.normality.*` and `com.oat.experimenter.stats.analysis.*`). These remain exercises for the reader, and potentially future documentation. Other topics that remain subjects for future documentation include integration of OAT concerns into custom software (see unit tests `com.oat.junit` and examples `com.oat.experimenter.examples` as starting points), the architecture of the execution pipeline (see `com.oat.AlgorithmExecutor`) and the experimentation subsystem (`com.oat.experimenter.*`) including the offline experimentation template (`com.oat.experimenter.TemplateExperiment`).

If you have any suggestions and or tips and tricks for improving this document, please send correspondence to Jason Brownlee: *jbrownlee AT users.sourceforge.net.*

Acknowledgements

Thankyou to Tim Hendtlass for his support, and for Antonio Ianiero for prompting me and providing the tipping point for finishing and releasing this document.