OSCP: Optimization Service Connectivity Protocol

Obi C. Ezechukwu
Department of Computing
Imperial College London
Exhibition Road
London SW7 2BZ
United Kingdom

mailto:oce@doc.ic.ac.uk

Istvan Maros
Department of Computing
Imperial College London
Exhibition Road
London SW7 2BZ
United Kingdom

mailto:im@doc.ic.ac.uk

Departmental Technical Report 2004/5

ISSN 1469-4174

May 2004

Abstract

Optimization software e.g. solvers and modelling systems, expose software and vendor specific interfacing mechanisms to client applications e.g. decision support systems, thus introducing close coupling. The 'Optimization Service Connectivity Protocol' (OSCP) is an abstraction of the interfaces to optimization software, which is aimed primarily at simplifying the process of integrating optimization systems into software solutions by providing an abstracted, uniform and easy to use interface to such systems, regardless of system or vendor specific requirements. This paper presents a high-level overview of OSCP including descriptions of its main interfaces, and illustrates its use via examples.

Keywords: *Java, Optimization, and Decision Support*

1 Introduction

The *Optimization Service Connectivity Protocol (OSCP)* is a Java abstraction of the interfaces to optimization software, intended primarily for alleviating the burden of implementing bridging or interfacing routines to such software. In order to appreciate its purpose, it is important to first consider the process of integrating optimization systems into software solutions e.g. decision support systems. A common approach is to represent models as programming language structures using a high-level programming language such as C++ or C, and creating 'bridges' i.e. interfaces to specific solvers. Alternatively, the solver could also be embedded in, or implemented as part of the solution using a callable library approach, the latter of which is very rare and should be avoided except in situations where it is necessary to implement algorithms that exploit specific characteristics of the model(s) being solved. Another approach is to implement model(s) using a high level algebraic modelling language supported by a modelling system, and invoking the modelling system via application libraries or command-line arguments. The advantage of this approach is that the task of compiling models to the 'matrix' formats expected by target solvers is delegated to the modelling system. With either approach, there is a requirement to implement custom interface routines to the target optimization system, which is a time consuming affair. Moreover, this introduces tight coupling between software solutions and target optimization systems. Additionally, in the same way that certain solvers are better at, or are aimed exclusively at solving certain types of models, individual modelling systems have specific strengths or features which make them particularly suitable for certain types of models. Consequently, a change in the structure of the model(s) used by the client application could trigger a change in the target optimization system, leading to additional software development effort.

The interface to a piece of software is generally defined by one or more components or subroutines, and their corresponding I/O data structures. In the case where it is possible to define abstractions for the data formats, it is possible to abstract away from the interfaces altogether. This is possible where the data abstraction can easily be transformed to representations used by the target software, or where the target software explicitly supports the formats or structures dictated by the abstraction. A good example of an interface abstraction is the JDBC [5] API for the Java language, which abstracts away from interfaces used by database API, the only requirement being that the target databases are SQL compliant. In essence, the SQL language provides a common query input format.

In the case of optimization software, AML [2], ORML [4] provide abstractions of the I/O formats utilised by algebraic optimization systems, and as such, it is possible to build on these in order to define a generic set of interfaces for such software. OSCP achieves this by utilising the typing mechanisms and data syntax provided by AML and ORML, and insulates the client application from the interfacing requirements of the underlying optimization software.

OSCP in itself does not provide the functionality required to interface to an optimization system, but rather delegates this task to specific ***implementations*** of its interfaces, where each *implementation* is supplied by a ***provider***. The terms 'implementation' and 'provider' are used to distinguish software and vendor responsibilities. OSCP in essence specifies a software contract i.e. describes what the software should look like and how it should behave; a specific *implementation* is a piece of software that satisfies this contract. The *provider* is the party, individual, or organisation that supplies i.e. makes available a particular *implementation.* However, a number of reference implementations are also provided in the OSCP distribution, including a full-featured implementation for the MPL modelling system. Additional implementations are also available for the GAMS and AMPL modelling systems. The major strength of the OSCP toolkit lies in the fact that client software are insulated from the details of the underlying implementation including the means by which it is loaded or invoked. In simple terms, interaction is with a software entity that satisfies a particular contract regardless of how the entity chooses to do so.

This paper summarises the objectives of the OSCP toolkit; provides a brief overview of its core interfaces and functionality; and indicates the future direction of the toolkit particularly in terms of enhancing the functionality contained in the distribution.

2 Objectives

OSCP is intended solely for easing the process of interfacing to optimization systems, and as such, its main aims and objectives revolve around this. These can be summarised as follows:

i. **Ease of integration:** It is likely that optimization software, which forms part of a decision support solution, will have to be integrated into the existing IT infrastructure of the organisation wishing to implement the solution, or has to be integrated with other software which contribute to the solution. OSCP aims for an easy and well-structured method of integrating optimization systems into software solutions by abstracting away from the interfacing requirements of each individual system.

ii. **Interoperability:** Different software specify different interfaces and data structures, thereby complicating the task of utilising multiple software in tandem. The requirement for combining software is one that often arises in real world applications of operations research theory. The challenges associated with meeting this requirement become clearer if we consider a situation where the output from one software is required as input to another. If both use different data structures or data representation technology, combining both would require a programming exercise to convert the data structures utilised by one into that expected by the other. The OSCP recommendation provides an easy means of building interoperable solutions. A single and straightforward typing mechanism is provided by AML and ORML; consequently, client applications can rely on a single data representation format. Furthermore, the interface to optimization software is abstracted; as such, client applications have a single unified interface irrespective of the interfacing requirements of each software.

iii. **Vendor Independence:** The OSCP interfaces do not incorporate any vendor specific constructs and are completely vendor independent i.e. do not require explicit vendor support. Vendor implementations may of course make use of vendor libraries and ultimately have to invoke vendor software, however client applications are completely oblivious of which implementation is being used at any point in time or how it is loaded or invoked. As such, applications can be designed and implemented in a vendor independent fashion.

iv. **Portability:** OSCP is specified as a set of Java classes, abstract classes and interfaces which implies that in itself, it is platform independent thus portable across different hardware and OS architectures. It relies on XML based typing mechanisms, which implies the portability of its data structures. This does not however mean that implementations may not make use of hardware or OS specific features, in which case users should be made aware of the limitations of the specific distribution via its associated documentation.

3 Basic Functionality

The main OSCP package (illustrated in figure 3.1) is *"org.elsinore.oscp"*, and contains the top-level classes and interfaces. These include interface abstractions for *optimizers,* and the *factories* used to create them. The term '*optimizer'* is used to refer to an abstraction of a piece of software capable of accepting problem input in AML format, searching for an optimal solution to the problem, and producing an ORML solution report.

The following subsections describe how the main OSCP classes and interfaces are used and the role they play in the optimization process. The descriptions are written in the context of using *optimizers* for the purpose of obtaining ORML solution reports for optimization problems.

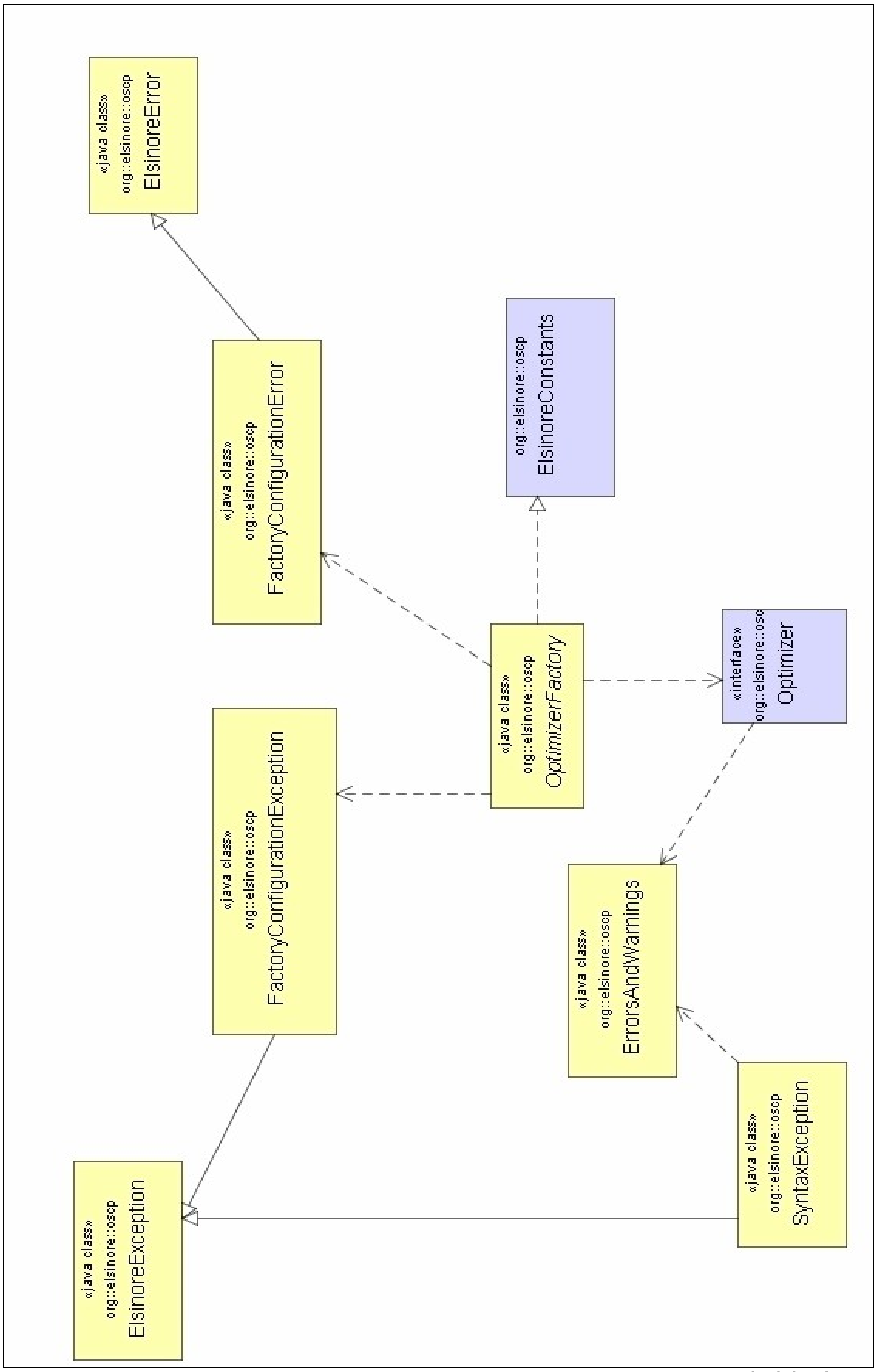

Figure 3.1: OSCP top-level class diagram

3.1 Creating an Optimizer

The ***Optimizer*** interface illustrated in figure 3.1, defines the interface contract for an optimization system. An Optimizer is created by an ***OptimizerFactory*** instance, which in turn is supplied by a vendor and can be specified by the system property *"org.elsinore.oscp.OptimizerFactory".* This property names a concrete subclass of the OptimizerFactory to serve as the default factory implementation for the Java virtual machine process. For example, to set the default factory to the MPL reference implementation, we pass in the following arguments to the java virtual machine using the '-D' flag:

```
java -Dorg.elsinore.oscp.OptimizerFactory=com.elsinore.ri.mpl.MPLOptimizerFactory ……….
```

An instance of the default factory is obtained by invoking the *'newInstance'* method, as illustrated by the following code segment:

```
private OptimizerFactory getFactory(){
   return OptimizerFactory.newInstance();
}
```

The *'getFactory'* method shown above can be re-written to override the default factory by passing in a new factory name to the *'newInstance'* method as follows:

```
private OptimizerFactory getFactory(){
   return OptimizerFactory.newInstance("com.elsinore.ri.gams.GAMSOptimizerFactory");
}
```

Once created, a factory may need to be configured, depending on platform or vendor specific requirements. Although not possible to standardise or abstract away from vendor or platform specific configuration requirements, it is possible to provide a single or consistent means of specifying configuration information. This is achieved via the *'configureFactory'* method which accepts a *java.util.Map* instance. The precise contents of the map i.e. the configuration set is specified by the vendor, however developers can insulate the application from changes in the vendor configuration set using standard and straightforward Java mechanisms. For example, by reading the configuration from a properties file, changes to the configuration need only result in changes to the properties file as opposed to application code.

In the case of the MPL implementation included in the OSCP distribution, two configuration parameters are mandatory, and the following code segment illustrates how these can be set:

```
private Collection getSolvers()
{
   ArrayList result = new ArrayList();
   result.add("Conopt.dll");
   return result;
}

private HashMap getFactoryConfiguration()
{
   HashMap result=new HashMap();
   result.put("com.elsinore.ri.mpl.workingdirectory","C:\\MPL");
   result.put("com.elsinore.ri.mpl.solvers", getSolvers());

   return result;
}
```

```
public static void main (String[] args)
{
   try
   {
      TestMPLOptimizer l_tmoTestOptimizer=new TestMPLOptimizer();
      OptimizerFactory l_ofFactory = l_tmoTestOptimizer.getFactory();
      logger.info("Created MPL Optimizer instance");

      l_ofFactory.configureFactory(l_tmoTestOptimizer.getFactoryConfiguration());

                                             …
                                             …
                                             …
```

As is clear from the code, the *'getFactoryConfiguration'* method creates a map containing the configuration set for the target optimizer, which includes the working directory for the MPL process, and the list of solvers available to the process.

An Optimizer instance is obtained from a factory by invoking the *'createOptimizer'* method, as illustrated below:

```
Optimizer l_oOptimizer=l_ofFactory.createOptimizer();
```

A factory can be re-used multiple times once created and configured i.e. there is no limit to the number of optimizers it can create. However, factories are not required to be thread-safe, as such it is advisable to utilise external synchronization mechanisms in multi-thread applications, except indicated otherwise in the vendor documentation.

3.2 Working with Optimizers

It is possible that several optimizer instances can share the same underlying optimization system instance or process, however unlike factories, Optimizer instances are required to be thread safe, consequently any underlying pooling of resources should be transparent to client applications. Optimizers are also conversational i.e. they maintain state, which restricts the number of models that can be processed concurrently by an optimizer to one. If there is a need to process several models concurrently, multiple optimizer instances should be used.

Optimizers can be used perform three main operations: obtaining the native representation of a model; obtaining the native representation of a model instance; and solving a model instance. The term 'native representation' is used to refer to the system or vendor specific syntax or representation format used by the target optimization system. For example, in the case of the MPL reference implementation, an optimizer obtained from the factory would use the MPL modelling language as its native representation. This is intuitive as the model ultimately has to be processed and solved by the MPL modelling system, which accepts as input valid MPL models. The native representation is most likely obtained by applying one or mode XSL [7] stylesheets to the supplied AML model and instance data. This is the technique employed in the reference implementations included in the OSCP distribution. It is possible to achieve the same goal using custom code or programming constructs, but this is not recommended as XSL was created specifically for this purpose i.e. transforming XML [6] based content.

An optimizers state is initialised by the specifying the model on which it is to operate. For most applications it will also be necessary to specify the models data, e.g. if a solution to the model is required. The model is specified by calling the '*setModel*' method on the optimizer instance, and the instance data is specified by calling the *'setInstanceData'* method:

```
String l_strAMLModel= ……
String l_strAMLdata= …….

l_oOptimizer.setModel(l_strAMLModel);
l_oOptimizer.setInstanceData(l_strAMLdata);
```

The *'setModel'* method has the effect of clearing all previously stored data from the optimizer's storage arrays, consequently starting a new 'conversation'. In the case where a model is solved repeatedly using different data sets, there is no need to call the *'setModel'* method each time the data set changes. Calling the *'setInstanceData'* method accomplishes this task on its own, and does not affect the conversational state of the optimizer. However, it is likely that with each new data set, the resulting model instance would have to be validated, before any instance dependent operations are performed such as obtaining a solution.

Some applications may require access to the native representation of the model or an instance of it, possibly for presentation purposes or error debugging. Two convenience methods are provided for facilitating this, namely: *'getNativeModelRepresentation'* and *'getNativeInstanceRepresentation'*. The use of these two methods is illustrated below:

```
String l_strNativeRepresentation = l_oOptimizer.getNativeModelRepresentation();
String l_strNativeInstanceRepresentation = l_oOptimizer.getNativeInstanceRepresentation();
```

The latter of these two calls is quite likely to produce highly verbose output depending on the size of the data set. This is due to the fact that the native instance representation actually includes the native representation of the instance data i.e. parameter and set data.

The following code sequence illustrates a request to the optimizer to solve or attempt to solve the current model instance:

```
String l_strORMLresult = l_oOptimizer.solveInstance();
```

The *'solveInstance'* instance method in effect invokes the target optimization system or software in order to obtain the solution to the current model. It returns an ORML string, which in case of a feasible model contains the solution details, or an infeasible return status in the case of an infeasible model.

3.3 Configuring Optimizers

The options package (illustrated in figure 3.2), provides functionality for controlling the behaviour of a given optimizer instance. The top-level class is '***OptimizerOptions'*** which can be accessed by invoking the *'getOptimizerOptions'* method on an optimizer instance. The options instance is largely a free format name-value pair store with the exception of its constituent '*ReportingOptions'* composite, which controls the level of reporting information. The reason for this is to enable vendors to define vendor specific options, which in effect extend the operation of optimizers beyond the scope defined by the framework. In essence, vendor specific options can be used to tailor the behaviour of an optimizer to leverage vendor specific strengths, for example, specifying a pre-processor for simplifying models prior to invoking a solution algorithm, or specifying the solver or solution algorithm to be used by the given optimizer.

The *'ReportingOptions'* instance can be obtained from the options instance by invoking the '*getReportingOptions'* method. This class in contrast to the '*OptimizerOptions'* is a fixed format structure, as it deals with the level of reporting information included in the ORML report. It essentially replicates the structure of an ORML solution report, providing boolean options for excluding or including specific attributes in the report. This ensures that it remains intuitive to developers, and reduces the possibility of incorrect configuration.

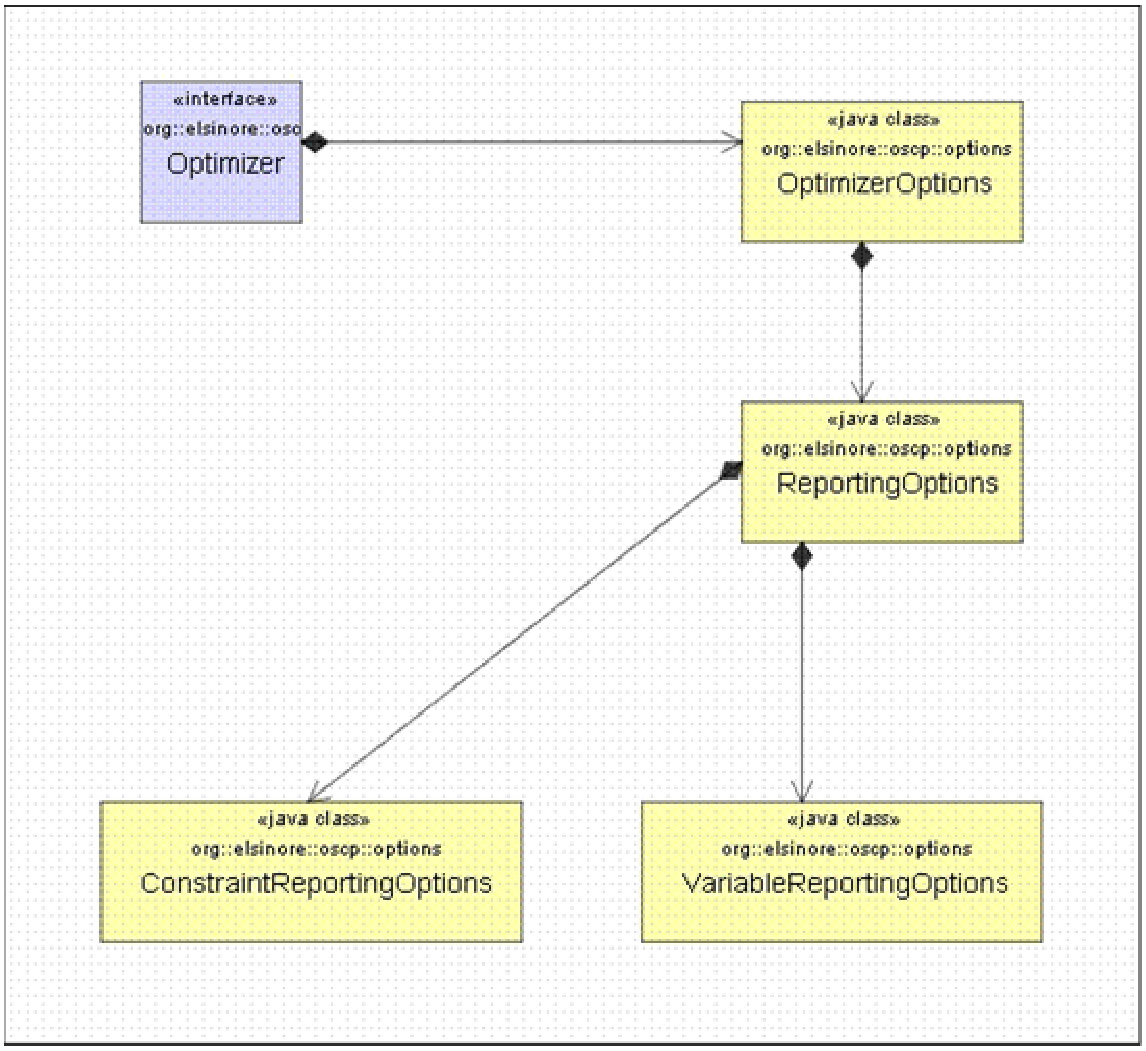

Figure 3.2: Options package class diagram

The *'ReportingOptions'* class is composed of two sub-types *'ConstraintReportingOptions'* and *'VariableReportingOptions'* which control the reporting of constraint and variables solution information respectively. These two types are summarised briefly below:

i. ***ConstraintReportingOptions:*** Encapsulates the reporting options for constraints, and can be accessed from the encapsulating instance through an accessor.
ii. ***VariableReportingOptions:*** Contains the reporting options for decision variables, and has to be obtained from the encapsulating instance through an accessor method.

By default, all the reporting options are set to a value of *'true'* which forces the inclusion of all possible attributes in the ORML solution report. Consequently, client applications have to explicitly turn-of or exclude unwanted material from the solution report.

3.4 Exception Classes

OSCP defines a number of exception classes which encapsulate, and can be used to report error conditions that can occur in the course of using a particular implementation. Vendors are at liberty to extend these, however vendor implementations are not allowed to propagate vendor or system specific exceptions. In simple terminology, the exceptions that can be thrown by any given implementation must match those defined in the OSCP interfaces or contract. This insulates upstream applications/clients from vendor specific exceptions. The OSCP defined exceptions and error classes are summarised as follows:

i. ***ElsinoreError:*** The top-level OSCP Error class. Implementations should avoid throwing this error except where necessary. Client applications should however always include a top level 'catch' block to deal with errors of this type. It denotes a critical and unrecoverable runtime error, for example if an error persistently occurs trying to invoke the underlying optimization engine.
ii. ***FactoryConfigurationError:*** A subclass of the ElsinoreError, which is usually thrown because of a system or factory mis-configuration that the application cannot correct.
iii. ***ElsinoreException:*** The top-level OSCP exception class. All vendor specific exceptions must be subclasses of this type.
iv. ***FactoryConfigurationException:*** A FactoryConfigurationException is thrown when the factory is configured incorrectly, but the erroneous configuration can be changed i.e. it denotes a recoverable error.
v. ***SyntaxException:*** An exception, which denotes an error in the model or data syntax, or structure. It includes a reference to an *ErrorsAndWarnings* object that provides additional details on the error(s) and related warning(s).
vi. ***ErrorsAndWarnings:*** An encapsulation of validation errors and warnings. Instances of this class are usually created and populated during the model or instance validation phases.

4 Future Work

The future work on the OSCP framework will focus on enhancing its usability and increasing the functionality provided by it or which can be accessed through it. The following list provides more details of the planned changes, and enhancements:

i. **Completing GAMS and AMPL Reference Implementations:** Currently, the GAMS and AMPL reference implementations are limited in the sense that they currently do not transform native result output to ORML format. This limitation is due to the fact that both modelling systems do not provide callable libraries or a well defined application interface, but rather have to be invoked as runtime processes and the system output parsed manually to obtain the solution or infeasibility details. Future work on OSCP will focus on implementing result parsers for both GAMS and AMPL with the aim of making both implementations fully compliant and functional.
ii. **Providing Implementations for AIMMS, LINGO and OPL Studio:** In order to establish itself as the de-facto standard for integrating optimization software into Java applications, it is necessary that the OSCP distribution include implementations for the vast majority of optimisation software currently in use. In accordance with this objective, future releases of the OSCP distribution will include implementations for the AIMMS, LINGO and OPL Studio modelling environments.
iii. **Enhancement of Interfaces:** For simplicity, current OSCP interfaces rely on the basic Java string type, however additional interfaces that utilise the native java XML types will be provided in future releases, particularly the 'javax.xml.transform.Source' type. This will have no impact on existing interfaces, as these will be maintained in their current state. In addition to support for java XML types, new interfaces will also be provided to leverage advances in optimization or software technology.

5 Conclusion

This paper has presented a high level overview of OSCP including its core interfaces. OSCP achieves the abstraction of optimization system interfaces by using the typing structure and syntax provided by AML and ORML. A full reference implementation of the OSCP interfaces is available for the MPL modelling system, and as such proves that the scheme advocated by OSCP is feasible and can be successfully implemented. Additional implementations are also available for the GAMS and AMPL modelling systems.

Further work on the framework will involve enhancements to the core functionality accessible through it, and a widening of the supported optimization platforms by providing fully functional implementations for additional modelling systems e.g. LINGO, and OPL Studio.

6 Bibliography

1. *Optimization Reporting Markup Language*, http://www.doc.ic.ac.uk/~oce/elsinore/orml.htm

2. Obi C. Ezechukwu and Istvan Maros. “*AML: Algebraic Markup Language*”, DoC Departmental Technical Reports 2003/12, Imperial College, London, 2003
3. Obi C. Ezechukwu and Istvan Maros. “*OOF: Open Optimization Framework*”, DoC Departmental Technical Reports 2003/7, Imperial College, London, 2003
4. Obi C. Ezechukwu and Istvan Maros. “*ORML: Optimization Reporting Markup Language*”, DoC Departmental Technical Reports 2003/13, Imperial College, London, 2003
5. Maydene Fisher, Jon Ellis, Jonathan Bruce. “*Jdbc Api Tutorial and Reference*”, Addison Wesley, 2003
6. *XML*, http://www.w3.org/XML/

7. *XSL*, http://www.w3.org/Style/XSL/