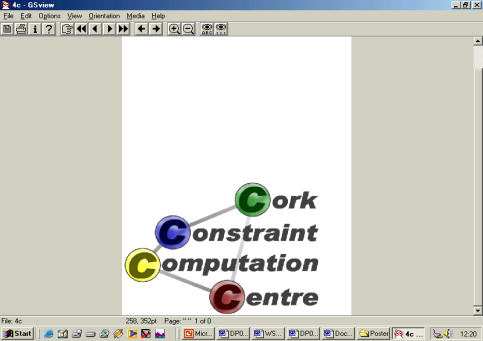
Cork CONsTRAINT COMPUTATION CENTRE



CDDC

Applying Integrated Rules and Constraint Technology to Diagnose and Resolve Capacity Deficiency Problems for Large Data Centres

Version 1.0

User Manual

September 28, 2010

Table of Contents

[Introduction 3](#_Toc272876742)

[Formal Problem Description 4](#_Toc272876743)

[Framework organization 6](#_Toc272876744)

[General Architecture 6](#_Toc272876745)

[Prototype Implementation 8](#_Toc272876746)

[The Data Feed 8](#_Toc272876747)

[The Business Rules 10](#_Toc272876748)

[The CSP Generator 10](#_Toc272876749)

[Results 11](#_Toc272876750)

[Installation Instructions 14](#_Toc272876751)

[Pre-requisites 14](#_Toc272876752)

[Installation Description 14](#_Toc272876753)

[Executing the Example 14](#_Toc272876754)

# Introduction

While the most data centers recognize the importance of capacity planning, the proper systems mainly monitor the overall performance and inform administrators about capacity deficiency problems. For example, the EMC data center in Cork monitors the performance of hundreds and thousands of their storage devises collecting resource utilization samples every 15 minutes. Their current system produces graphical utilization reports but then human administrators have to decide how to move load around or purchase and install additional equipment to avoid capacity deficiency problems. So, EMC is looking for a more sophisticated capacity optimization solution. We were also contacted by Vkernel Corporation, whose product “Capacity Bottlenecks Analyzer” can monitor performance of virtualization servers and effectively predict bottlenecks in VMware infrastructure. However, even this highly advanced product lacks optimization capabilities that may recommend the best way to resolve the bottlenecks on the consistent basis. Resolving capacity deficiency is an important research problem and its solution can benefit different data centers.

In this project, we developed an integrated business rules and constraint programming technique to diagnose and resolve capacity deficiency problems for the modern data centers with a large number of hardware resources including disks, CPU, memory, and networking devices. Our software solution allows data centers to define company-specific business rules that will help them to diagnose and predict an excessive use of their resources. Based on these, our constraint-based engine will recommend an optimal load reconfiguration solution that will allow a customer to better utilize their existing resources and proactively add new resources while minimizing the total maintenance cost.

# Formal Problem Description

In our model, data centres consist of a set of servers. Depending on the type of data centre, these servers can be used to provide computing power, data storage, a combination of these, or any other purpose. This is of no consequence to the model itself. Furthermore, there is a set of processes. At regular time intervals, a set of sensors provide information on the processes, e.g. their CPU load, the amount of memory required or energy consumption. A function provides the output of a sensor for a given time point and a given process, i.e. equals the value measured by sensor at time for the process . If a process is not active at a certain time point t (e.g. because it has finished), the output of is undefined, i.e. . By default, sensor 0 returns the server that the process is running on, i.e. iff runs on at time .



The total requirements placed on a server can be computed from the processes running on that server as follows:

We will use to denote the set of values of all sensors for a given server at a particular time. We let denote the set of all possible combinations of sensor readings, i.e. .

To identify which servers are of interest, we introduce a classification of possible labels (for example, this could be equal to . We assume there is a function that, given a set of sensor readings for the past time steps, can give the state a server is in. In order to optimise our solution, we introduce a cost function that, given a label, returns a virtual cost value for that particular label.

Through the classification of servers, we may identify that an unwanted situation has arisen. For example, we may find a number of servers that are classified as critical. To address this issue, we could move processes away from these servers onto others. Thus, a solution is a reallocation of processes that brings the data centre to an improved state. In terms of the model, we want to compute a set .

There are a number of constraints that should hold for a solution. Firstly, there is a maximum value that we want to satisfy for each sensor of a server . Secondly, some processes may never run together, i.e. be present on the same server. To this end, there is a function that given some process returns the list of all processes that cannot share the same resources with that process. Finally, some processes may need a particular set of servers. The function denotes the possible servers that a process can run on. (Thus, if there are no restrictions on .)

The last aspect we introduce is a distance metric between servers. The function returns the distance between two servers. This could simply be the physical distance, but is more likely to also take into account the network topology in the data centre. When moving processes, we want to minimise the total distance of the processes that are moved.

The full problem statement can now be formalised as follows.

Thus, we are interested in finding a new allocation of processes to servers for the next time step, i.e. computing for all , such that the allocation satisfies all constraints. The cost of this allocation is mainly influenced by the cost associated with the labelling of the servers.

# Framework organization

## General Architecture

The general system architecture is shown in Figure 1.

The main elements are:

Figure System Architecture

* **Data Feed** The system is flexible to different customer specific data feeds. The current implementation is based on standard Excel tables. The business object model defines the exact content of the performance monitoring data, including resource utilization data samples for different resource types over certain time periods.
* **Business Rules and Constraints Repository** The system communicates with a data center administrator (who can be a business analyst not necessarily a developer). The administrator specifies the business rules and constraints that are specific for the particular data center and describe different conditions and thresholds that diagnose an excessive use of the center resources. Examples of the rules and constraints are,
  + No disk should spend a certain amount of time above a 70% utilization for two consecutive time periods
  + The utilization of a resource should not be more than a certain percentage of the average utilization for a specified percentage of time.

The OpenRules Business Rules Management System (BRMS) is used to provide business analysts with a simple user interface to create and manage business rules and constraints.

* **Business Rule Engine and CSP Generator** This system provides a generic Business Rule Engine that is used to execute the rules from the Business Rules Repository against the performance data to produce the diagnostics about excessive use of the system resources. Additionally, the Rule Engine includes a CSP Generator. The main objective of this component is to dynamically generate a constraint satisfaction problem (CSP) that will compute a recommended course of action to remedy the overloaded system.
* **Constraint Solver** Two implementations are present, that differ in the choice of CP solver. One is built on the Intellify platform by ThinkSmart Technologies, the other is based on a reference implementation of JSR 331. Whichever implementation is used, the Constraint Solver will solve the CSP and generate recommendations for an optimal workload reconfiguration that will allow a customer to better utilize their existing resources and proactively add new resources while minimizing the total maintenance cost.

In the next section, we will describe the various parts in more detail, using the prototype implementation that is part of the distribution (see page 14 for instructions on obtaining and installing the system).

# Prototype Implementation

The prototype implementation focuses on the management of storage servers. Such servers do not run any applications, yet are dedicated to the storage and retrieval of information. Information is stored on virtual disks (“volumes”) that may span many physical disks, even across different servers. Each volume has a list of applications that are using that particular volume, and a list of I/O operations per application. Different volumes may occupy the same physical disk. Thus, the total amount of data that is being transferred to/from a disk is determined by the activity of the different volumes on it. If several volumes with high transfer rates are stored on the same disk, the disk is overloaded and response times drop. In such a case, we want to redistribute the volumes over the active disks in such a way as to alleviate any bottle necks. Moving a volume is a costly operation, however, and so we want to minimise the number of moves over time.

## The Data Feed

The data feed is used to describe the problem. It consists of the entities that are present (in this case for example disks, volumes (organized in logical and meta devices) and applications), their attributes and their performance characteristics. Since the data feeds for different environments are not the same, the contents of the data feed need to be described by a developer. For this reason, the data are presented to the system in the form of a rule, where the logical part of the rule allows the data to be described.

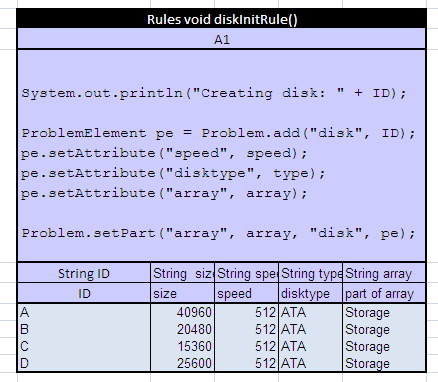
For example, consider the partial data feed depicted in . This describes four disks that are part of a disk array. As can be seen from the bottom rows of the table, each disk is characterised by an identifier, its size, speed and disk type, and the name of the array it is part of is given.

Figure : Example data feed description

The top half of the table describes the data feed by making the appropriate calls to the global Problem object. This object stores the relationships between the various parts of the data feed as ProblemElement objects. In this case, the following three types of functions are used:

* Problem.add(String type, String name) This function creates and returns a new ProblemElement of the given type and with the given name.
* ProblemElement.setAttribute(String name, String value) This function stores a given (attribute, value) pair with the element.
* Problem.setPart(String ownerType, String ownerName, String partType, ProblemElement part) This function is used to describe the part hierarchies in the system. In this case, it records that the disk we have just created is part of the array by the given name.

The purpose of describing the data feed is to allow for flexibility. The rules system and the CSP generator will use this information to do their tasks. So far, we have seen how we can create a new element, store its attributes and tell the system that this element is part of another element. Another important aspect are the sensor readings we have for the various parts. The following is a list of functions that can be used for this purpose:

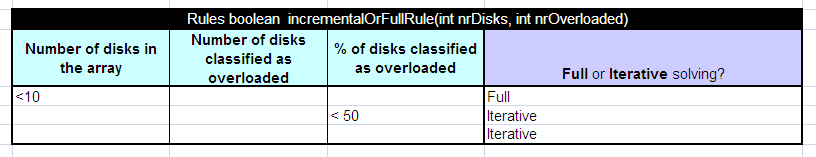
* ProblemElement.setSensorHistory(String sensorName, int[] history) adds a list of sensor readings to a problem element. In our example, the sensor readings are for applications, and so you may find an example of this function in the relevant rule
* ProblemElement.setSensorFrequency(String sensorName, int frequency) This stores the time between sensor readings; it can be used by the rules to determine whether a particular series of readings does or does not exceed the threshold values.

Finally, certain sensor values may need to be propagated up the hierarchy; for example, if each application has a sensor value for its current load, the total load for a disk equals the sum of all the applications that are on it. For this purpose, we can associate a number of *accumulators* with each ProblemElement. Each accumulator is identified by a name, and the following function is used to describe these:

* Problem.accumulate(String ownerType, String ownerName, String accumulatorName, Double value) This function should be called directly after calling Problem.setPart(…); it increases the named accumulator of the owner by the given amount. (Several subsequent calls can be used to update a number of accumulators.)

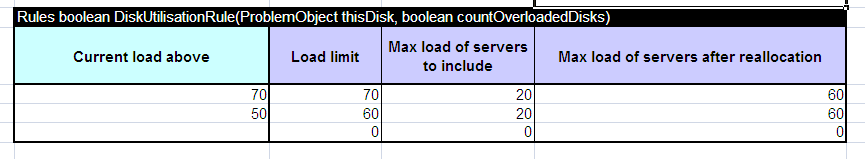
## The Business Rules

In the prototype implementation, we consider two different ways of solving the problem. In the first approach, we diagnose all the disks that are present, and then build a single optimization model to reallocate the workload over the disks. However, for large problems, it can take quite some time to compute the optimal recommendation. Therefore, we also introduced a second approach. Here, we reallocate the workload from each overloaded disk as it is encountered. While this is faster, we also expect this to be less optimal, as we make decisions that are only locally optimal.

For this reason, there are two rules involved in performing the diagnosis and building the CSP. The first is pictured in the figure below. It specifies when we want to solve the model an an iterative manner, and when we want to consider the full problem as a whole. Only two rules (and a default) are shown here, but the rules can be extended, of course.

The code for this rule either calls the Optimiser.solveFully() method, or Optimiser.solveIncremental(), depending on the specified strategy.

The second rule specifies the actual diagnosis. For simplicity, we only deal with the current load factor here, but in practice, this should reflect the history and the sampling frequency as well. The code for this rule is explained in the next part, where we discuss the CSP generator.



## The CSP Generator

Having defined the structure of the problem with the data feed, we can generate a CSP from the rules by specifying which elements can be reallocated. This is done through a global Optimiser object, which includes the following method: Optimiser.add( ProblemElement owner, String movableType). This specifies that the given element should be added to the problem, and that its parts of type movableType can be reallocated between other elements of the same type. At this point, we can also impose maximum values for the different accumulators through the following method: Optimiser.addLimit( ProblemElement owner, String accumulatorName, double limit).

The method Optimiser.next() indicates that we are about to solve a new problem. Optimiser.solve() is used to compute a solution. Calls to next() and solve() are silently ignored when the optimizer is set to compute global solutions. In that case, a Optimiser.SolveFull() is required after all disks have been assessed to compute the solution.

## Results

As a test, we generated a benchmark set consisting of 100 problem instances that were randomly generated from actual data. The problems have the following characteristics: we consider a storage array with disks, each of which is represented by a server in our model. The array is used by applications, where each application requires space on between 1 and 3 disks (we say that the virtual disk for application requires between 1 and 3 volumes). If application requires, say, space on 2 disks, this gives rise to 2 processes in our model, and , that we have to accommodate. For security reasons, these cannot be allocated on the same disk (as they are part of the same RAID device). Therefore, an constraint is imposed over such sets of processes. Finally, only 4 applications can use a disk at the same time due to space restrictions. To reiterate, the disks in the storage array are represented by the servers in our formal model; the volumes on those disks are represented by the processes running on the servers. We assume unit distance between all disks, in effect minimising the total number of changes we make to the allocation of applications to disks.

The values of the load sensors were chosen such that either or of the disks is considered to have a high or very high load, and the goal of the system is to achieve a maximum load of 60% for any given disk.

Figures and show the results of our experiments, taking the average of the 10 instances for each number of disks tested. Notice that the figures represent both the CPU time required to solve the instance, as well as the quality of the resulting solution (as measured in the number of moves).



Figure Results with 12.5% of disks overloaded



Figure Results with 25% of disks overloaded

The two figures show a similar pattern: when the number of disks increases, solving the whole problem as one quickly leads to performance issues, and we cannot prove that we have found the optimal solution within 15 minutes. On the other hand, when we deal with each overloaded disk in isolation (and finding an optimal result during each iteration), it takes less than 10 seconds for the largest of problems. Thus, this part of our hypothesis is confirmed by these experiments.

The other part of our hypothesis stated that we expect to find that by solving iteratively, we produce lower quality solutions, i.e. we require more changes to achieve the desired load levels. Again, the figures confirm this hypothesis.

A closer examination of the results, shows exactly why this happens. When we look at the overloaded disks in isolation, the only way to deal with the issue of one of the disks being overloaded is to move at least 1 application away from the disk, and replace it with another volume (either belonging to some other application with a lighter load, or even an unused volume from another disk). Thus, 2 changes are required to deal with the issue. For example, consider 3 disks, , , and , each with two applications on them with loads: , and . Suppose we want to achieve a maximum load of 60, and consider disk before disk . The local problem generated to deal with consists of disks and (is not considered, since it's overloaded). It is straight-forward to see that we can achieve the objective by swapping the application with load 25 with one of the applications from disk (totaling 2 moves). The same holds when we consider , and we achieve a total of 4 moves. Of course, in general, we may undo some of the moves we have made during an earlier iteration in order to satisfy the load on other disks, so we expect to see somewhere just under changes given that there are overloaded disks. (This number is confirmed by the graphs above.)

On the other hand, when we consider the solutions found when solving the whole problem at once, we observe that often, load is transferred between overloaded disks as well. In our previous example, for example, we can achieve a solution with only 3 moves: Move the application with load 40 from to , move the one with load 25 from to and move the one with load 10 from to . This requires that load is transferred between overloaded disks. Having this global view helps in detecting such situations, which explains the difference between the two approaches.

When deciding between local or global solution strategies, it helps to keep these results in mind.

# Installation Instructions

## 

## Pre-requisites

To work with the CDDC system you have to pre-install the standard Java software from, which can be obtained from http://java.sun.com/j2se/. Any version greater than 1.6 will do.

Below are the installation instructions for Java for Windows XP: recommendations: Install Windows Platform - J2SE Development Kit 6u1 (or later) from <http://java.sun.com/javase/downloads/index.jsp>.   Accept all defaults, and your JDK will be installed at C:\Program Files\Java\jdk1.6.0\_01 (or a similar location when a version different than 1.6.0.01 is installed).

Now you want to set up your environment variables in a way similar to the one below (but adjusting for the right location):

set JAVA\_HOME=C:\Program Files\Java\jdk1.6.0\_01

To set up environment variables, left-click on the Windows Start button, then right-click on My Computer, and chose Properties.  Select the tab "Advanced" and then press "Environment Variables".  Add or correct the User variable JAVA\_HOME accordingly.

Finally, click on Start + Run and enter "cmd" to open a DOS window.  Type

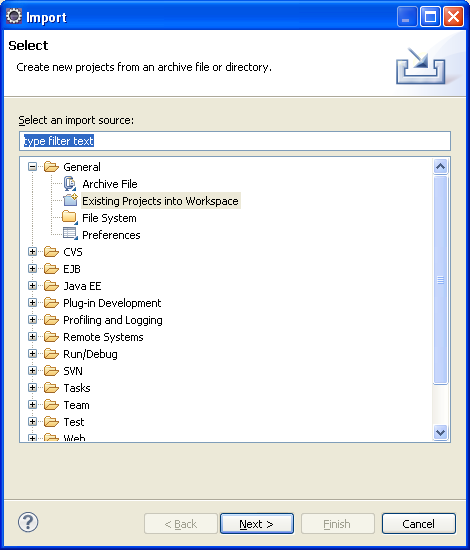
> java –version

to make sure that you use the correct version of Java.

## 

## Installation Description

The software can be installed on its own, or directly imported into the Eclipse Java development environment. This is the recommended setup; Eclipse can be obtained from http://www.eclipse.org/.

* **Stand alone installation** The software is distributed as a single .zip file, which can be extracted using the appropriate tools. In Windows XP, double-click on the CDDC.zip file, and choose “Extract all files” from the menu on the left hand side. For details on how to extract the files on other operating systems, please consult the operating system manual.
* **Eclipse** The recommended approach is to import the software into the Eclipse IDE as follows. From the File menu, choose “Import”, and select from the “General” group “Existing projects into workspace”, before clicking “Next”. In the dialog that follows, choose “Select archive file”, then “Browse” and select the CDDC.zip file. Click “Finish” to start the import. See also the following figures:

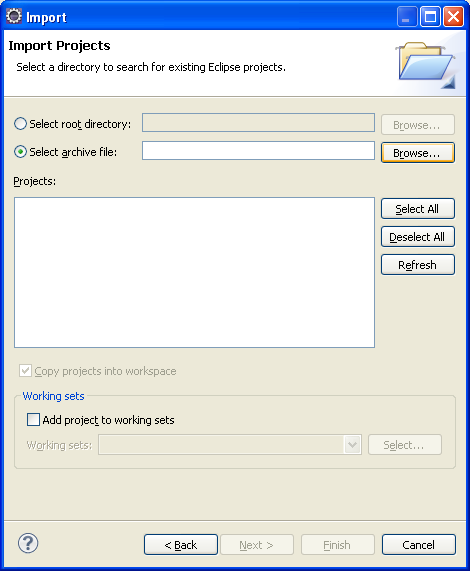


Figure Importing a project into Eclipse

## Contents

After installing, you will find two projects installed. Each of these contains one or more Java packages, and the associated files that are required. (In a manual, stand-alone installation, there will be two directories by the same names.)

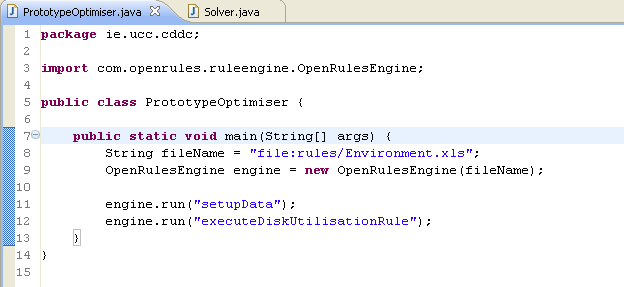
* **CDDC Prototype**

**ie.ucc.cddc** This package is the main prototype, based on the JSR 331 reference implementation.

* **CP Experiments**

**ie.ucc.cddc.CPexperiments.\*** These packages contain the solver and the problem generator for the experiments discussed in An Integrated Business Rules and Constraints Approach to Data Centre Capacity Management, R.P.J. van der Krogt, J. Feldman, J. Little and D. Stynes. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, 2010. It requires the Intellify optimisation library of ThinkSmart Technologies (not included).

## Executing the Prototype Example

In this section we assume the software is imported into Eclipse. In the ie.ucc.cddc package, which is part of the cddc project, find the PrototypeOptimiser class. This contains the main() method:

This sets up the environment, and then runs two rules:

Figure Main method of the prototype

1. setupData This rule loads the data; followed by
2. executeDiskUtilisationRule This rule diagnoses the disks and runs the optimizer if necessary.

These rules are located in the genericRules.xls file, which can be found in the same package.

To run the prototype demo, simply open the PrototypeOptimiser.java file, and right-click anywhere on the Java code. From the menu that opens, choose “Run as”, and then “Java application”, as shown in .

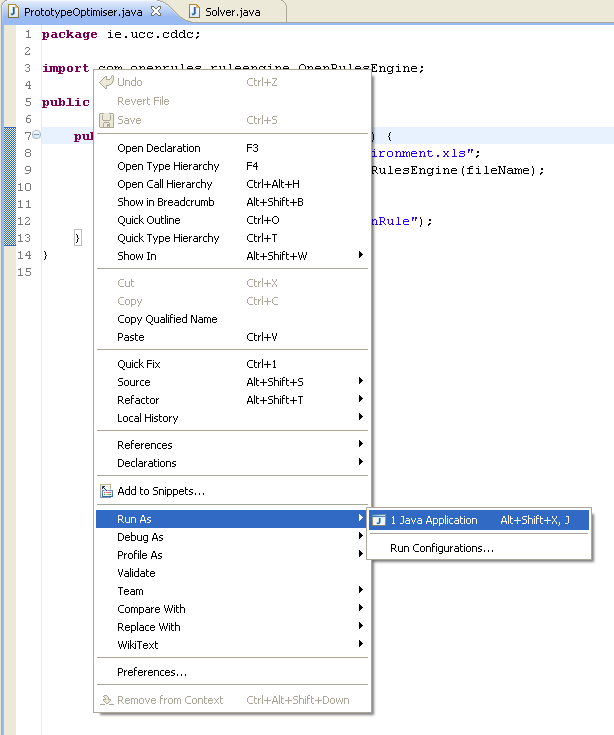


Figure How to run the prototype