BMC Software Inc.

# A Multi-Threaded Software Program for Populating Structured Configuration Management Database (CMDB) Data

Author:

Henry Liu

Posted: March 2009

## Overview

Today's organizations depend on complex IT infrastructure and business services to run their daily operations efficiently. Both the IT infrastructure and business services of an organization can be managed with software using a CMDB or a CMS as a central repository for the data known as Configuration Items (CIs). CIs represent the IT asset and services data as well as their inter-relationships. The idea of *A Multi-Threaded Software Program for Populating Structured CMDB Data* can help solve the problem of populating data into the CMDB for performance and scalability test purposes with large volume of CIs and certain data structures or hierarchies. It can also help vendors and customers size the hardware capacity required for actual production deployments of CMDB-based software. It is also an effective method for populating CMDB data for reproducing and trouble-shooting CMDB-related performance and scalability escalations in customer's environment as well. In this technical disclosure bulletin, we describe how such a software program has been developed and in use at BMC.

## Background

All CMDB-based software products provide Application Programming Interfaces (APIs) for performing various tasks of creating, querying, updating, and managing CMDB data. Such APIs are used by CMDB-based software applications to discover and manage IT asset data in the form of CIs (Configuration Items). Having a software program that can be used for generating simulated CIs in a CMDB in large volumes would help not only a CMDB software vendor but also an application software vendor in testing the performance and scalability of CMDB APIs and CMDB-based applications.

## Solution

The proposed implementation is characterized by the following features:

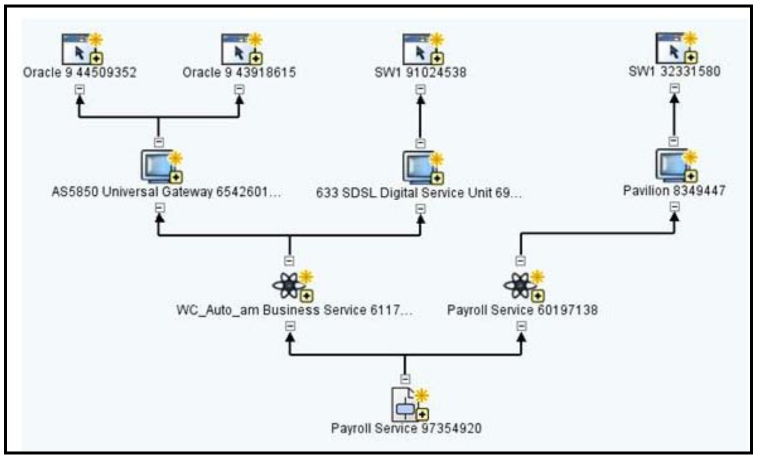
- A multi-threading programming model which is the underlying mechanism for generating CMDB data in large volumes concurrently.

- A CMDB data model that is confined by certain CI-relationship hierarchical structure. This data model is specified in a text file as input to the program.
- An indexing scheme which tracks two indices: one is the index of the CI to be created and the other is the index of its parent CI. The index of a CI to be created is sequential, whereas the parent index of a CI is determined by both the index and the level of the CI to be created.

In addition, the implementation of such a software program should provide sufficient convenience for users to get up to speed quickly in learning how to use it. The said convenience can be provided by using a few external input text files to define the CMDB data model, the test environment, and a product catalog that might be part of a CMDB data. The program output should be directed to a text file that summarizes the total number of CIs created, the duration of the run, and the throughput in terms of CIs / second averaged over all the threads.

## Drawings

The following figure shows an example CMDB data model that this solution is capable of producing.  It consists of 4 business applications, three computer servers, 2 business services, which all belong to one business service.



Note that this screenshot is for demo-purpose only. The program can actually generate a lot more such instances at each level.

## Implementation

The following pseudo-code is provided to illustrate how the algorithm for generating the hierarchical CMDB data structure can be implemented.

**Step 1:**

Given {namespace, classId, relationship, cardinality}, construct an array of dimension *cardinality* that will hold the instanceIDs of the parent objects to be associated with their child objects later. This can be accomplished with a pair of setter / getter classes.

**Step 2:**

```
// build the model which is a 2-dimensional array for holding
// the definition for each instance

Read the model definition file and extract the info {classId,
cardinality, level, relationship} from each line;

If (topLevel) {
      setTopLevel;
} else {
      // process range
      if (cardinality is a range) {
            calculate rangeStart;
            calculate rangeEnd;
            calculate rangeStartIndex;
            calculate range;
      } else {
            set range = 0;
}
// process fractional
if (cardinality is fractional) {
      extract part1 of the cardinality;
      extract part2 of the cardinality;
      calculate cardinality;
      set isFractional to true;
}
determine how many instances to create at this level;
determine the start index for this level;

loop (for each instance at the current level) {
if (first level) {
      set parentIndex to 0;
} else {
calculate parentIndex based on the startIndex
and number of instances of the parent level;
}
assign {classId, parentIndex, relationship} to the first three
elements of the attributes of the current instance;
calculate the inclusivity and assign it to the fourth attribute of
the current instance;
set the index for the next instance;
// end loop

if (end of the model definition file) exit;
```