Hardware Inventory Visualization

Technical Documentation

Technical Documentation Team

May 21, 2025

Contents

1 Introduction

This document provides a comprehensive technical overview of the Hardware Inventory Visualisation system. The system processes CSV files containing hardware inventory data and generates interactive SVG visualisations that allow users to explore the relationships between hardware chassis, LPARs (Logical Partitions), and applications.

The system consists of two main components:

1. A CSV parser that generates PlantUML and C4 diagram files
2. An SVG generator that creates interactive diagrams with drill-down capabilities

This documentation details the algorithms, data structures, and techniques used in the implementation, enabling developers to recreate the system or extend its functionality.

2 System Architecture

The overall architecture follows a two-stage pipeline:

1. **Stage 1:** CSV files are parsed and transformed into PlantUML and/or C4 diagram files
2. **Stage 2:** PlantUML files are converted into interactive SVG diagrams

Each stage is implemented as a separate Python script to maintain modularity and allow for independent usage or extension.

2.1 Data Flow

The data flow through the system proceeds as follows:

1. Input CSV files containing hardware inventory data are read
2. The data is parsed and organised into a hierarchical structure
3. PlantUML and/or C4 diagram files are generated from the hierarchical structure
4. PlantUML files are converted to SVG using the PlantUML jarPython
5. The SVG is enhanced with JavaScript to add interactivity
6. The interactive SVG is output as the final product

This pipeline architecture allows for easy extension or modification of individual components without affecting the rest of the system.

3 CSV Parser Implementation

The CSV parser component is responsible for reading the input CSV files, parsing the data, and generating PlantUML and C4 diagram files. This section details the implementation of this component.

3.1 Data Structures

The parser uses the following primary data structures:

1. **Systems Dictionary**: A dictionary mapping system names to system information, including LPARs and resource totals
2. **LPARs Dictionary**: A dictionary mapping LPAR names to LPAR information, including applications and resource allocations
3. **Applications Dictionary**: A dictionary mapping application IDs to application information

The hierarchical relationship between these structures mirrors the real-world relationship between systems, LPARs, and applications.

3.2 CSV Reading Algorithm

The CSV files are read using Python's built-in `csv` module. The algorithm for reading the BOII2 CSV file is as follows:

The algorithm for reading the Sample BOI CSV file is similar:

3.3 Application Matching Algorithm

A critical part of the implementation is matching applications to LPARs based on name similarity. The algorithm for this is as follows:

This matching algorithm uses both exact and partial matching to maximise the connection between applications and LPARs despite potential naming inconsistencies between the CSV files.

Algorithm 1 BOII2 CSV Reading Algorithm

```
1: procedure LoadBOII2CSV($filename$)
2:     Open $filename$ for reading
3:     Create CSV reader with header row
4:     for each $row$ in CSV do
5:         Extract $system\_name$ from ”Managed System Name” column
6:         Extract $system\_serial$ from ”Managed System Serial” column
7:         if $system\_name$ not in $systems$ then
8:             $systems[system\_name] \leftarrow$ new system with $system\_serial$
9:         end if
10:        Extract $lpar\_name$ from ”POR - Virtual Name - use this ONE” column (or alternatives)
11:        Extract $lpar\_cpu$ from ”LPAR CPU” column
12:        Extract $lpar\_memory$ from ”LPAR MEM” column
13:        Create new LPAR with extracted information
14:        Add LPAR to $systems[system\_name].lpars$
15:        Add LPAR to $lpars$ dictionary
16:        Update $systems[system\_name]$ resource totals
17:    end for
18: end procedure
```

Algorithm 2 Sample BOI CSV Reading Algorithm

```
1: procedure LoadSampleBOICSV($filename$)
2:     Open $filename$ for reading
3:     Create CSV reader with header row
4:     Initialize $computer\_apps$ dictionary
5:     for each $row$ in CSV do
6:         Extract $computer\_name$ from ”Computer Name” column
7:         Create application with information from row
8:         Add application to $computer\_apps[computer\_name]$
9:         Add application to $apps$ dictionary
10:    end for
11:    Match applications to LPARs using name similarity
12: end procedure
```

Algorithm 3 Application Matching Algorithm

```
 1: procedure MatchAppsToLPARs($computer\_apps$)
 2:     for each $(computer\_name, apps)$ in $computer\_apps$ do
 3:         $matched \leftarrow False$
 4:         Convert $computer\_name$ to lowercase
 5:         if $computer\_name$ in $lpars$ (exact match) then
 6:             Add $apps$ to $lpars[computer\_name].applications$
 7:             $matched \leftarrow True$
 8:         else
 9:             for each $(lpar\_name, lpar)$ in $lpars$ do
10:                 Convert $lpar\_name$ to lowercase
11:                 if $lpar\_name$ contains $computer\_name$ OR $computer\_name$ contains $lpar\_name$ then
12:                     Add $apps$ to $lpar.applications$
13:                     $matched \leftarrow True$
14:                     break
15:                 end if
16:             end for
17:         end if
18:         if not $matched$ then
19:             Add $computer\_name$ to unmatched computers list
20:         end if
21:     end for
22: end procedure
```

3.4 PlantUML Generation Algorithm

The PlantUML generation algorithm converts the hierarchical data structure into a textual representation following PlantUML syntax:

Algorithm 4 PlantUML Generation Algorithm

```
1: procedure GeneratePlantUML($output\_file$)
2:     Open $output\_file$ for writing
3:     Write PlantUML header and styling
4:     for each $(system\_name, system)$ in $systems$ do
5:         Write system rectangle with metadata
6:         for each $lpar\_name$ in $system.lpars$ do
7:             Retrieve $lpar$ from $lpars$ dictionary
8:             Write LPAR rectangle with metadata
9:             Group applications by type
10:            for each $(app\_type, apps)$ in grouped applications do
11:                Write package block for app type
12:                for each $app$ in $apps$ do
13:                    Write component for app with metadata
14:                end for
15:                Close package block
16:            end for
17:            Close LPAR rectangle
18:        end for
19:        Close system rectangle
20:    end for
21:    Write PlantUML footer
22: end procedure
```

The algorithm produces a hierarchically structured PlantUML diagram that reflects the relationships between systems, LPARs, and applications.

3.5 C4 Generation Algorithm

The C4 generation algorithm follows a similar pattern but uses C4 model syntax:

This algorithm produces a C4 model diagram that represents the system architecture using the C4 model's standardised notation.

4 SVG Generator Implementation

The SVG generator component converts PlantUML files into interactive SVG diagrams. This section details the implementation of this component.

Algorithm 5 C4 Generation Algorithm

```
procedure GenerateC4($output\_file$)
    Open $output\_file$ for writing
    Write C4 header and include required libraries
    for each $(system\_name, system)$ in $systems$ do
        Write System_Boundary for system with metadata
        for each $lpar\_name$ in $system.lpars$ do
            Retrieve $lpar$ from $lpars$ dictionary
            Write Container for LPAR with metadata
            Count applications by type
            for each $(app\_type, count)$ in application counts do
                Write Component for application group
                Write Rel between LPAR and application group
            end for
        end for
        Close System_Boundary
    end for
    Write legend and footer
end procedure
```

4.1 PlantUML to SVG Conversion

The conversion from PlantUML to SVG is performed using the PlantUML jar:

Algorithm 6 PlantUML to SVG Conversion Algorithm

```
procedure GenerateSVG($input\_file, output\_file$)
    Ensure PlantUML jar is available (download if needed)
    Execute command: java -jar plantuml.jar -tsvg $input\_file$ -o $output\_dir$
    Check if output file exists
    Return path to generated SVG
end procedure
```

4.2 PlantUML Hierarchy Extraction

To make the SVG interactive, the hierarchy from the PlantUML file must be extracted:

This algorithm parses the PlantUML file line by line, extracting the hierarchical structure of objects (rectangles, components, and packages) and their relationships.

4.3 SVG Enhancement Algorithm

The SVG enhancement algorithm adds JavaScript and CSS to the SVG to make it interactive:

Algorithm 7 PlantUML Hierarchy Extraction Algorithm

```
 1: procedure ParsePlantUML($plantuml\_file$)
 2:     Initialize empty hierarchy dictionary
 3:     Initialize empty current path stack
 4:     for each $line$ in $plantuml\_file$ do
 5:         if $line$ matches rectangle definition pattern then
 6:             Extract label and ID
 7:             Extract metadata and object type
 8:             Create object entry
 9:             if current path is not empty then
10:                 Add object as child to current parent
11:             else
12:                 Add object as root in hierarchy
13:             end if
14:             if object has children (ends with {) then
15:                 Push object ID to current path
16:             end if
17:         else if $line$ matches component definition pattern then
18:             Extract label and ID
19:             Create component entry
20:             if current path is not empty then
21:                 Add component as child to current parent
22:             else
23:                 Add component as root in hierarchy
24:             end if
25:         else if $line$ matches package definition pattern then
26:             Extract label and ID
27:             Create package entry
28:             if current path is not empty then
29:                 Add package as child to current parent
30:             else
31:                 Add package as root in hierarchy
32:             end if
33:             if package has children (ends with {) then
34:                 Push package ID to current path
35:             end if
36:         else if $line$ is closing brace then
37:             Pop from current path
38:         end if
39:     end for
40:     Return hierarchy
41: end procedure
```

Algorithm 8 SVG Enhancement Algorithm

```
procedure MakeInteractive($svg\_file, plantuml\_file, output\_file$)
    Extract hierarchy from PlantUML file
    Parse SVG file as XML
    Find all group elements in SVG
    for each $group$ in groups do
        Extract group ID
        Find title element with PlantUML ID
        if matching element found in hierarchy then
            Find rect or polygon element
            Extract position and dimensions
            Store element info
            Add onclick handler to group
            Add hover styling to rect/polygon
        end if
    end for
    Add JavaScript with element info and interactive functions
    Add CSS for styling interactive elements
    Add HTML panel for displaying details
    Save modified SVG
end procedure
```

The SVG enhancement adds the following interactive features:

1. Click handlers on elements to show details
2. Hover effects to indicate clickable elements
3. A details panel that displays information about the selected element
4. Drill-down navigation for exploring the hierarchy
5. Breadcrumb navigation for returning to higher levels

4.4 JavaScript Implementation

The JavaScript implementation adds the following functions to the SVG:

1. **showDetails(id)**: Displays details for an element
2. **hideDetails()**: Hides the details panel
3. **drillDown(id)**: Navigates to a child element
4. **navigateTo(index)**: Navigates to a specific level in the path

The JavaScript code also includes the hierarchy data extracted from the PlantUML file, enabling the interactive features to work without additional server requests.

5 Command-Line Interface

Both components include command-line interfaces to facilitate usage and integration with other tools.

5.1 CSV Parser Command-Line Interface

The CSV parser's command-line interface accepts the following arguments:

- **–input-dir**: Directory containing the input CSV files
- **–output-dir**: Directory where the output files will be written
- **–format**: Output format (plantuml, c4, or both)

5.2 SVG Generator Command-Line Interface

The SVG generator's command-line interface accepts the following arguments:

- **–input**: Input PlantUML file or directory
- **–output**: Output SVG file path
- **–plantuml-jar**: Path to plantuml.jar (optional)
- **–temp-dir**: Directory for temporary files (optional)
- **–pattern**: File pattern to match when input is a directory

6 File Handling

Both components include robust file handling to ensure reliability across different environments.

6.1 Input Validation

Input files and directories are validated to ensure they exist and are accessible:

Algorithm 9 Input Validation Algorithm

```
1: procedure ValidateInput($input\_path, is\_directory$)
2:     if $is\_directory$ then
3:         if not path exists or not is directory then
4:             Report error and exit
5:         end if
6:     else
7:         if not path exists or not is file then
8:             Report error and exit
9:         end if
10:    end if
11: end procedure
```

6.2 File Path Handling

File paths are handled using Python's `pathlib` module to ensure cross-platform compatibility:

Algorithm 10 File Path Handling Algorithm

```
1: procedure HandleFilePath($path, is\_output$)
2:     Convert $path$ to Path object
3:     if $is\_output$ then
4:         Create parent directories if they don't exist
5:     end if
6:     Return Path object
7: end procedure
```

6.3 Special Characters and Spaces

File paths with special characters and spaces are handled correctly using proper quoting and escaping:

Algorithm 11 Special Character Handling Algorithm

```
1: procedure HandleSpecialCharacters($path$)
2:     Convert $path$ to string
3:     If necessary, quote the path
4:     Return processed path
5: end procedure
```

7 Implementation Considerations

This section discusses important considerations in the implementation of the system.

7.1 Performance Considerations

The system's performance is affected by the following factors:

1. **CSV Size**: Larger CSV files require more memory and processing time
2. **Number of Systems/LPARs/Applications**: More entities result in larger diagrams and SVGs
3. **PlantUML Rendering**: The PlantUML jar's performance affects the SVG generation time
4. **SVG Size**: Larger SVGs may be slower to load and interact with in browsers

To address these concerns, the implementation includes:

1. **Efficient Data Structures**: Using dictionaries for O(1) lookups
2. **Minimised Duplicate Processing**: Avoiding redundant operations
3. **Incremental Parsing**: Processing files line by line rather than loading entirely into memory
4. **Optimised SVG**: Minimising unnecessary SVG elements

7.2 Error Handling

The implementation includes comprehensive error handling:

1. **File I/O Errors**: Handling missing or inaccessible files
2. **CSV Parsing Errors**: Handling malformed CSV data
3. **PlantUML Execution Errors**: Handling PlantUML jar issues
4. **SVG Generation Errors**: Handling failures in SVG enhancement

Errors are reported with clear messages to facilitate troubleshooting.

7.3 Cross-Platform Compatibility

The implementation ensures cross-platform compatibility:

1. **Path Handling**: Using `pathlib` for platform-independent path manipulation
2. **File Operations**: Using platform-agnostic file operations
3. **External Process Execution**: Properly handling shell commands across platforms
4. **Temporary Files**: Using the `tempfile` module for platform-appropriate temporary directories

8 Algorithm Complexity Analysis

This section analyses the time and space complexity of the key algorithms.

8.1 CSV Parsing

- **Time Complexity**: O(n), where n is the number of rows in the CSV files
- **Space Complexity**: O(n), where n is the total number of entities (systems, LPARs, applications)

8.2 Application Matching

- **Time Complexity**: O(c * l), where c is the number of computers and l is the number of LPARs
- **Space Complexity**: O(c), additional space for tracking unmatched computers

8.3 PlantUML Generation

- **Time Complexity**: O(s + l + a), where s is the number of systems, l is the number of LPARs, and a is the number of applications
- **Space Complexity**: O(s + l + a), for the generated PlantUML text

8.4 PlantUML Hierarchy Extraction

- **Time Complexity**: O(n), where n is the number of lines in the PlantUML file
- **Space Complexity**: O(d), where d is the depth of the hierarchy (for the current path stack) plus O(e) where e is the number of entities in the hierarchy

8.5 SVG Enhancement

- **Time Complexity**: O(g), where g is the number of group elements in the SVG
- **Space Complexity**: O(g), for storing element information

9 Testing Strategy

A comprehensive testing strategy should include:

1. **Unit Tests**: Testing individual functions and methods
2. **Integration Tests**: Testing the interaction between components
3. **End-to-End Tests**: Testing the complete pipeline
4. **Edge Case Tests**: Testing unusual or extreme inputs

Key test cases should include:

1. Empty CSV files
2. CSV files with missing columns
3. Large CSV files
4. CSV files with special characters
5. Unmatched applications
6. Systems with no LPARs
7. LPARs with no applications

10 Extension Possibilities

The system can be extended in various ways:

1. **Additional Diagram Types**: Supporting other diagram formats
2. **More Interactive Features**: Adding filtering, searching, or comparative views
3. **Enhanced Visualisation**: Adding charts, graphs, or heat maps
4. **Integration with Other Tools**: Adding support for exporting to other tools
5. **Real-Time Updates**: Supporting live data updates

11 Conclusion

This document has provided a detailed technical overview of the Hardware Inventory Visualisation system, including algorithms, data structures, and implementation considerations. With this information, developers should be able to understand, recreate, and extend the system as needed.

The modular design, with separate components for CSV parsing and SVG generation, facilitates maintenance and extension. The careful attention to file handling, error handling, and cross-platform compatibility ensures reliability across different environments.

The interactive SVG output provides a valuable tool for exploring hardware inventory data, enabling users to understand the relationships between systems, LPARs, and applications in a intuitive way.