RC23699 (W0508-105) August 19, 2005
Computer Science

IBM Research Report

Managing System Capabilities and Requirements Using Rough Set Theory

Larisa Shwartz, Naga Ayachitula, Surendra Maheswaran, Genady Grabarnik
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Managing System Capabilities and Requirements using the Rough Set Theory

Larisa Shwartz
IBM Thomas J Watson Research Center,
Hawthorne, NY 10532
lshwart@us.ibm.com

Naga Ayachitula
IBM Thomas J Watson Research Center,
Hawthorne, NY 10532
nagaaka@us.ibm.com

Surendra Maheswaran
IBM Thomas J Watson Research Center,
Hawthorne, NY 10532
suren@us.ibm.com

Genady Grabarnik
IBM Thomas J Watson Research Center,
Hawthorne, NY 10532
genady@us.ibm.com

As the size and complexity of software systems increases, the reuse of independent pieces of software combined in different ways to implement complex software systems has become a widely accepted practice. . The operating system and middleware environments provide some support for representing the dependencies among software components in an explicit way. This representation can be manipulated in order to automate the deployment and configuration of new software applications. The variety of dependencies could be very large, while the complexity of inter-dependencies very high.

This paper discusses this issue in terms of Rough Set Theory. We focus on software requirements, defined as dependencies and their relative structure.

In this paper we introduce a procedure, a formalization model and an algorithm for eliminating conflicting and redundant requirements. We also define a minimal topology for large distributed applications.

Keywords-requirements; dependencies; system capabilities, rough set theory; formalization model.

I. INTRODUCTION

Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level). If they are, we call it a component and reuse it. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.

Digital computers are themselves more complex than most things people build: They have very large numbers of states and components. This makes conceiving, describing, and testing them difficult. Software Systems have orders-of-magnitude more states than computers do. Likewise, a scaling-up of a software entity, through the addition of a new functionality (or the installation of a new application) is not merely a repetition of the same elements in larger sizes; it is necessarily an increase in the number of different components, inter-dependencies and interactions.

For three centuries, mathematical and the physical sciences made great strides by constructing simplified models of complex phenomena, deriving properties from the models, and verifying those properties through experimentation. This paradigm worked because the complexities ignored in the models were not the essential properties of the phenomena. It does not work when the complexities are the essence.

The complexity of software requirements and dependencies is an essential property, not an incidental one. It is necessary to describe software entities without abstracting their complexities. The classic problems of deploying software products derive from this essential complexity and its nonlinear relationship with a number of its requirements. Today complex software products include a number of installation options, starting with alternative operating systems and going as far as offering multiple topology options.

This complexity has entered the market recently calling for additional levels of expert knowledge. Multiple installation tools rely on the assumption that both the deployment topology and machines are predetermined for the software in question.

The process of software deployment is based on software dependencies that identified by installation prerequisites. Prerequisites are hardware or software ***requirements***. Prerequisites are fulfilled by ***capabilities***. Capabilities are attributes of software or hardware such as an operating system version or the disk size on a target system.

Rough Set Theory deals with data tables called information system. The theory provides an extended apparatus for the decision support using logical operations on decision and condition attributes. A decision algorithm that emerges is simplified to define an optimal data description.

In this paper, we use some of the concepts of Rough Set theory to interpret and model the process of matching application requirements to existing system capabilities. Using a fairly simple model to analyze the software application requirement, we define entities to represent software requirements elements and their expected attributes. Further, we propose an algorithm for eliminating conflicting and redundant requirements, as well as defining the minimal topology for large distributed applications.

This paper proposes a model and a process of analyzing Application requirements in order to determine sets of minimal non-redundant requirements. Each of these sets can constitute an installation for a particular target.

In this work we also address the question of finding a minimal number of servers required for the application within its' admissible topologies.

The remainder of the paper is structured as follows. Initially, we introduce the problem of planning the software deployment and describe some of the existing standards and techniques.

In section two we describe the concept of system capabilities, provide examples of capabilities description and provisioning software that uses this concept.

Section three focuses on concept of non-redundant and non-conflicting requirements and provides an illustration of the process of minimizing requirements. It also explains and offers an example of the Application with multiple deployment topologies.

In section four we give an introduction to Rough set theory and its notation and describe software requirements in terms of Rough set theory. In this section we also introduce the formalization models for creating minimal non-conflicting sets of requirements, simple matching it to the set of system capabilities.

We conclude with our plans for future work.

II. SYSTEM CAPACITY: DESCRIPTION OF CAPABILITIES

The recent trend of increased company acquisitions has highlighted the necessity for convergence of multiple systems. As a result, the non-software industry has been showing significant interest in the complex problem of system capacity detection

Our approach to this problem is quite different from the solutions proposed in the documents and recommendations that emerged from this process. In much of computer literature the term system capacity is used to denote hardware capacity. We consider the hardware facilities of the system as part of its capabilities, however our main interest here is the software installed on this hardware. Another obvious difference is our goal, which is neither to evaluate nor predict the performance or cost of the system: we are interested in the capacity of the system as its potential to additional software installations. While acknowledging the complexity of a system defined as a set of applications running on multiple servers, for the sake of simplicity we initially refer to examples restricted to a one machine setup. For further simplicity of concepts in this paper, we also limit the capabilities to those we are directly interested in.

The software installed can be described at application level and in terms of software componentry installed. The common approach to expressing the software capacity of the system is to use a collection of installed applications. While this approach has certain advantages (such as brief description and clear indication of the major system usage), it is definitely lacking knowledge of the system's fine-grained capabilities. Software componentry, as a common and convenient means for inter-process communication, relies on the very detailed list of installed components to take advantage of its interchangeability and possible reuse by other applications. Having the list of componentry, however, doesn't eliminate the need of providing a high-level view of installed software. In some cases this high level view could provide the basis for such business level decisions as choosing the deployment system for an application that requires processes already running on this system.

Therefore, it is advisable to define the capability of the system in terms of both application level and componentry level. It is apparent that this definition allows overlapping and redundant information. This collection for many systems is rather large, and it is clear that some order should be established to facilitate efficient use of this collection.

We will examine the following capabilities for the system under review: hardware capabilities, application-level capabilities and software component capabilities (Fig. 1):

HardwareResource:
- Hardware.Type
- Hardware.Memory
- Hardware.Disk

Operating System OS:
- OS.Name
- OS.Version
- OS.ServicePack

Application:
- Application.Architecture
- Application.Vendor
- Application.BaseName
- Application.Subtype
- Application.Type
- Application.ServicePack
- Application.RevisionLevel
- Application.Language
- Application.OperationalStatus
- Application.AvaliabilityStatus
- Application.CriticallIndex

SoftwareComponent SC:
- SC.Name
- SC.Type
- SC.Version
- SC.BuildNumber
- SC.RevisionLevel
- SC.Language
- SC.OperationalStatus
- SC.AvaliabilityStatus
- SC.CriticallIndex

Figure 1: Capabilities of the System

Notice, that application and software component have mostly the same set of capabilities, which is easy to explain as functionally those two categories describe the same ability of the system to participate in or initiate and sustain the process, and the Application is the group of Software Components that run as logical entities. We will consider the set of described capabilities from a functional adaptation point of view, which implies that while there are many aspects to system capabilities, our interest lies with one of most essential and basic ones- an offered functionality or state transformation performed by the Application or Software Component.[1]

This approach to expressing system software capacity through a collection of capabilities is used by IBM Tivoli Provisioning Manager 3.1. The image below shows the software installed on specific machine that expresses the locale capability of the software

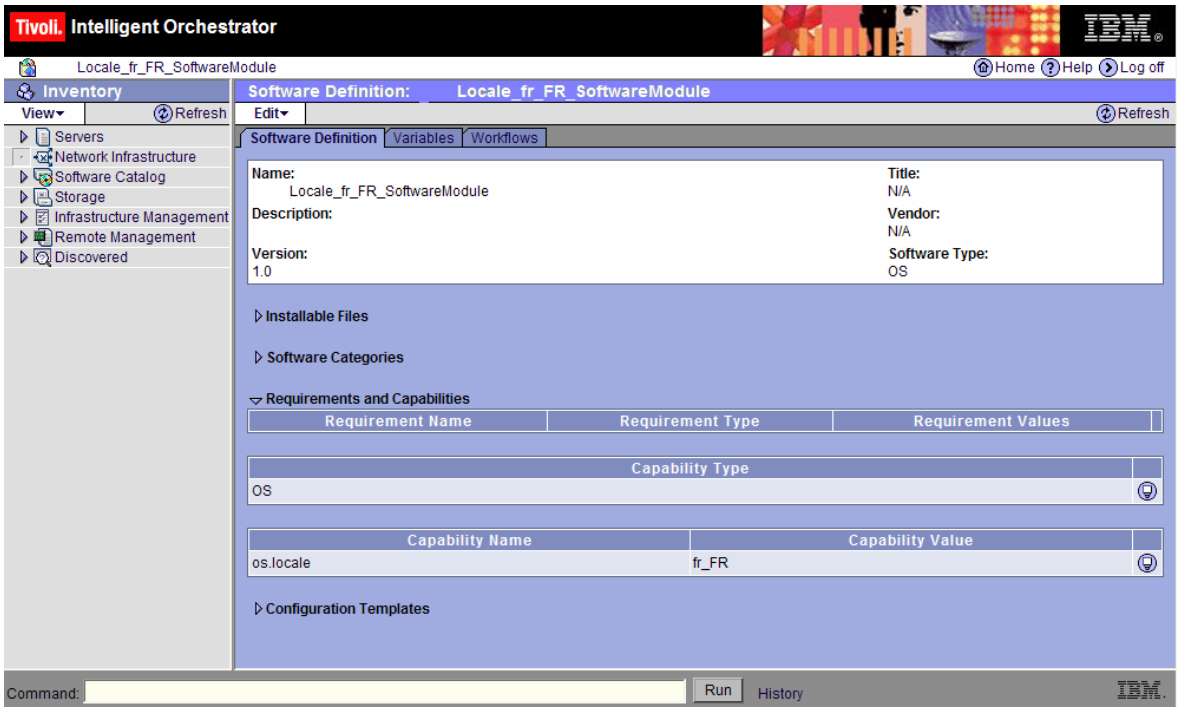

Figure 2: Provisioning Manager.

Similarly, it is permissible for each software to express a number of different types of capabilities or for a set of installed products to express alternative capabilities of the same type. So in addition to the "Locale fr_FR_Software module" that has capability os.locale=fr_FR (Figure 2), the software module "Locale en_CA_Software module" that has capability os.locale=en_CA could exist on the same machine; thereby expressing the capacity of this system to accept future installations of either of French or Canadian English locale.

III. REQUIREMENTS

A set of Requirements for the Application installations is a focal point of this paper. While traditionally software requirements are described in a human-reading format, since the beginning of the automation era multiple attempts have been made to establish a unified descriptor standard that could be easily processed by automation. Despite these efforts there are still multiple known requirements descriptors. One often used req. descriptor is Solution Installation Schema (IUDD) that was submitted to W3C as a standard for the description of installable unit characteristics in 2004.[2, 4] This schema is used to describe a single installable unit (IU), its content, checks, dependencies and configuration. A more refined version of this schema (SDD) [12] emerged from OASIS. This schema is used by the CHAMPS System [13] to generate a Run-time dependency Model based on the results of a discovery process.

IU is a key abstraction of IUDD used to describe the building blocks that comprise the Application. In this paper we will refer to these building blocks as 'components'. However, we won't be concern with the mapping of units used here to the installable units' types described in the IUDD schema, as the notions and results presented here are independent of the requirements descriptors. IUDD and other deployment descriptors relay on the assumption that topology is defined and machines are allocated for an installation of a particular Application. Meanwhile we attempt to determine a set of minimal non-redundant requirements as well as define a minimal application topology for the installation.

A. Complex Applications installation requirements

Example: This example will describe some portion of the installation dependencies for IBM Tivoli Intelligent Orchestrator (TIO). This Application could be deployed within the following topologies:

- Topology:
  - One-node topology
  - Two-node remote directory server.
  - Two-node remote directory and database server.
  - Two-node remote database server.
  - Three-node topology.

Figure 3: Topology choices.

The higher level optional requirement immediately throws the requirements representation into a new realm of complexity. So for our first example we will consider a one-node topology with the following Hardware Requirements.

- Hardware Minimum Requirement
  - Disk space
    - 30 GB
  - RAM
    - 2GB
  - Processor
  - Dual -Processor 2.4 GHz and up each Intel Pentium
- Required authorization
  - Administrator (on the operating system)

Figure 4: Hardware Requirements.

Here we will introduce yet another simplification: we will assume that system of interest satisfies the hardware requirements, thereby eliminating another factor in our considerations.

Operating System
- Microsoft Windows 2000

OR
- Microsoft Windows 2003

Software
- Directory Server
  - IBM Tivoli Directory Server Version 5.2
    - Operating System (following OR list)
      - Windows 2000
      - Windows Server 2003, Standard or Enterprise
      - Windows NT 4.0 with Service Pack 6 or later
      - ...
    - Database
      - DB2 Version 8.2 with FixPak 5 or later
    - OPTIONAL
      - GSKit (following OR li
        - IBM JRE
        - JDK 1.4.1
  - Microsoft Active Directory
    - Windows 2000
      - SP 4 or higher
    - Windows XP
- Application Server
  - WebSphere Application Server 5.1.1 (following OR list)
    - Windows-Base
    - Windows-ND
    - Windows-Express
    - ...
  - Cumulative Fix 3
    - V5.1.1 Java SDK Service Release
- Database Server
  - IBM DB2 Universal Database Enterprise Edition 8.2
    - Operating System (following OR list)
      - Windows 2000
        - Service Pack: SP 4 or later
      - Windows 2003
        - Service Pack: SP 2
      - Windows XP
        - Service Pack: SP 2
    - Browser
      - Mozilla
        - V 1.0 or greater
      - IE
        - Operating Systems (following OR list)
          - Windows XP
            - IE V 6.0 o higher
          - Other Windows Operating Systems
            - IE V5.5 or higher
- Cygwin
  - Cygwin Version 1.3.22

Figure 5: TIO Requirements for Windows OS

In this extended set of requirements we have omitted the details for non-Windows systems, yet again assuming that the targeted system has a Windows operating system installed.

These simplifications resulted in a significantly reduced, hence more comprehendible, set of requirements.

B. Redundant or conflicting requirements

Based on the example above it is easy to see how complex multi-application installations like IBM TIO often set conflicting objectives .

In our example, DB2 Universal Database Enterprise Edition 8.2 could be installed on different types of hardware and multiple Operating Systems (such as AIX, HP-UX, Linux, etc). By limiting Installation topology to one-node deployment we reduced the set of requirements, and at the same time increased the number of conflicting requirements. To resolve the conflicting requirements we must find atomic requirements that are in conflict with one another or to find applications with conflicting requirements.

Another activity that goes hand-in-hand with conflict resolution is illuminating redundant or irrelevant requirements. This type of requirements adds unnecessary complexity and additional dimensions to the set of requirements, therefore reducing comprehensibility and processing speed.

So after a further reduction of the requirements through elimination of conflicting, redundant and irrelevant entries, the set looks much more comprehensible

ok

- Operating System (following is or list)
  - AIX
    - AIX 5.1
    - AIX 5.2
    - AIX 5.3
  - x/Linux
    - RHEL 2.1 AS
    - RHEL 3.0 AS
  - Solaris
    - Solaris 8
    - Solaris 9
  - z/Linux
    - SuSe Linux 8.1
  - MS Windows
    - Windows 2000
    - Windows 2003
- Software
  - Directory Server
    - IBM Tivoli Directory Server Version 5.2
      - Operating System (following OR list)
        - Windows 2000
        - Windows Server 2003, Standard or Enterprise
        - Windows NT 4.0 with Service Pack 6 or later; a Windows NT file system (NTFS) is required for security support
        - AIX 4.3.3
        - AIX 5.1
        - AIX 5.2
        - Red Hat Enterprise Linux 3.0
        - UnitedLinux 1.0
        - SuSE Linux Enterprise Server 8
        - ...
      - Database
        - DB2 Version 8.2 with FixPak 5 or later
        - DB2 8.1 ESE w/FP3+
        - Oracle 8i S/E R3 8.1.7+
        - Oracle 9i S/E R2 9.2+
        - Oracle 10g
      - OPTIONAL
        - GSKit (following OR list)
          - IBM JRE
          - JDK 1.4.1
    - Microsoft Active Directory
      - Windows 2000
        - SP 4 or higher
      - Windows XP

Figure 6: TIO Requirements - part 1 1

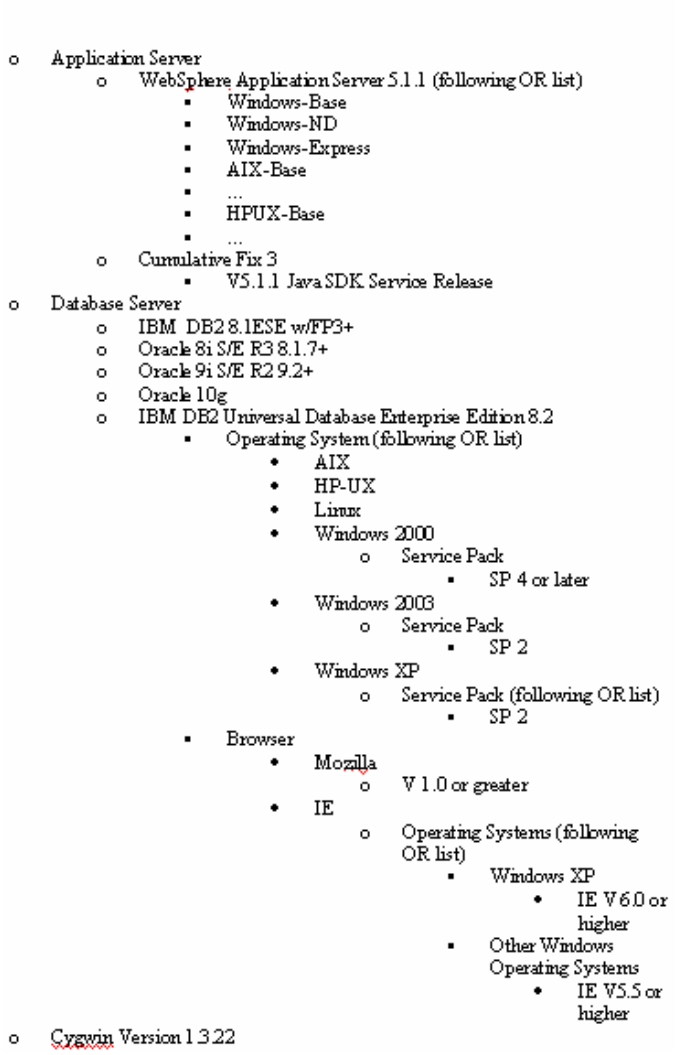

Figure 7: TIO Requirements - part 2

IV. Simple Matching: Overview and Formalization Model

We will introduce the formalization model using Rough Sets theory.

- *Introduction of Rough Sets*

The theory was introduced by Zdzislaw Pawlak in the early 1980's, and based on this theory one can propose a formal framework for the automated transformation of data into knowledge. The Rough Set theory is mathematically relatively simple. Despite this, it has shown its fruitfulness in a variety of areas. Among these are information retrieval, decision support, machine learning, and knowledge based systems (for example see [5]).

Rough set based data analysis starts from a data table, called an information system. The information system contains data objects of interest characterized in terms of some attributes. When in the information sytem the decision attributes and conditions attributes are clearly defined a decision table. The decision table describes desicions in terms of conditions that must be satisfied in order to carry out the decission specified in the decision table. With every decision table we can associate a decision algorithm which is a set of 'if…then…' decision rules. The decision rules can be viewed as logical a description of the basic properties of the data. The decision algorithm can be simplified, leading to optimal data description .

We will soon extend the Rough Sets theory to handle multi-dimentional requirements.

- *Formalization Model for creating minimal non-conflicting sets of requirements*

We will describe the algorithm for finding minimal non-conflicting sets of requirements based on the previous example: deployment of IBM Tivoli Intelligent Orchestrator. The decision table in our case has Requirements on the various levels – from Applications and Components to Operating System (OS) and Hardware Resources (HR). An Application requirement could have another Application or Component OS or HR as a requirement, while a Component is atomic in a sense that it may depend only on OS or HR.

Step 1.

We will create a sequence of decision tables for Application Requirements on different levels. In order to handle Optional requirements we will introduce a relational variable that represents this nature of the requirement: more specifically, all mandatory requirements will have the same Relation Variable, while optional or alternative requirements will be assigned different values. We will say that each table represents a level state of the Application.

Application/Component	Requirement	Relation Variable
IBM Tivoli Intelligent Orchestrator	Directory Server	
	Application Server	1
	Database Server	1
	Cygwin	1

Highest Level. Table 1

Note: All four Applications are necessary for completing the installation, therefore the Relational Variable for each is has the same value 1.

Application/Component	Requirement	Relation Variable
Directory Server	IBM Tivoli Directory Server Version 5.2	1
	Microsoft Active Directory Server	2

Directory Server's Decision Table. Table 2

Here, since only one of the Directory Server Applications requires the Relation Variables have different values??. Similarly, decision tables have to be created for each Application and Component in Figure 7. Below Table N is a decision table for component Internet Explorer.

Application/Component	Version	Operating System	Relation Variable
Internet Explorer	6.0 or higher	Windows XP	M
	5.5 or higher	All Windows but WinXP	M+1

Table N

Step 2
Now, we will iterate through the decision tables formed in Step 1 to create a unified decision table for IBM Tivoli Intelligent Orchestrator.

The following Table 1.2 is the resulting table for Table 1 and Table 2 in Step 1.

Application/Component	Requirement	Relation Variable
IBM Tivoli Intelligent Orchestrator	Cygwin	1.1
	Application Server	1.1
	Database Server	1.1
	IBM Tivoli Directory Server Version 5.2	1.1
	Cygwin	1.2
	Application Server	1.2
	Database Server	1.2
	Microsoft Active Directory Server	1.2

Table 1.2

Decision Table 1.2 has two groups of requirements identified by Relation Variable (RV). The Relational Variable can also be viewed as a representation of the "requirement path".

It is important to note here, that this representation is always possible due to distributive law

$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$.

At each level independent groups of requirements minimized to exclude redundant requirements, as well as groups with conflicting requirements could be identified and excluded from consideration.

Enumeration of all possible options and knowledge of the order of iterations define Relation Variable in a unique manner. In reverse nope, Relational Variable, enumeration of the options and order of iteration allows for the restoation "requirement path".

Relationships among attributes of capabilities vs. requirements will be defined later.

We will consider an algorithm where requirements are processed "wide-first". Let f[][] denote the two dimensional array for storing groups of requirements as showen in Information Tables 1, 2, etc. .

Note, that here we consider that there are no cyclical dependencies. In practice cyclical dependencies are possible (for example kernel dll depended on user dll, and user dll depended on kernel dll)

```
Procedure
Eliminate_conflicting_and_redundant_requirements

for i=1 to 'group requirements count' do
  countRV <- requirements count for the
  fixed RV //for one application
  bRemove = false;
  for j=1 to countRV do
     a: for z=j+1 to countRV do
          if f[i][j] ∧ f[i][z]=f[i][j] then
               remove_dependent_line(i,z)
               continue a;
               //do not remove this requirement
               if f[i][j] ∧ f[z][k]= ∅ then
                    bRemove =true;
                    break a;
        endfor;//z
  endfor;//j, requirements of f[i][*]
  if bRemove then
     remove group requirements f[i][*];
endfor;//i, group requirements number
```

Procedure 1

A. Formalization model for capabilities/ requirements matching

After eliminating conflicting and redundant requirements, we are left with well-optimized sets of components. Each set, as mentioned above, has a value that indicates its' "requirement-path". This value allows or the identification of higher level of requirement for each set by looping through tables with appropriate identifiers to mark applications that produced this requirement. These marked Applications will than be added to the groups. This process will extend each minimized non-conflicting set of requirements with Application level requirements, creating partially ordered sets. In our example above, one of the sets will have the following structure

{(TIO), (IBM Tivoli Directory Server Version 5.2, IBM DB2 Universal Database Enterprise Edition 8.2, WebSphere Application Server 5.1.1, Cygwin), ……..(Windows 2000, SP 4), …}

We've separated the requirements of the same level by parentheses for additional clarity.

Newly created sets, while redundant as whole (for example all requirements in the set are included in some way in the requirement TIO), are unique in the subset of level-requirements.

This type of requirements set could be easily expressed in the same terms as capabilities of the system (see section 2), which makes easy comprehendible simple algorithm for matching capabilities to requirements:

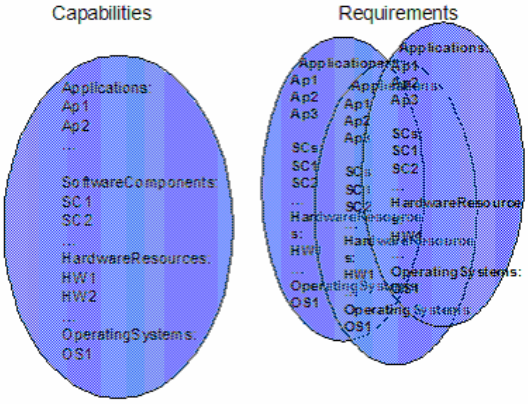

Figure 8: Matching Capabilities and requirements

Looping through sets of requirements, applications followed by software components to identify the match between an existing capabilities set and each set of requirements. This simple one-to-one algorithm could produce multiple matching pairs: (Capability, RequirmentsGroup) with missing requirements clearly identified. Those requirements combined into the ordered list (this time it is components first then applications) will define installation procedures. This algorithm relies on the fact that Capabilities include all installed Applications and Software Components, thus ensuring that a match found on the Application level, will be found on? for the components, which are required for this application, and therefore both Application and Components (dependencies) excluded from install procedures.

Let f[][] denote the two dimensional array for storing groups of requirements as showed in Tables 1, 2, etc. c[] is a capabilities array .

```
Procedure: Find_set_of_compliments_to_admissible_installations
J <- empty set //set of admissible installations
a: for i=1 to 'group requirements number' do
     K<- empty set //set of missing components
     for j=1 to requirements number of f[i][*] do
          bInclude = true;
          b:for z=1 to 'capabilities number' do
               if f[i][j]=c[z] then
                    bInclude=false and break b;
               //do not include this requirement
               if f[i][j] contradicts c[z] then
                    continue a:
          endfor;//Z
          if bInclude then
               K add the requirement f[i][j];
     endfor;//j, requirements of f[i][j]
     J add K;
endfor;//i, group requirements number
```

Procedure 2

The optimization of this algorithm will be introduced in future work (see conclusion and future work session).

The result of this procedure is multiple sets of components and applications, each representing a complete installation. By introducing the cost function that defines the goal of optimization, the best installation can be identified. Some well known optimization goals/metrics are: time of installation (downtime for installation), additional licensing cost, minimal complexity (introduced in [6]), etc. If additional information is available about components/applications that are included in the installation, the optimization function could be formulated as CPU utilization, required IO, storage space etc.

In a closely managed application rich environment, the challenge could be to identify a machine that will lead to minimal downtime, insuring that customer agreements are satisfied.

Now we will consider the task of identifying an optimal machine out of a pool of existing groups for optimal (in terms of cost function) application installation.

Let fC[][] denote a two-dimensional array for storing cost of one of the installations on a particular machine. To efficiently find the minimal value of the cost function on the set of installations on all machines, we will update the above algorithm with a procedure for determining the minimum of this function.

```
Procedure find_optimal_node_and_installation
optimal_node<-0; min_cost<- infty //optimal node
K<- empty set //optimal installation
for k=1 to 'number of nodes' do
 temp_cost<- infty; temp_K<- empty_set
 a: for i=1 to 'group requirements number' do
    t_cost<-0; t_K<- empty_set
    for j=1 to requirements number of f[i][*] do
      bInclude = true;
      b: for z=1 to 'capabilities number' do
         if f[i][j]=f[z] then
           bInclude=false and break b;
           //do not include this requirement
         if f[i][j] contradicts c[z] then continue a:
           endfor;//Z
         if bInclude then
              t_K add requirement f[i][j];
              t_cost+=fC[i][j];
    endfor;//j, requirements of f[i][*]
    if t_cost < temp_cost then
         temp_cost=t_cost; temp_K <- t_K
 endfor;//i, group requirements number
 if temp_cost<min_cost then
    optimal_node<-k; K<- temp_K;
endfor;// k, number of nodes
```

Procedure 3.

After installation is completed and verified, the collection of requirements that are propagated into install procedure could be added to the set of system Capabilities, thus completing the capabilities-requirements loop.

B. Formalization model for minimal topology determination in case of multi-node Application installation

The process of eliminating conflicting requirements described above (Section 3) could result in the production of an empty set, which will mean that this Application in fact can not be installed on one machine. Then the problem (at least from a practical point of view) becomes determining the \ minimum number of machines required for a given application.

The necessary constraint in this case is the absence of conflicting requirements for one machine. In considering a set of all requirements, we will introduce the concept of Height as number of disjoint requirements on each element (component). The function EH(ε) will be defined as the maximum number of disjoint requirements for the element ε for given Installation. We summarize the above in

Proposition 1: *The following condition on number of nodes in topology is necessary for the existing of installation on the specified number of nodes (#nodes):*

$$\min_{Ins\in\mathcal{J}} \max_{\mathcal{E}\in Ins} EH(\mathcal{E}) \le \#nodes$$

, here J is a set of all admissible installations.

To further refine the condition we will consider the following notion: we will express the set of requirements as a set of the subsets of independent groups of non-conflicting requirements. For example, when requirement for the Application(A) has subrequirements Database(D) and Application Server(S) and those subrequirements D and S could be installed on different nodes("allowed separations"), continuing this process we will get at least two sets of requirements that correspond to different nodes.

Definition: *To reflect this notion we introduce the concept of Partition denoted by*

$$\mathcal{P}(Ins) = \{\{\mathcal{E}_1, ..., \mathcal{E}_{i_1}\}, ..., \{\mathcal{E}_{i_{n-1}}, ..., \mathcal{E}_{i_n}\}\}$$

In principal, it is enough to consider minimal partition, which is one that does not contain further subdivisions.

The following inequality provides an upper estimate for the number of nodes in an admissible installation.

Proposition 2: *The number of nodes in the installation is not greater than count of partitions:*

$$\max_{Ins\in\mathcal{J}} \#\mathcal{P}(Ins) \ge \#nodes$$

Refining the estimate of **Proposition 1**, we get the following proposition.

Proposition 3: *The following inequality is valid.*

$$\min_{Ins\in\mathcal{J}} \max_{\mathcal{G}_i\in\mathcal{P}(Ins)} \max_{\mathcal{E}\in\mathcal{G}_i} EH(\mathcal{E}) \le \#nodes$$

Moreover, left side of the inequality defines precise minimal number of nodes required for the installation.

The described above process of creating groups of non-conflicting requirements (based on number of disjoint requirements for each component) with subsequent separation of those into partitions ("allowed separation") and then installations, will allow one to calculate the minimal number of nodes required for the installation of this Application.

V. Conclusions and Future Work

In this paper we have concentrated on the provisioning planning that precedes the installation. The complexity of decisions that has to be made prior to installation traditionally was neglected. New generation of distributed software makes these decisions increasingly tough. We use concepts of requirements and capabilities to apply Rough set theory apparatus to facilitate these decisions. We propose algorithm for defining sets of non-redundant non-conflicting requirements.

We also suggested a solution for finding precise minimal number of nodes required for the installation.

This will be further extended to include solutions for identifying the best target for new installation within the group of machines and defining the best topology for the application to be installed.

We will continue to apply Rough Set theory formalization that allows the introduction of complex match functions between Capabilities and Requirements, thus permitting analysis of matching algorithms from different points of view: performance, system utilization, cost and business impact optimization.

REFERENCES

[1] *Shikawa, Y. Matsumoto, M.J. Identifuing the Structure of Business Process for Comprehensive Enterprise Modeling.* IEICE Trans. On Inf. And Syst. Vol. E83-D. No4. 691-700

[2] *Installable Unit Package Format Specification Version 1.0*. W3C Member Submission 12-July-2004. Version URL: http://www.w3.org/Submission/2004/SUBM-InstallableUnit-PF-20040712/. Latest version URL: http://www.w3.org/Submission/InstallableUnit-PF/

[3] *Skowron A. and Rauszer C.:The Discernibility Matrice and Functions in Information Systems. Intelligent Decision Support.* Handbook of Applications and Advances of Rough Sets Theory. Dordrecht: Kluwer (1992) 331-362

[4] *Installable Unit Deployment Descriptor Specification Version 1.1*. W3C Member Submission 12-July-2004. Version URL: http://www.w3.org/Submission/2004/SUBM-InstallableUnit-DD-20040712/. Latest version URL: http://www.w3.org/Submission/InstallableUnit-DD/

[5] Z.Pawlak. Rough Sets. Theoretical Aspects of Reasoning about Data. Kluwer Academic Publishers, Dordrecht, 1991

[6] A model of Configuration Complexity and its Application to a Change management System Aaron B. Brown, Alexander Keller, Joseph L. Hellerstein http://www.research.ibm.com/people/a/akeller/Data/im2005.pdf

[7] All about IBM Tivoli Configuration Manager Version 4.2, December 2002. IBM Redbook, Order Number: SG24-6612-00. see also: http://www.redbooks.ibm.com.

[8] D. Ressman and J. Valdes. Use of Cfengine for Automated, Multi-Platform Software and *Patch Distribution. In Proceedings of the Fourteenth Systems Administration Conference (LISA 2000), New Orleans, LA, USA, December 2000. USENIX Association – open source alternative to Tivoli Config Manager*

[9] Helene Kirchner, M. Cerioli, Z. Qian, M. Wolf (Editors), Algebraic System Specification and Development Survey And Annotated Bibliography,2nd edition, 1997, number 3 in Monographs of the Bremen Institute of Safe Systems, Shaker, 1998.

[10] Deja R Conflict Analysis. Rough Sets Methods and Applications New Developments. In: L.Polkowski et al. (eds.) Studies in Fuzziness and Soft Computing, Physica-Verlag, pp.491-520

[11] Pawlak Z (1998) An inquiry into Anatomy of Conflicts. Journal of Information Sciences 109 pp.65-78

[12] OASIS Solution Deployment Descriptor (SDD) TC http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sdd

[13] A.Keller, J.L.Hellerstein, J.L. Wolf, K.-L. Wu, V.Krishnan The CHAMPS System: Change Management with Planning and Scheduling Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (MONS 2004), Seoul, Korea, April 2004