TECHNICAL RESEARCH REPORT

Graph-Based Visualization of System Requirements Organized for Team-Based Design

by Vimal Mayank, Natalya Kositsyna, Mark Austin

SEIL TR 2005-3
(ISR TR 2005-92)

ISR Technical Report 2005-92

Graph-Based Visualization of System Requirements Organized for Team-Based Design

By Vimal Mayank[1], Natalya Kositsyna[2] and Mark Austin[3]

Last updated : June 29, 2005.

[1]Faculty Research Assistant, Institute for Systems Research, University of Maryland, College Park, MD 20742, USA.
[2]Faculty Research Assistant, Institute for Systems Research, College Park, MD 20742, USA.
[3]Associate Professor, Department of Civil and Environmental Engineering, and Institute for Systems Research, University of Maryland, College Park, MD 20742, USA.

Contents

Chapter 1

Introduction

1.1 Problem Statement

In systems engineering circles, it is well known that requirements management capability improves the likelihood of success in the team-based development of complex multidisciplinary systems. Two key elements of this capability are an ability to identify and manage requirements during the early phases of the system design process. This is when errors are cheapest and easiest to correct. Furthermore, all requirements must be validated against the purposes and functions of the system, and verified against appropriate specifications. Now that documents containing thousands and, sometimes, tens-of-thousands of requirements are commonplace, requirements management tools are an indispensable enabler of the system development process.

Present-day requirements management tools such as SLATE [1], CORE [3], DOORS [4] provide the best support for top-down development where the focus is on requirements representation, traceability, allocation of requirements to system abstraction blocks, and recently, step-by-step execution of system models. We note that at this time, computational support for the bottom-up synthesis of specific applications from components is relatively poor. Most of today's requirements management tools represent individual requirements as textual descriptions with no underlying semantics. As a result, computational support for the validation and verification of requirements is still immature – although some tools do have a provision for defining how a particular requirement will be tested against relevant attributes, it is not enough. Current tools are incapable of analyzing requirements for completeness or consistency. Search mechanisms are limited to keywords, which can be limiting for custom jargon in multidisciplinary and multilingual projects.

State-of-the-art practice is to organize groups of requirements (e.g., functional requirements, interface requirements) into tree hierarchies. However, when requirements are organized into layers for team development, graph structures are needed to describe the comply and define relationships among requirements. When software tools employ a tree-based model to display relationships among requirements, gaps appear between the visual representation and the underlying graph-based data structures. Systems engineers currently use manual procedures to identify and close these gaps. In an effort to mitigate the limitations of this slow and error prone process, in this study we formulate algorithms and implement

software tools for the graph-based organization and visualization of requirements.

Team-Enabled Systems Engineering at NASA Goddard

This work is motivated by needs of the Global Precipitation Measurement Project [6] at NASA Goddard Space Flight Center. Briefly, NASA's GPM project will improve climate, weather, and hydro-meteorological forecasts through more frequent and more accurate measurement of precipitation across the globe. The implementation of NASA GPM is a multi-national effort that will require the launch and operation of at least seven satellites and the participation of development teams in at least five countries. The system design and implementation is expected to occur through 2015.

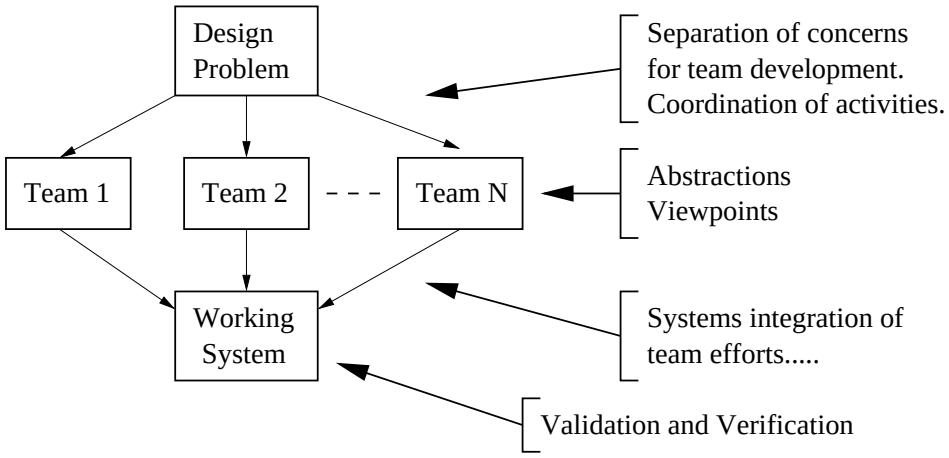

Figure 1.1: Key concerns in Team Development of Systems (Source: Discussion with David Everett, NASA Goddard)

As indicated in Figure 1.1, methodologies for the team development of system-level architectures need to support the following activities:

1. Partitioning the design problem into several levels of abstraction and viewpoints suitable for concurrent development by design teams. These teams may be geographically dispersed and mobile.

2. Coordinated communication among design teams.

3. Integration of the design team efforts into a working system.

4. Evaluation mechanisms that provide a designer with a critical feedback on the feasibility of system architecture, and make suggestions for design concept enhancement.

Throughout the development process, teams need to maintain a shared view of the project objectives, and at the same time, focus on specific tasks. It is the responsibility of the systems engineer to gather and integrate subsystems and to ensure ensure that every project engineer is working from a consistent set of project assumptions. This requires an awareness of the set of interfaces and facilities to which the system will be exposed.

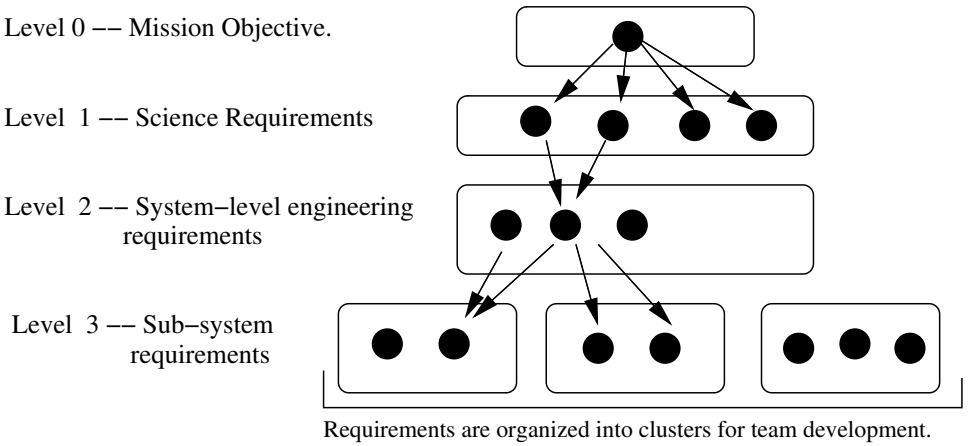

Figure 1.2: Layered Development and Organization of Requirements at NASA Goddard

Figure 1.2 is a high-level schematic of layered requirements development at NASA Goddard Space Flight Center. Generally, layers of requirements correspond to levels of responsibility. Initially, the requirements elicitation process is motivated by a statement of "mission objectives" that will aim to advance our scientific understanding in one or more ways. Levels 1 and 2 focus on the definition of "science" and "high-level engineering" requirements, respectively. At Level 3, engineering requirements are organized into clusters (e.g., ground segment; communications segment; satellite segment) suitable for team development. Requirements at Levels 4 and 5 may be directed to a specific sub-system (e.g., an instrument that will fly on a satellite) or component (e.g., a circuit board).

As of June 2004, the NASA GPM project contains about 1100 requirements. Present-day practice is to manage requirements using SLATE, a commercial software for developing complex systems, for manipulating their requirement documents. SLATE uses a folder mechanism to store requirements. Different types of requirements may be stored in different folders, and then connected using complying and defining links. Systems engineers like to work with data/information organized into folder/tree structures because they are familiar and convenient. For example, tree structures naturally occur when paragraphs, requirements, and so forth are extracted from a Word [19] document. SLATE has a Visio [17] interface for the block diagram visualization of requirements.

Organization and Management of Requirements

The tree representation of requirements hierarchies is fundamentally flawed and, in our opinion, only works well when requirements comply/define from a single source[1]. As illustrated in Figures 1.2 and 1.3, a number of complicating relationships are possible. First, a child node (complying requirement) may have more than one parent node (defining requirement). This means that requirements documents need to support multiple inheritance structures. Furthermore, as requirements are classified and broken down into granular components, they can also trace across the same level (i.e., one requirement may

[1]In our opinion, tree representations are useful only to see one level of complying or defining requirement (i.e., a very small subsection of the requirement document).

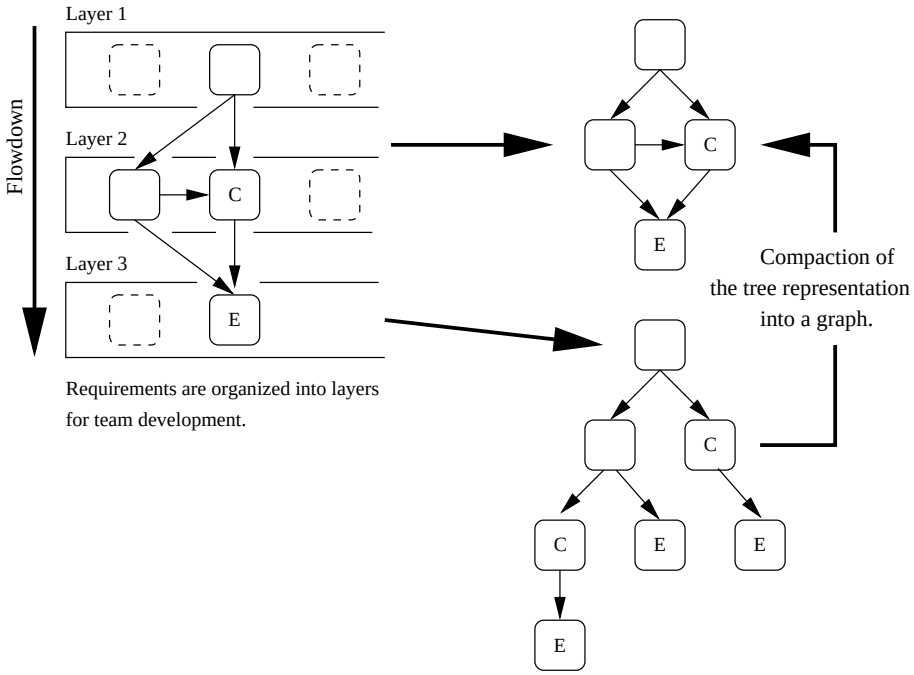

Figure 1.3: Many-to-Many Relationships in Layers of Requirements. On the right-hand side we show extraction and visualization of requirements as a tree, followed by compaction back in to a graph format.

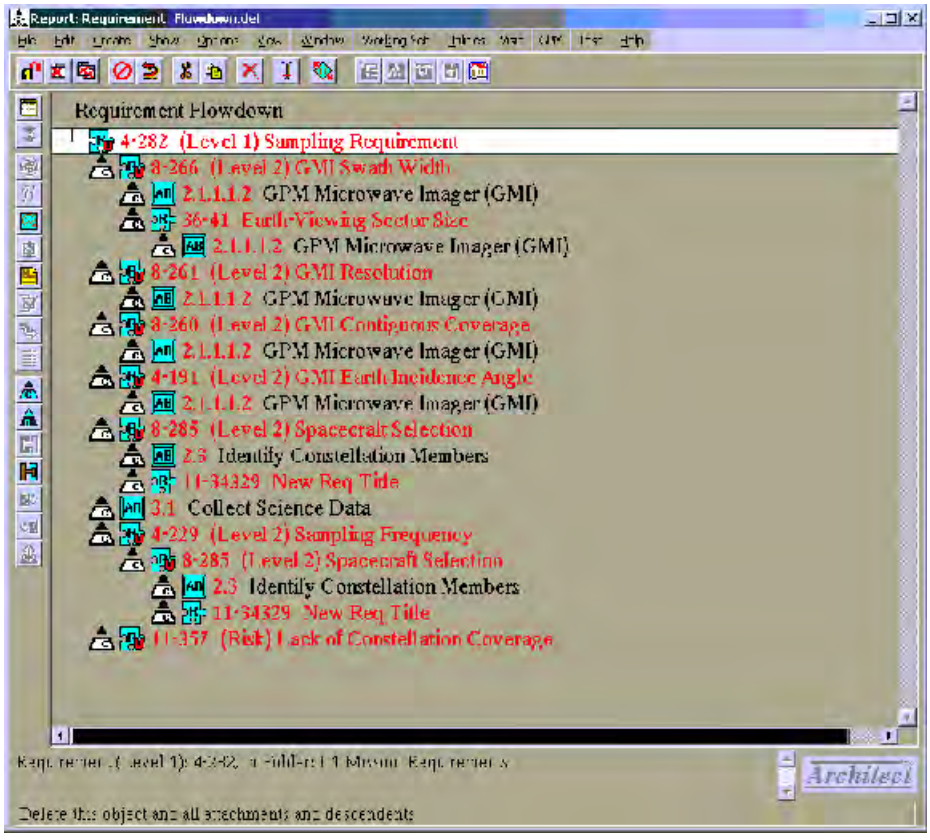

Figure 1.4: Tree Representation of Requirements in SLATE (Source: Dave Everett, GPM Project Group, NASA Goddard)

comply or define the other requirements). Figure 1.3 shows, for example, a partial requirement document with requirements arranged into layers. Here, requirement C in layer 2 defines requirements E in layer 3. Conversely, requirement E complies with requirement C. When the underlying data structure for the requirements is a graph, but visualization procedures assume that a tree structure is sufficient, resolution of this incompatibility is handled through the duplication of nodes in the visual representation. As a case in point, the tree representation of requirements in Figure 1.3 has duplicate entrices for nodes C and E. Similarly, Figure 1.4 shows the tree structure of a complying requirement relationship modeled in SLATE [1]. The tree representation leads to repetitions of the node GPM Microwave Imager under the Sampling Requirement.

When the overall number of requirements is very small (i.e., less than, let's say, a dozen requirements), the duplication of requirements nodes is not a significant barrier to a complete understanding of relationships in the requirements document. A systems engineer can easily find all of the duplicates and manually reconstruct the graph structure. For other cases, including all real world problems, the presence of the duplicate nodes vastly complicates the task of interpreting/understanding the requirements document structure. When the manual identification of all complying and defining requirements becomes intractable, designers cannot be absolutely certain all comply/define relationships have been identified. This leads to decision making based on an incomplete picture of a requirement's role in the system design.

1.2 Scope and Objectives

In a departure from state-of-the art practice, in this work we formulate algorithms and develop Java-based software that can read the tree representation of the requirement database, and construct and visualize the block diagram representation with all duplicate nodes removed. The input-to-screen transformation involves two steps:

1. The tree representation of the requirement document is provided as input and a graph data structure is constructed, which does not have duplicate nodes.
2. Construction of a block diagram visualization of this graph structure. This involves using a hierarchical graph layout algorithm to position the requirement nodes on the screen.

Since the heart of this work involves a new and different way of requirements document visualization, efficiency of the systems engineering process can be enhanced with several new features of requirements management and visualization, namely:

1. Visualizing a sub-section of the requirement document.
2. Differential update of the database of requirements, based on the changes done while visualizing the requirements.
3. Annotating the requirement nodes with their attributes of interest.

All of the above are analyzed in detail below:

1.3 XML/RDF Representation of Requirements

For the purposes of this work, input data from the SLATE database has a tree representation of the requirements containing duplicate nodes. The extensible markup language (XML) [20] provides such a schema to represent hierarchies with customizable tags defined by the user. This XML file can be generated conveniently using a script from SLATE.

Let us now look at a particular requirement node in detail. The fragment of code:

```
1   <Req ROIN="8-378" type="Mission Objective" level="0">
2           <Title text="Programmatic Requirements"/>
3           <Description text="The GPM shall adhere to program requirements
4                              as agreed between the NASA Earth Science Enterprise
5                              and the NASA/GSFC GPM program office."/>
6           <ReqList>
7                  <CompReq ROIN="8-2"/>
8                  <CompReq ROIN="8-4"/>
9                  <CompReq ROIN="8-5"/>
10                 <CompReq ROIN="8-7"/>
11                 <CompReq ROIN="8-8"/>
12                 <CompReq ROIN="8-10"/>
13              <CompReq ROIN="8-11"/>
14              <CompReq ROIN="4-468"/>
15              <CompReq ROIN="8-251"/>
16              <CompReq ROIN="8-259"/>
17              <CompReq ROIN="8-57"/>
18              <CompReq ROIN="8-43"/>
19              <CompReq ROIN="8-26"/>
20          </ReqList>
21          <Attribute text="Assigned To" value="None"/>
22          <Attribute text="Change Proposals Allowed" value="Not Assigned"/>
23          <Attribute text="Criticality" value="None"/>
24          <Attribute text="ID" value=""/>
25          <Attribute text="Inheritable" value="Yes"/>
26          <Attribute text="Qualification Date/Time" value="YY/MM/DD-24:00"/>
27          <Attribute text="Rationale" value=""/>
28          <Attribute text="Requirement State" value="Uncontrolled"/>
29          <Attribute text="Requirement Status" value="Active"/>
30          <Attribute text="Requirement Title" value="Programmatic Requirements"/>
31          <Attribute text="Test Report Number" value=""/>
32          <Attribute text="Verification Date/Time" value="YY/MM/DD-24:00"/>
33          <Attribute text="Verification Description" value=""/>
34          <Attribute text="Verification Level" value="None"/>
35          <Attribute text="Verification Method" value="None"/>
36          <Attribute text="Verification Status" value="None"/>
37          <Attribute text="Verified By" value="None"/>
38   </Req>
```

defines the attributes and the connectivity of the requirement object with unique (requirement object identification no) ROIN 8-378, its type as Mission Objective and numerical level as 0 (Line 1). Next, Title and Description tags identify the name of the requirement and what the requirement does respectively. ReqList identifies the ROIN of the requirement objects, which directly comply from 8-378 (for e.g. Requirement 8-2). The duplicates issue arises because in real world problems, a requirement such as 8-2 might have more than one parent apart from 8-378, and it will exist in the ReqList of all those parent requirement objects. Next several Attribute tags identify the name of the attribute and its value. These attributes vary for different type of requirements and from project to project. Different requirement nodes with their connectivity properties and the attribute lists are followed one after another to form one

big XML file.

The abovementioned XML schema has been tailored for this implementation. However, as the standard for product data exchange AP233 evolves [10, 11], the input mechanism can be switched from reading the above mentioned schema to read the STEP AP233 file, which is generated by different requirement management tools like SLATE and DOORS. We envision that by providing this capability, this tool will provide a powerful way to visualize the requirement structure from a variety of requirement databases as the AP233 demonstrator is still in its earlier stages of implementation at the time of writing this report.

Removal of Duplicate Requirement Nodes

Next, an equivalent data-structure is generated from this input XML file that does not contain duplicates. The Resource Description Framework (RDF) [12] defines an excellent mechanism for specifying connectivity relationships among objects in a general and simple way. Briefly, a RDF statement contains triplets viz. subject, predicate and object. Within the Semantic Web Layer Cake [2], the RDF layer lies immediately above the XML layer. It provides semantics to encoded metadata and resolves the circular references, an inherent problem of the hierarchical structure of XML.

We employ RDF for the graph data structure representation for the following reasons:

1. RDF does not allow creation of duplicate resources. Any attempt to create a duplicate resource, causes the model to discard the request and use the old created resource in its place. This feature greatly simplifies the problem of eliminating the duplicates from the XML representation. All requirement objects have unique ROINs. These ROINs are used to construct unique resources. So if the same requirement objects are encountered later in the XML file, RDF automatically discards the request of creating a duplicate resource and use the resource created earlier.

2. RDF facilitates bulk graph operations such as union and intersection of the graphs through its powerful Jena API. Thus, the framework helps in merging the updated requirement documents obtained from different sources like sub-contractors and other design teams. This scenario is especially helpful in circumstances where a couple of requirements are assigned to a particular group (such as a team or a sub-contractor) and, in turn, each team or sub-contractor defines their own sets of new requirements, or, incorporates the dependencies that they have on the requirements assigned to other teams.

3. The Jena API [8], which is used to manipulate RDF documents, provides RDQL (RDF Document Query Language) [13] to write powerful queries to access the various graph nodes based on their connectivity property. This capability is extremely useful in visualizing a particular sub-section of the requirement document. This topic is discussed in more detail in Section 1.5.

RDF Schema for Storage of Requirements Objects

The RDF schema to store the connectivity properties of requirement objects C and E (see Figure 1.3) is as follows:

```
<rdf:RDF
   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
>

<rdf:Description rdf:about='http://somewhere/C'>
    <vcard:N>C</vcard:N>
    <vcard:Given rdf:resource='http://somewhere/E'/>
</rdf:Description>

<rdf:Description rdf:about='http://somewhere/E'>
    <vcard:N>E</vcard:N>
</rdf:Description>
</rdf:RDF>
```

The first block of code defines XML namespaces that are utilized by the RDF statements (namespaces take care of name conflicts and enable shorthand notations for URIs).

```
xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
```

The `xmlns:rdf` namespace is the default RDF schema recommended by W3C. The `xmlns:vcard` is a simple RDF schema for properties about a person. The latter comes prepackaged with the vocabulary of the RDF API. For simple RDF models, the `vcard` schema can be utilized. As models become more complex, however, customized schemas and associated RDF API need to be written for the application's purpose.

The second and third blocks of RDF code contain statements about two requirement objects C and E in the requirement graph. Their labels are stored through `vcard:N` property, and the connection between the C and E is stored by `vcard:Given` property. Again, these two choices are made among a list of available properties in the vcard schema, which closely resembles the purpose for which it is used. As more requirement objects and connections between them are defined during the course of parsing the XML document, the above RDF code keeps on growing by adding new `rdf:description` tags for new objects and appending `vcard:Given` section under the existing defining requirement objects for specifying the connectivity.

RDF Representation of Requirements Objects

The representation of the requirement objects in RDF requires three triplets having the format (subject, predicate, object). This format constitutes an RDF statement and, in general, RDF document will contain many such statements/triplets. As a case in point, three RDF statements are contained in the snippet illustrated above:

```
1. (http://somewhere/C http://www.w3.org/vcard-rdf/3.0#N "C")
2. (http://somewhere/C http://www.w3.org/vcard-rdf/3.0#Given http://somewhere/E)
3. (http://somewhere/E http://www.w3.org/vcard-rdf/3.0#N "E")
```

The equivalent RDF graph representation is shown in Figure 1.5. This representation was obtained using W3C RDF graph validation services [18].

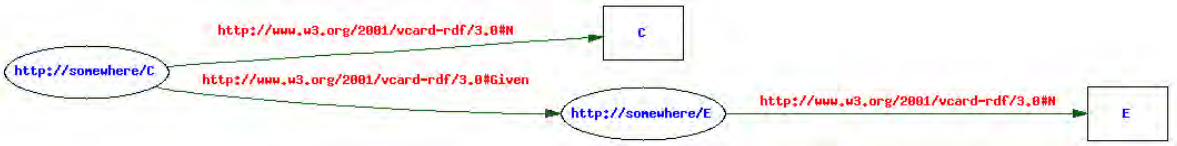

Figure 1.5: RDF Graph of the Requirement document having two nodes and one edge

1.4 Graph Layout using Hierarchical Graph Layout Algorithm

Once the graph of the requirement objects is constructed using RDF, the next step involves displaying these objects in a manner which can be easily comprehended by the end user. Over the past three decades, graph drawing technology has matured to the point where layout algorithms now exist for a wide range of problems and nodal topologies [5, 7, 9]. For our purpose, since the requirements document defines the objects in a hierarchical fashion, we chose to implement a modified graph layout algorithm proposed by Sugiyama [16].

This algorithm takes as its input, a directed acyclic graph. The algorithm output is a hierarchy of nodes organized into horizontal layers. Because requirements always flow down, the assumption of an acyclic graph (i.e., no loops) is automatically met by the input requirements document. This assumption is also met in SLATE because of the tree representation for requirements. As a safe guard, should a loop (accidentally) exist inside the graph, then it can be detected by observing the console output.

The modified Sugiyama algorithm processes the graph in four key steps:

1. Assigning the graph into different vertical layers to form a proper hierarchy. If the input graph contains a cycle, console output stays in this stage indefinitely.
2. Minimizing the edge crossings in the graph for readability.
3. Using priority layout heuristic to specify positions of the nodes within a particular layer.
4. Removal of dummy nodes which were inserted in phase 1, and replacing the nodes with bends in the line, thereby displaying the final graph.

The mathematical details of each step can be found in Appendix A. For detailed information about this algorithm, the interested reader is referred to Sugiyama [15].

PaladinRM Software and Requirements Visualization

We have implemented the requirements visualization software in a tool called PaladinRM – see Figure 1.6. Figures 1.7 and 1.8 show the screen shot of requirement documents 8-378 consisting of

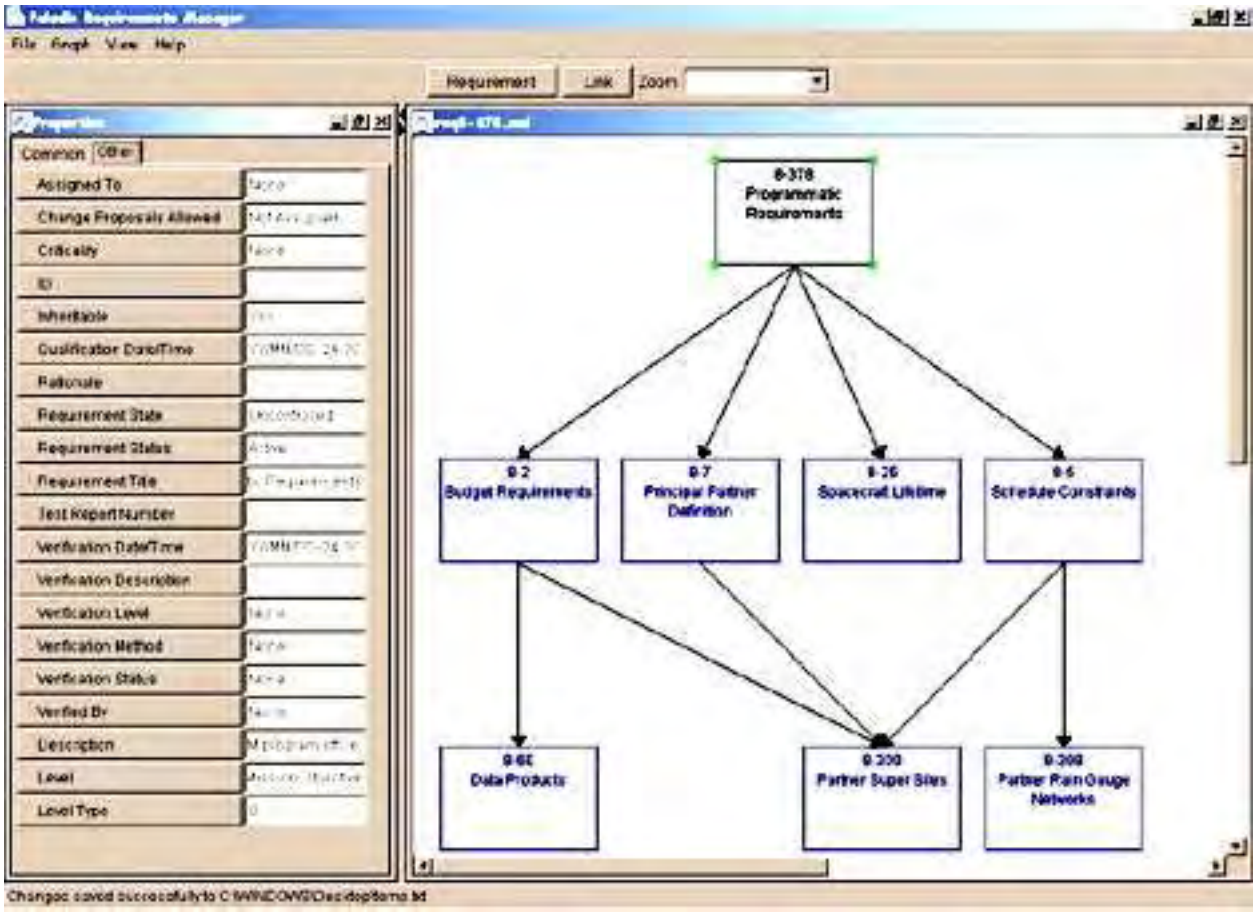

Figure 1.6: Screendump of the PaladinRM Graphical User Interface (GUI)

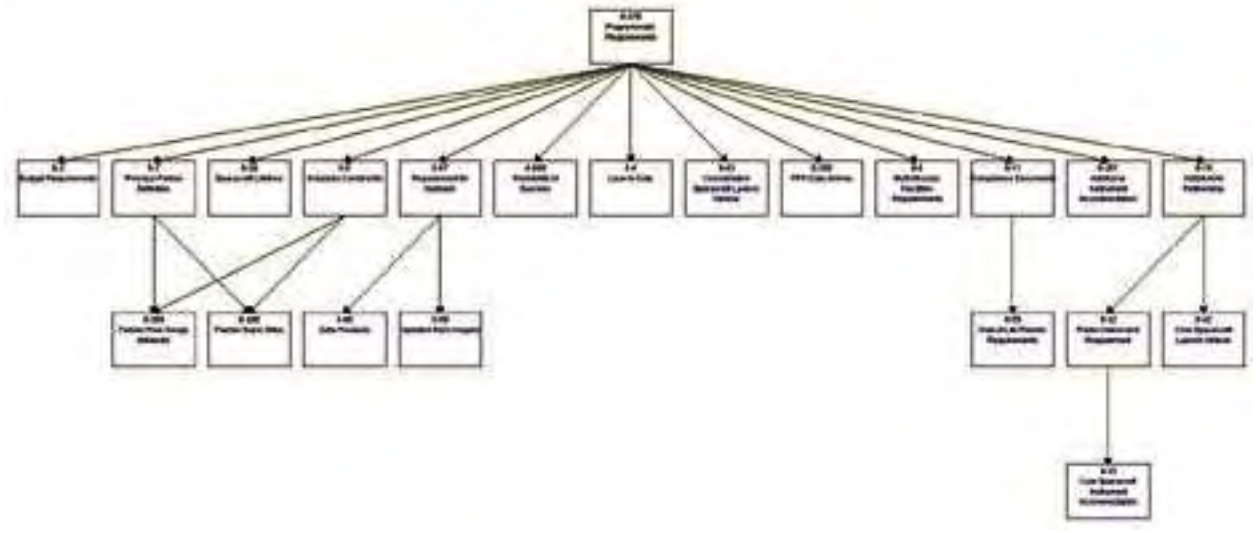

Figure 1.7: A simple requirement graph

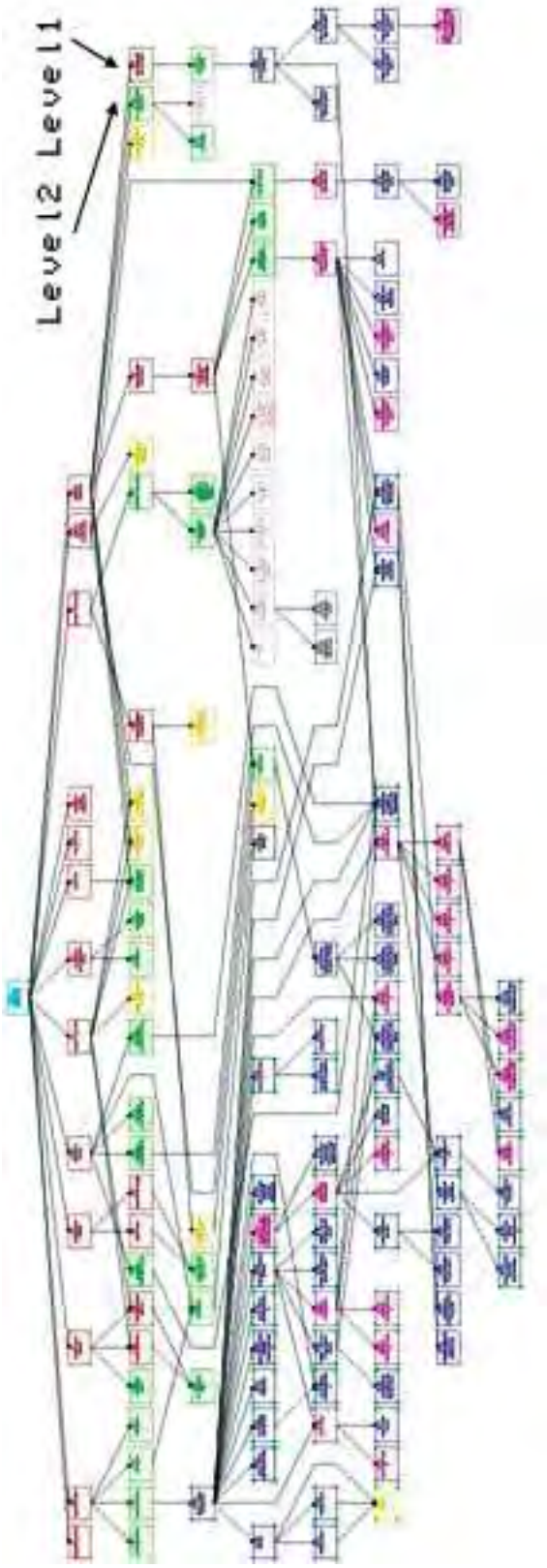

Figure 1.8: More complex requirement graph

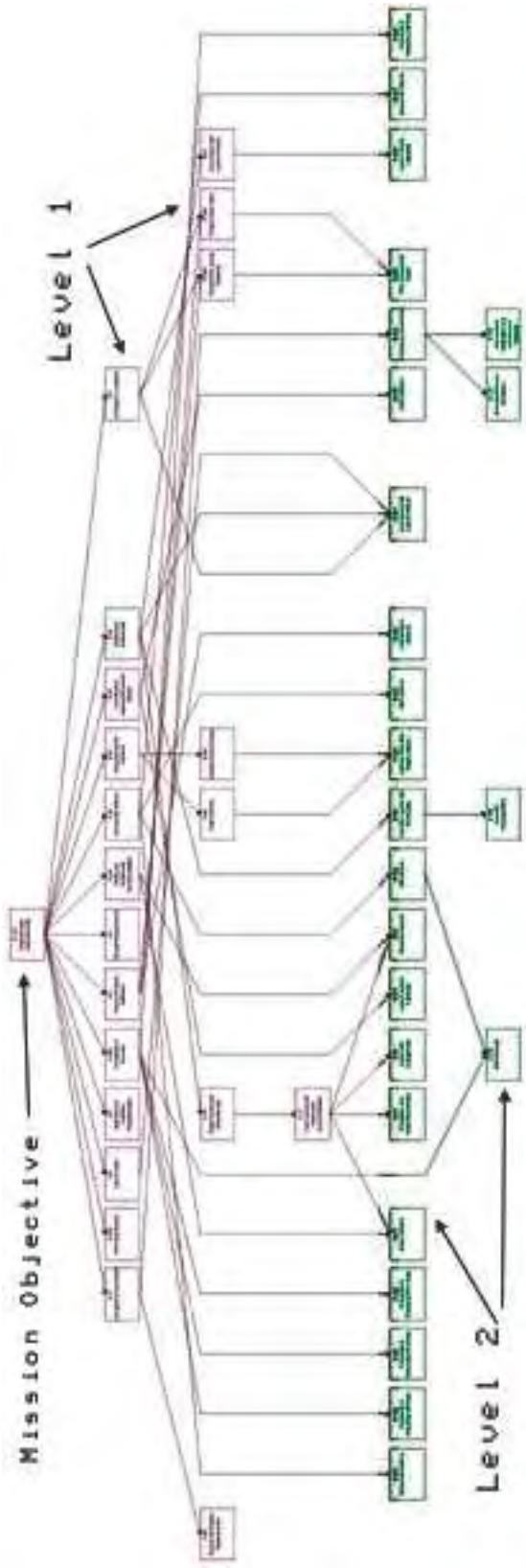

Figure 1.9: A Requirement graph with requirement clustered into different vertical groups based on their levels

a section showing only Level 1 requirements and the entire hierarchy for this requirement, respectively. Figure 1.9 shows another view of the same requirement graph with requirements clustered vertically based on their level assignments. PaladinRM has been tested on suites of requirements containing up to 1100 requirement nodes, which currently forms the entire set of GPM project requirements.

1.5 Visualizing Sub-Sections of Requirement Document

In the context of requirements engineering, traceability is about understanding how high-level requirements - objectives, goals, aims, aspirations, expectations, needs - are organized and transformed into low-level requirements. Understanding connectivity relationships among requirements is therefore roughly equivalent to understanding relationships between high- and low-level layers of information.

For models/documents containing hundreds of individual requirements, often cutting across multiple levels of organization, manual comprehension of the entire document is infeasible. In an effort to mitigate this bottleneck, in this work we provide a mechanism by which the end user can visualize parts of a document either selecting the attribute (Requirement type), or, by specifying the root requirement node of interest and the direction of visualization. The selection/visualization options are: "Up" to see the defining requirements, "Down" to see the complying requirements, and "Both" to see both defining and complying requirements. Each of these cases is illustrated in Figure 1.10.

Selective Visualization

Two types of selective visualization have been implemented in PaladinRM. This pathway of development is motivated, in part, by the observation that within the context of the NASA GPM project, containing thousands of requirements, individual persons and discipline-specific groups will be only be concerned with parts/sub-sections of the requirements document. For example, a mission system engineer may only worry about Mission objectives, Level 1 and Level 2 requirements. An electrical engineer may only be concerned with requirements at levels 4 and 5 relating to specific electrical components. To handle this range of needs, the XML requirements file contains the attribute type in the Req tag, describing the particular type of a requirement object. Filtering can be carried out on the basis of this attribute to see part of the requirement document of interest. In support of the hierarchy visualization technique, Figure 1.11 shows a level selection dialog box with Mission Objective and Level 1 requirement results selected. The filtered result is the requirements graph shown in Figure 1.7.

Directed Visualization

PaladinRM also provides users with the ability to select and visualize the complying and defining requirements emanating from/to a particular root requirements node (i.e., a local viewpoint of the requirements document). The software includes an option for specifying the number of levels to trace from the root requirement node – this feature reflects the observation that requirement hierarchies can be very deep and nested. The screendump in Figure 1.12 shows the first part of this process, where users select the number of levels and direction of visualization. Figure 1.13 shows relevant parts of the larger requirements document (i.e. see Figure 1.8) to which the query applies.

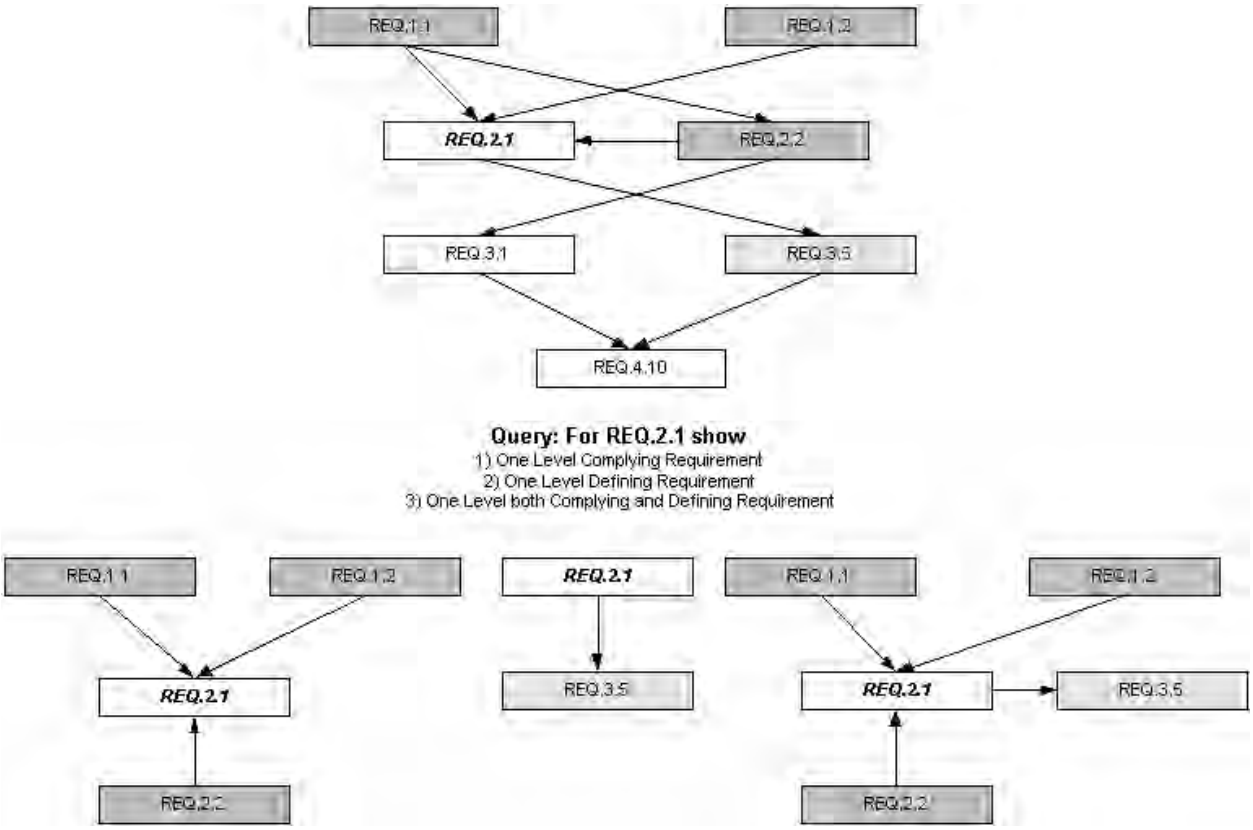

Figure 1.10: Extraction and Visualization of "Complying" and "Defining" Requirements in the Neighborhood of Requirement 2.1

Figure 1.11: A Level Selection Dialog to Visualize Parts of the Requirement Document

Figure 1.12: Dialog Box to Choose the Viewing Direction (Down/Up/Both) and the Number of Hopping Steps

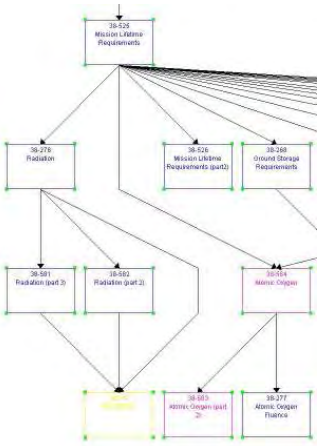

Figure 1.13: Requirement graph with nodes highlighted which satisfy the level selection

1.6 RDQL Approach to Retrieve Nodes and Links

RDQL [13] is a query language designed for RDF in Jena [8] models. A meta-model specified in RDF consists of nodes (which could be either literals or resources) and directed edges. RDQL provides a way of specifying a graph pattern that is matched against the graph to yield a set of matches. In this framework we have requirements (nodes in the RDF meta-model) that are connected by the direct edges specifying the relationship of complying and defining requirements. The originating node of the link specifies a defining requirement and the terminating node defines a complying requirement.

The upper half of Figure 1.10 shows a graph of requirements organized into four layers. Complying and defining relationships are interleaved among the requirements. We want to see a controlled visualization of the complying and defining requirements with respect to REQ.2.1. Expected results are shown for the required query at the bottom of the figure. The equivalent RDF model for the entire requirement document is illustrated in Figure 1.14.

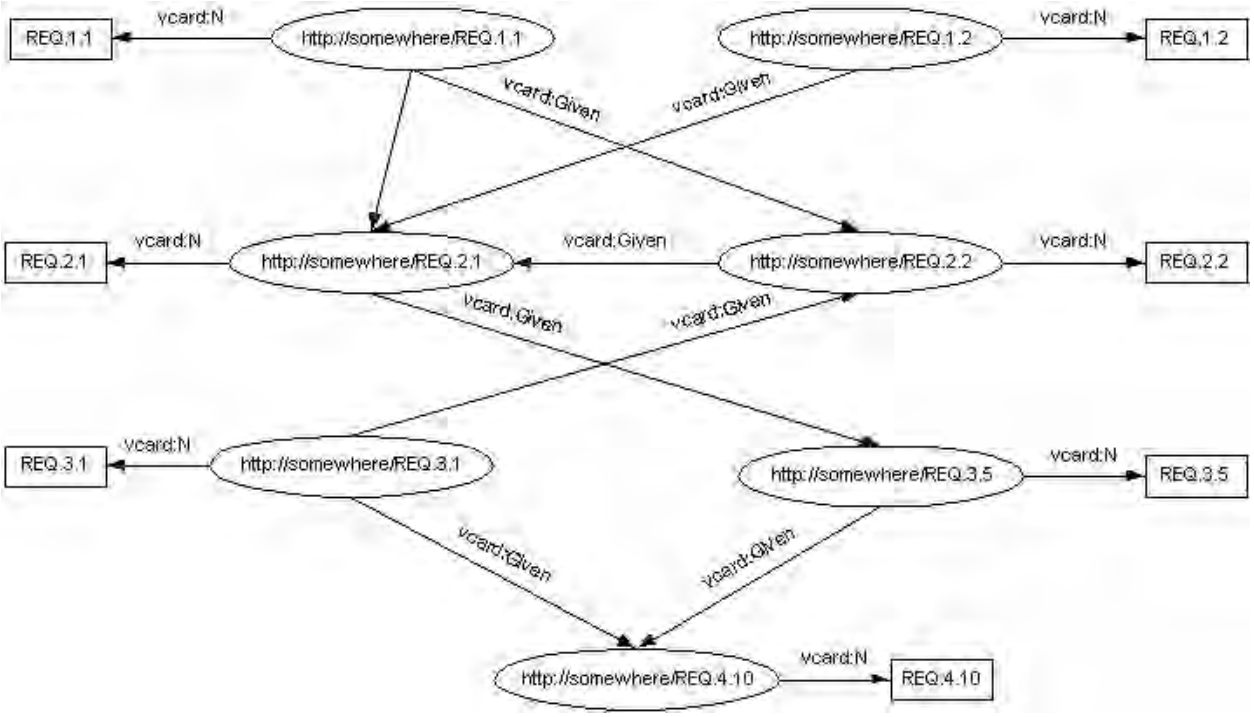

Figure 1.14: Equivalent RDF model of the Requirement Graph shown in Figure 1.10

RDQL works by executing the string queries, which are passed to a query engine. The query engine looks at the structure of the query and pattern of the query is matched against all the triplets in the RDF file on which the query is running. It returns an iterator of the result set which can be inspected to retrieve the desired result.

Query for Complying requirements One Level Down:

The Query string to see the complying requirements is as follows:

```
String queryString = "SELECT ?X " +
"WHERE( <http://somewhere/"+currentElement+">,
        <http://www.w3.org/2001/vcard-rdf/3.0#Given>, ?X)";
```

The current element is the REQ.2.1 from which we want to see the complying requirements. ?X represents a clause which returns the resources satisfying the given property.

Query for Defining requirements One Level Up:

The Query string to see the defining requirements is as follows:

```
String queryStringLevelUp = "SELECT ?X " +
"WHERE(?X, <http://www.w3.org/2001/vcard-rdf/3.0#Given>,
           <http://somewhere/"+currentElement+"> )";
```

Query for both Complying and Defining Requirements around One Level:

The Query string to see both complying and defining requirements around one level is obtained by a combination of above two queries executed together. A multiple-level query can be recursively executed on all the obtained results until it reaches the number of level, or a leaf requirement, whichever occurs earlier.

1.7 Differential Update of the Requirement Database

We also need a mechanism for feeding modifications back to the originating requirements document. Because requirements are visualized as a block diagram hierarchy, a systems engineer can easily identify by inspection the flawed links or missing links between requirement nodes. These links can be modified within PaladinRM. PaladinRM provides has the capability of storing the initial state of the graph, comparing it with the final state, and then automatically generating a list of modifications to a text file. This text file can be read back using a script, and used to update the requirements database.

1.8 Annotating Requirement Nodes with Attributes

Requirement management tools use the tree based representation to show the attributes of a particular requirement node. Also, they provide powerful query mechanism to query requirement nodes based on the input attribute range. This approach has its own advantages and shortcomings. While it provides a intuitive way to investigate a particular requirement, and generate a report based on the attributes, it falls short in identifying the bottleneck requirements. In real world scenarios, it often happens, that a particular requirement is affecting a whole set of requirements beneath it. Unless this requirement is properly resolved, many other requirements remain pending.

In the block diagram representation of requirements, we have added two attributes which can be selected by the end-user and displayed along with the requirements title and ROINs. See, for example,

Figure 1.15: Property Dialog Box to Choose the Properties to be Displayed on the Requirement Objects

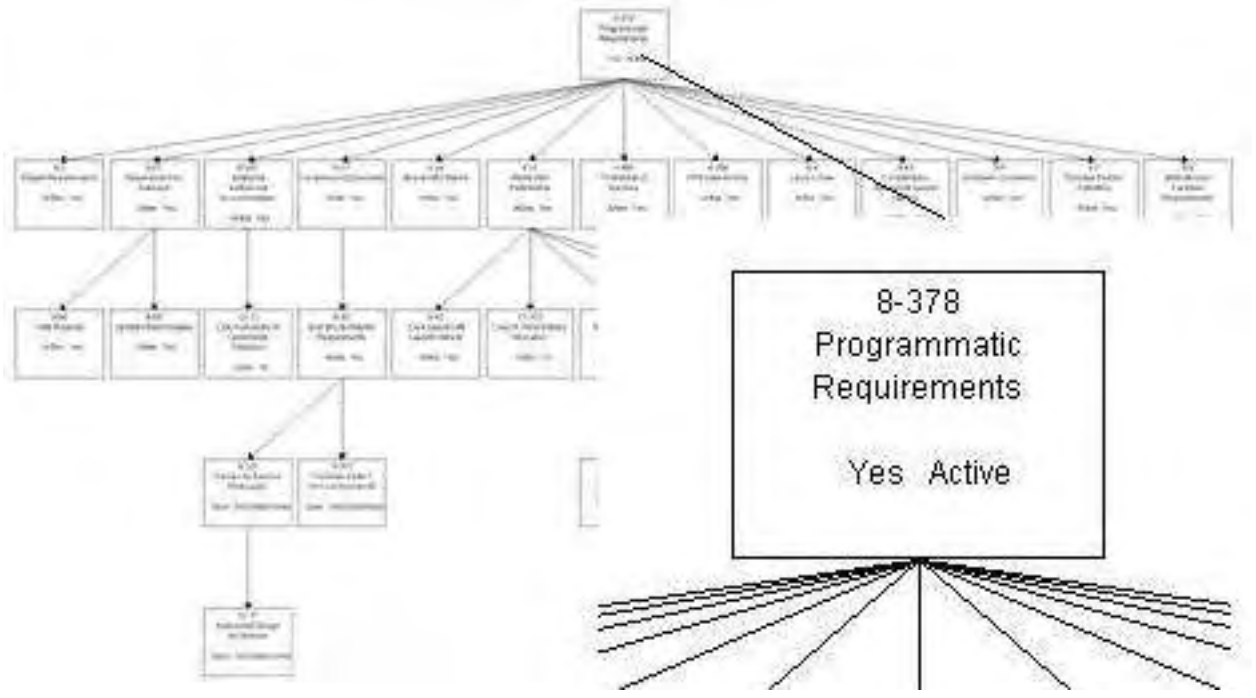

Figure 1.16: Requirements Graph with Nodes Annotated with Properties Chosen

Figures 1.15 and 1.16. It is evident that by looking at this block diagram, bottleneck requirements can be easily identified. This visualization mechanism, along with an ability to visualize subsections of requirements documents, provides a very powerful mechanism for querying and displaying requirements in a way that makes sense to systems engineers. First a systems engineer can select which requirements (Level 1, Level 2 etc.) he/she wants to view, and then select the attributes that need to be displayed on those requirement blocks.

1.9 Acknowledgements

This work has been supported, in part, by an educational grant from the Global Precipitation Measurement Project at the NASA Goddard Space Flight Center, and a research grant to Vimal Mayank from the Systems Engineering Group at NASA Goddard. We particularly wish to thank David Everett and Tom Philips at NASA Goddard for their input to the systems engineering and software development phases of this project. The views expressed in this report are those of the writers and are not necessarily those of the sponsors.

1.10 Conclusions and Future Work

PaladinRM does not strive to compete with commercially available products for requirement management. Instead, it provides easy-to-use mechanisms for visualizing requirements documents in way that can support decision making in systems engineering. The directions for future work are as follows:

1. AP233 [10, 11] is an emerging standard for systems engineering data exchange among vendor tools such as SLATE, DOORS, Rational Requisite PRO, and CORE [1, 3, 4, 14]. Once AP233 is fully developed and adapted by various vendors, the next step will be to update our XML encoding for requirements representation and traceability so that it is AP233 compliant. We will then be able to import data from other tools and represent and manipulate it in our GUI. The current API written to manipulate the AP233 documents is OLE API. There have been discussions with the AP233 group to provide a open-source Java API for the same, so that it could be used by this tool to read and write AP233 documents.

2. A framework for mapping system requirements onto system structure and behavior will complement the systems engineering process of developing complex real time systems. Presently, SLATE achieves this functionality through the abstraction blocks stored in a different hierarchy folder and connected to the requirement document via links. A better visualization framework, perhaps along the lines of the emerging SysML standard, is needed to map system structure (and other UML diagrams) onto system requirements. Mechanisms of this type will allow for the identification and acceleration of modular development of the systems by mapping chunks of system core components onto a set of requirements, and separating and assigning them to a particular team or sub-contractor.

Bibliography

[1] SLATE. See http://www.eds.com/products/plm/teamcenter/slate/. 2003.

[2] Berners-Lee, T., 2000. XML and the Web. Keynote address at the XML World 2000 Conference.

[3] CORE. See http://www.vitechcorp.com/productline.html. 2003.

[4] Dynamic Object Oriented Requirements System (DOORS). See http://www.telelogic.com/products/doorsers/doors/. 2003.

[5] Eades P, and Tamassia R. Algorithms for Drawing Graphs: An Annotated Bibliography. *Technical Report CS-89-09*, February 1989.

[6] Global Precipitation Measurement Project. See http://gpm.gsfc.nasa.gov/index.html. 2003.

[7] Herman I. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, January-March 2000.

[8] Jena - A Java API for RDF. See http://www.hpl.hp.com/semweb/. 2003.

[9] Kamada T., and Kawai S. A General Framework for Visualizing Abstract Objects an Relations. *ACM Transaction on Graphics*, 10(1), January 1991.

[10] Muller D. Requirements Engineering Knowledge Management based on STEP AP233. 2003.

[11] Oliver D. AP233 - INCOSE Status Report. *INCOSE INSIGHT*, 5(3), October 2002.

[12] Resource Description Framework (RDF). See http://www.w3.org/RDF. 2003.

[13] RDF Data Query Language (RDQL). See http://www.hpl.hp.com/semweb/rdql.htm. 2003.

[14] Rational Rose. See http://www.rational.com/products/rose/. 2003.

[15] Sugiyama K. *Graph Drawing And Applications For Software And Knowledge Engineers.* World Scientific, Singapore, 2002.

[16] Sugiyama K., Tagawa S., and Toda M. Methods for Visual Understanding of Hierarchical Systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2):109–125, January 1981.

[17] Microsoft Visio. See http://www.microsoft.com/office/visio/. 2003.

[18] W3C RDF Validation Service. See http://www.w3.org/RDF/Validator. 2003.

[19] Microsoft Word. See http://www.microsoft.com/office/word. 2003.

[20] eXtensible Markup Language (XML). See http://www.w3.org/XML. 2004.

Appendices

Appendix A. Hierarchical Layout Algorithm Details

The methodology for drawing requirements hierarchy is adapted from the algorithm proposed by Sugiyama, Tagwa and Toda [16]. This heuristically-driven algorithm is a hierarchical drawing method to draw acyclic directed graphs. Requirement nodes are placed on horizontal lines (called layers or vertical levels), and edges joining vertices are distributed as polylines with bends on the horizontal lines. The algorithm consists of four phases:

1. Making the general directed graph acyclic.

2. The assignment of vertices to layers in the acyclic directed graph.

3. The determination of the order of vertices on each layer.

4. The determination of the position of vertices on each layer.

The details of each step are as follows:

Step 1. Making General Directed Graphs Acyclic

In this work, the input requirement XML files have been provided from the NASA - GPM project. Each file contains a perfect hierarchy of requirements, without feedback edges. Hence, this step is omitted from the implementation.

If, however, this graph happened to contain (accidentally) a cycle, Step 2 of the implementation would execute for an infinite amount of time; the output message would confirm the existence of a loop. The problem of identifying the smallest number of feedback edges is a NP-complete problem. Several heuristic like depth first search, largest outdegree and the divide and conquer have been proposed which identify the feedback edges.

Step 2. The Assignment of Layers in Acyclic Directed Graphs

In this step the vertical level of each requirement node is calculated. This is done to layout the entire graph in one direction (top-down) with the node having no incoming edge (source or root node) being at the top and rest of the nodes fanning out progressively downwards. This step partitions the entire directed graph $G = (V, A)$ into $V_1, V_2, \ldots, V_h$ such that $i < j$ when $(u, v) \in A, u \in V_i \text{and} v \in V_j$. This means the source node is always placed higher than the target node for any edge belonging to the graph. For edge $e = (u, v), u \in V_i, v \in V_j$, the span of edge e is $s(e) = j - i$; In this step, after assigning the nodes to the vertical layers, the hierarchical graph is transformed into a proper hierarchical graph. That is, for the long edge (u, v) joining vertex u on the ith layer and vertex v on the jth layer, by adding dummy vertices $v_1, v_2, \ldots, v_{j-i-1}$, the edge (u, v) is replaced by path $u = v_0 - > v_1 - > \ldots - >$

$v_{j-I} = v$. This is because it is difficult to handle crossings of long edges and from Step 3 onwards span of all edges are one (i.e., the graph is assumed to be proper). It is desirable to have as few dummy vertices as possible to reduce machine computation time, reduce the number of bends that occur at dummy vertices and easier for a human being to follow short edges rather than long edges.

Step 3. The determination of the order of vertices on each layer

The step achieves minimization of edge crossings. The number of edge crossings in a drawing of a proper hierarchical graph is not dependent on the exact position of vertices, but solely on the order of vertices within each layer. Thus, the problem of minimizing the edge crossings is a combinatorial problem of choosing the appropriate order for each layer, and not a geometrical problem of choosing the x coordinate of each vertex.

Preprocessing

Suppose that $G = (V, A, h)$ is a proper hierarchical graph of height h. For G, the partition of V is expressed as $V_1, V_2, \ldots, V_h$, and of A as $A_1, A_2, \ldots, A_{h-1}$ (A_i is a subset of $V_i X V_{i+1}$). Now, if an order σ_i for all elements in each layer V_i is given, then G is called h layer graph and is written $G = V, A, h, \sigma)$. Here, $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_h)$.

A matrix realization is used for the h layer graph $G = (V, A, h, \sigma)$. The matrix $M^{(i)} = M(\sigma_i, \sigma_{i+1})$ represents A_i as a $\|V_i\|X\|V_{i+1}\|$ matrix, where the rows representing the vertices of V_i are ordered from the top by σ_i, and the columns are ordered by the order σ_{i+1} on the vertices of V_{i+1}. The (k, λ) element $m_{kl}^{(i)}$ of $M^{(i)}$ is given by:

$$m_{kl}^{(i)} = \begin{cases} 1 & \text{if } (v_k, v_l) \in A_i; \\ 0 & otherwise. \end{cases}$$

$M^{(i)}$ is called the incidence matrix. When the incidence matrices are arranged in order this is called the matrix realization ζ of the h layer graph, and is written as:

$$\zeta(\sigma_1, \ldots, \sigma_h) = M^{(i)} \ldots M^{(h-1)}$$

The row vector $r(v)$ corresponding to the vertex v of the incidence matrix $M^{(i)}$ expresses the incidence relationship of vertex v of the ith layer and vertices of the $i+1$th layer. When p and q are the number of vertices of the ith layer and the $i+1$th layer, the number c of crossings of the edge between the jth and the kth vertices v_j and v_k of the ith layer and vertices of the $i+1$th layer is given by:

$$c(r(v_j), r(v_k)) = \sum_{\alpha=1}^{q-1} \sum_{\beta=\alpha+1}^{q} m_{j\beta}^{(i)} m_{k\alpha}^{(i)}$$

Therefore the number of crossings between the ith layer and the $i+1$th layer is given by:

$$C(M^{(i)}) = \sum_{j=1}^{p-1} \sum_{k=j+1}^{p} \left(\sum_{\alpha=1}^{q-1} \sum_{\beta=\alpha+1}^{q} m_{j\beta}^{(i)} m_{k\alpha}^{(i)} \right)$$

Therefore the total number of crossings $C(\zeta)$ of ζ can be found by:

$$C(\zeta) = C(M^{(i)}) + \ldots + C(M^{(h-1)})$$

The row barycenter γ_k and column center ρ_l of incidence matrix $M = (m_{kl})$ are defined as:

$$\gamma_k = \sum_{l=1}^{q} l.m_{kl} / \sum_{l=1}^{q} m_{kl}$$

$$\rho_l = \sum_{k=1}^{p} k.m_{kl} / \sum_{k=1}^{p} m_{kl}$$

Reduction of Crossings in a 2-Layer Graph by the Barycenter Method

In the barycenter method, for the matrix realization $\zeta = M(\sigma_1, \sigma_2)$ of a 2 layer graph, the order of colums (rows) is fixed, and when rearranging the order of rows (columns) the row (column) barycenters are calculated and arranged in monotonically increasing order (in the case of equal barycenters, the initial order is preserved). When $M(\sigma_1', \sigma_2)$ is obtained by rearranging the columns of $M(\sigma_1, \sigma_2)$, this process is called row barycenter ordering and written as BOR(M). The column barycenter ordering, BOC(M), is defined in the same way. By repeatedly alternating row and column barycenter ordering, a reduction in the number of crossings can be achieved.

The algorithm consists of PHASE 1 and PHASE 2. IN PHASE 1 the alternating row/column barycenter ordering is carried out for a pre-assigned number of iterations(or until the number of crossing ceases). IN PHASE 2, if there are any rows or columns of equal barycenters left at the end of PHASE 1, the order of these rows or columns are reveresed by units of equal barycenter groups, and PHASE 1 is carried out. Therefore PHASE 2 contains PHASE 1 as a sub-algorithm. IN PHASE 2, the operation to reverse the order of equal barycenter rows (columns) for M is written ROR(M) (ROC(M)).

For h layer graphs as for 2 layer graphs, the barycenter ordering can be applied repeatedly consecutively to each layer. That is, first the ordering of layer V_1 is decided, then for $i = 1, 2, 3, \ldots, h-1$, the order of layer V_i is fixed and the order of layer V_{i+1} is decided such that the number of crossings between layer V_i and layer V_{i+1} is reduced. This is called the downward pass; conversely going from layer V_h to layer V_1 is called the upward pass. The algorithm is the same as for 2-layer graphs.

Step 4. Decision of the Layout Position of Vertices on Each Layer

From the crossing number reduction methods of the previous section, the layout order of vertices σ^* with few crossings is determined. To decide the placement of vertices in this step, this σ^* is taken as input, and the aim is to achieve the four drawing rules of least separation, closeness, balance, straight lines. This can be achieved using either the quadratic programming method, or using a heuristic namely priority layout method. In this work priority layout method is used to cut down on the machine computing time. The basic concepts are similar to the multi-layer barycenter method, and improvement steps for the layout coordinates of vertices on each layer of a multi-layer graph are consecutively repeated across

all layers from top to bottom, bottom to top. The name derives from the use of priority for each vertex in the improvement of vertex layout coordinates in each layer. The outline of algorithm is as follows:

1. The initial coordinates of each vertex (the kth vertex of the ith layer) of V are given by $x_{ik} = x_o + k$. Here x_o is a given integer. This integer is assumed to be zero in this work, as this merely defines a x offset of the graph drawing. K is also termed as the horizontal level of the node, for that particular vertical level. These vertical and horizontal levels constitute a grid like structure and the nodes are placed finally at the grid points.

2. Repeating highest layer $->$ lowest layer $->$ highest layer $->$ lowest layer, down twice and up once, the following are carried out for each vertical layer (the ith layer).

2.1. For V_i, when down (up) the priority of the real vertices are given by their Up (Down) total degree (achieving balance). Also the highest priority is given to dummy vertices (achieving straightness of the long edge).

2.2. For V_i, the improvement of the layout coordinates x_ik of each vertex (the kth) is carried out in order of highest priority according to the following.

2.2.1. When Down (Up), the kth vertex is, fulfilling the conditions below, brought as close as possible to the Up (Down) barycenter.

(1) The layout coordinates of vertices are limited to integer values. Also a vertex cannot have the same coordinate values as another vertex (achieving least separation).

(2) The layout order of vertices must not be changes (achieving minimization of number of crossings).

(3) In order to be brought closer to the Up (Down) barycenter, the only vertices, which can be moved are those with lower priority than the object of improvement, the kth vertex (achieving closeness).

For a simple example that illustrates the above algorithm step by step please refer to [15]. The output of the above 4 steps produces a graph with placement of each requirement node, including dummy vertices introduced in Step 2. The final drawing can be obtained by replacing all the dummy nodes with polylines, such that these dummy vertices act as intermediate points.