ABSTRACT

Title of thesis: **SYSTEMS ENGINEERING DESIGN AND TRADEOFF ANALYSIS WITH RDF GRAPH MODELS**

Nefretiti N. Nassar, Master of Science, 2012

Thesis directed by: Associate Professor Mark Austin
Department of Civil and Environmental Engineering and ISR

As engineering systems become increasingly complex the need for automation arises. This thesis proposes a multi-level framework for design of a home theater system cast as a component-selection design problem. It explores the extent to which the resource description framework (RDF) and Python can be used in a software pipeline for systems engineering design and trade-off analysis. The software pipeline models and visualizes RDF graphs, implements inference rules for the step-by-step selection of design component combinations that satisfy system requirements, identifies non-inferior Pareto-Optimal design solutions, and tracks the size of the RDF graphs during execution of the pipeline. The use of RDF and Python for automation provides a simplified replacement for present-day Semantic Web tools and technologies.

Last Modified: November 30, 2012

SYSTEMS ENGINEERING DESIGN AND TRADEOFF ANALYSIS WITH RDF GRAPH MODELS

by

Nefretiti N. Nassar

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science in Systems Engineering
2012

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible. I would like to first thank my beautiful family including my mother, Shaba, and Ben for their support – it has been overwhelming. I would like to thank my best friend, Whitney Ford, for encouragement and teaching me concepts of computer science. I would also like to thank my dearest friend, Richard Afoakwa, for enlightenment and helping me regain my passion for engineering. Finally, I would like to express my deep gratitude to my advisor, Dr. Mark Austin, for facilitating this learning experience and challenging me to become a better engineer.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

Modern-day system designs are undergoing a series of radical transformations to meet performance, quality, and cost constraints. To keep the complexity of technical concerns in check, system-level design methodologies are striving to orthogonalize concerns (i.e., achieve separation of various aspects of design to allow more efficient exploration of the space of potential design alternatives), improve economics through reuse at all levels of abstraction, and employ formal design representations that enable early detection of errors and multi-disciplinary design rule checking.

Solutions to these challenges are hindered by the multi-disciplinary nature of team-based systems and the diverse needs of professional systems engineers. Figure 1.1 shows, for example, a hypothetical situation where high-level project requirements are organized for team development, and project requirements are imported from external sources, in this case, the Environmental Protection Agency (EPA). 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

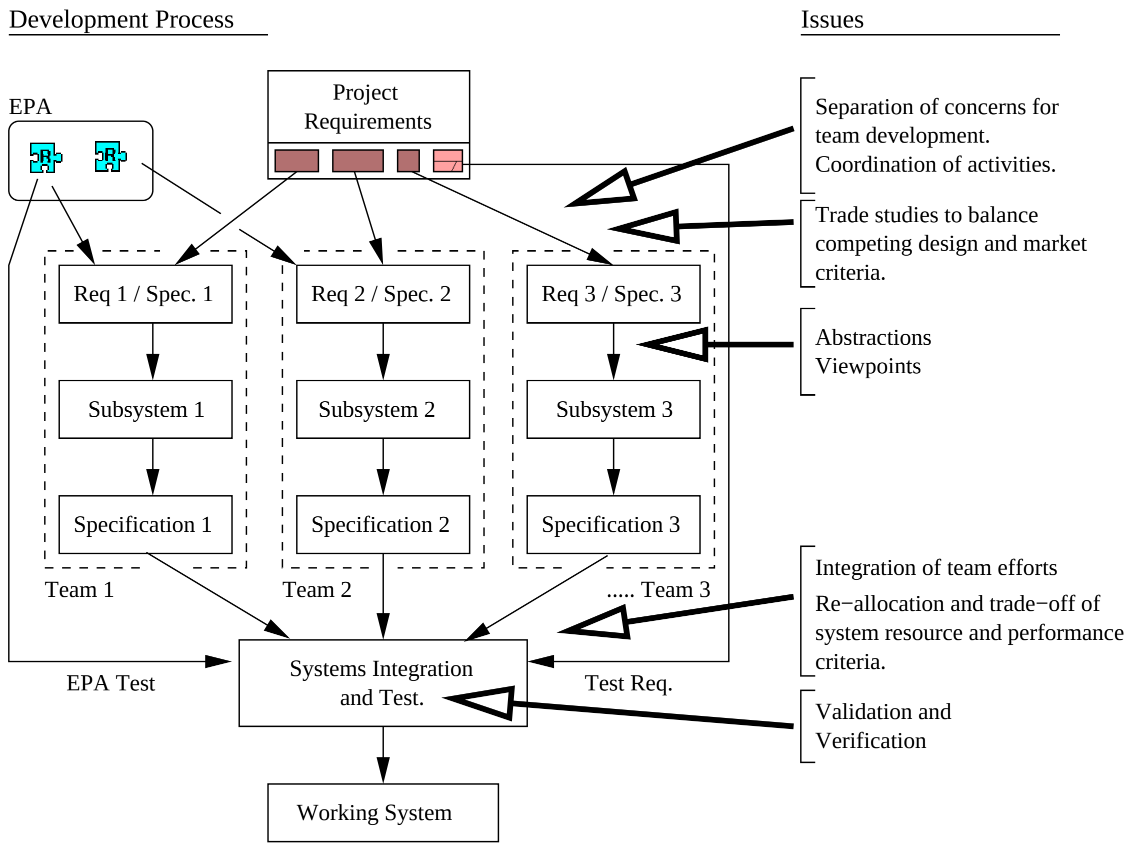

Development Process
Issues
EPA
Project Requirements
Req 1 / Spec. 1
Req 2 / Spec. 2
Req 3 / Spec. 3
Subsystem 1
Subsystem 2
Subsystem 3
Specification 1
Specification 2
Specification 3
Team 1
Team 2
..... Team 3
EPA Test
Systems Integration and Test.
Test Req.
Working System
Separation of concerns for team development.
Coordination of activities.
Trade studies to balance competing design and market criteria.
Abstractions
Viewpoints
Integration of team efforts
Re−allocation and trade−off of system resource and performance criteria.
Validation and Verification

Figure 1.1: Development process and key issues in the team-based development of engineering systems.

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. Systems engineers are responsible for trade studies to find a good balance in competing (design and business) criteria. These studies will be based on measures of effectiveness for system performance and market needs. Constraints will be derived from requirements. Together the measures of effectiveness and constraints will establish a solution space within which trade studies can be conducted.

As engineering systems become increasingly complex the need for automation arises, as a means of maintaining designer productivity and ensuring timely development of systems. Two key elements of required capability are an ability to: (1) Identify and manage requirements during the early phases of the system design process, where errors are cheapest and easiest to correct, and (2) Automatically synthesize large systems from smaller (simpler) systems. A central tenet of our work is that methodologies for strategic approaches to design will employ semantic descriptions of application domains, and use ontologies to enable validation of requirements problem domains, and communication (or mappings) among multiple disciplines. Present-day systems engineering methodologies and tools are not designed to handle projects in this way.

1.2 Literature Review

The proposed research is targeted towards the formal representation and management of requirements, and automated selection of components. We assume a component specification method has been chosen and a component library has been designed and built. Previous mathematical models utilize mathematical logic and techniques for system verification [8]. However, we seek to describe models in a formal way at a high level that we believe will lead to automation.

1.2.1 Top-Down and Bottom-Up Approaches to Design

Systems engineering methodologies are the confluence of top-down and bottom-up approaches to system development.

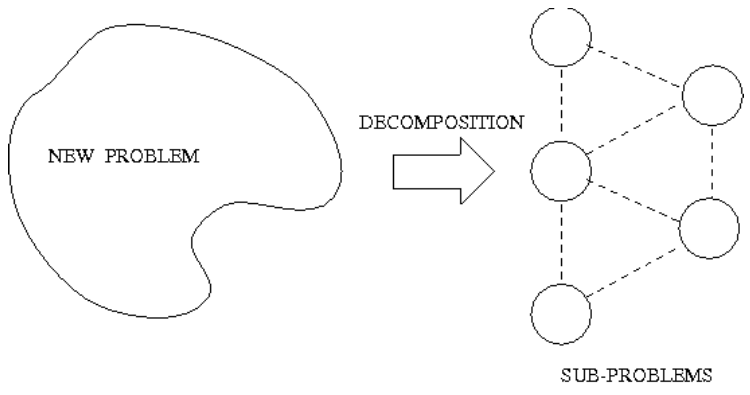


Figure 1.2: Top-down decomposition of systems.

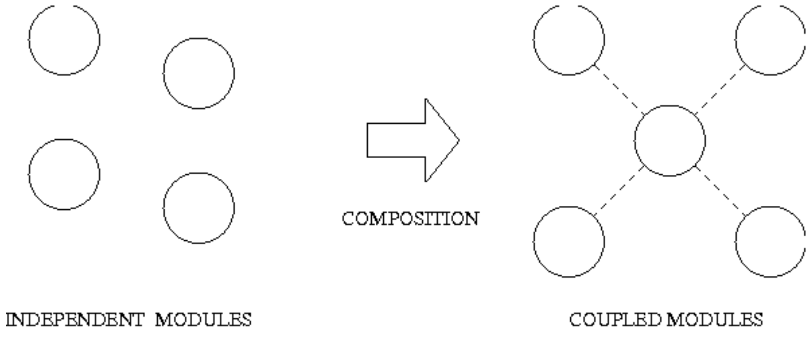


Figure 1.3: Bottom-up composition of systems.

Top-down design (decomposition). In a top-down strategy of development, a new design problem is simplified by decomposing it into a network of simpler sub-problems. See Figure 1.2. Top-down strategies of development allow for customized solutions – that is, you can create a system containing only the features that are needed. The main disadvantages of top-down development are increased development time and increased need for testing of new sub-systems. Also, the result of a top-down design is modules that are of a one-time-only form – they are not as easily reused because they were components without a preconceived vision of their future use.

Bottom-up development (synthesis). In a bottom-up strategy of development,

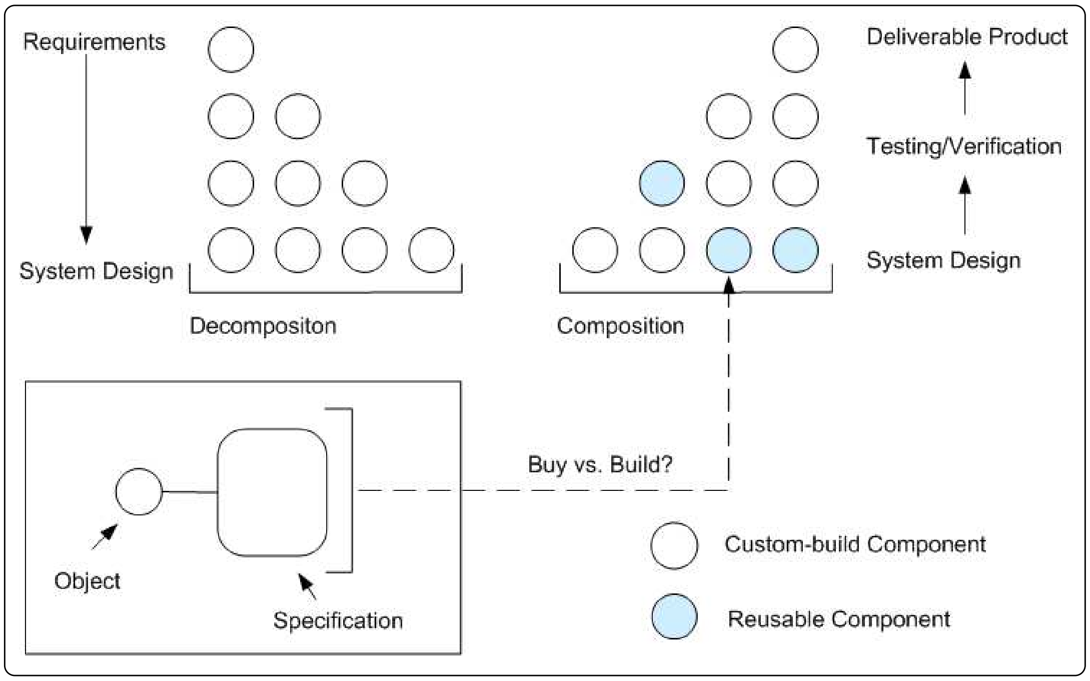


Figure 1.4: V model of system development – top-down decomposition (design) followed by bottom-up composition (implementation).

new design solutions (higher-level entities) are created through the synthesis (or composition) of independent modules. See Figure 1.3. The benefits of reuse include reduced development costs, improved quality (because components have already been tested), and shortened time-to-market. The main disadvantage of bottom-up development is that the system may contain many features not needed to solve a specific task.

Balancing top-down and bottom-up development concerns. An engineer should never set out to build a system without first considering available modules and/or components. Conversely, designers never create an engineering system without a preconceived vision of its future use. A balance of these criteria is usually needed and desirable. Figure 1.4 shows the role of top-down decomposition and bottom-up synthesis of components in a V-Model of system development. If components can be described by a formal specification, then in principle, an engineer ought to be able to examine the specification to see if it is capable of satisfying the design requirements.

1.2.2 Component Selection Design Problem

The component selection design problem can be stated as follows: We wish to choose a subset of components from a library of components to satisfy the requirements of a pre-specified system architecture, and component- and system-level requirements. As illustrated in Figures 1.4 and 1.5, we will assume that a representation for component specification exists, and that a component library has been designed an built.

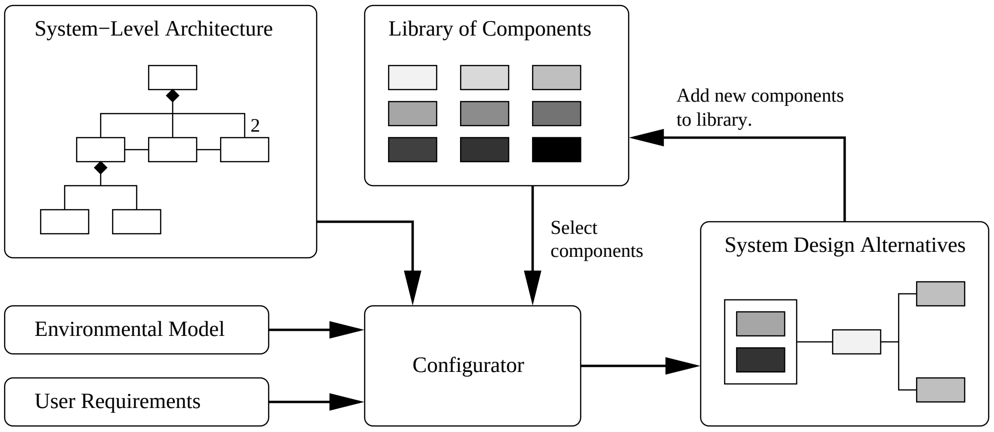


Figure 1.5: Schematic of the component-selection design problem.

Generally speaking, two outcomes to the component selection problem are possible. The first possibility is that a search procedure will find one or more combinations of components that satisfy all of the architectural, functionality and performance requirements. In such cases, we will choose a subset of feasible designs that maximize performance, minimize cost, and so forth. The second class of outcomes occurs when the design requirements are stated in such a way that no feasible designs exist. This problem can be solved by either relaxing the requirements (i.e., values on the inequality constraints representing the requirements), or by developing new components that will have superior performance and/or extended functionality.

Relationship to Assignment-Type Problem. Component selection is a specific example of the assignment-type problem (ATP). That is, given N items and M resources, devise an assignment of items to resources such that a given cost function is optimized and "K" restrictions are satisfied [11]. The mathematical representation of ATPs is:

$$\text{Minimize Objective } F(x) \tag{1.1}$$

$$\text{subject to } \sum_{j \in J_j} x_{ij} = 1, 1 \leq i \leq N, \tag{1.2}$$

$$G_k(x) <= 0, 1 \leq k \leq K, \tag{1.3}$$

where

$$x_{ij} = \begin{cases} 1 & \text{if item i is assigned to resource j,} \\ 0 & \text{otherwise.} \end{cases} \tag{1.4}$$

and $J_j = \{1, 2, ...M\}$ is the set of admissible/allowable resources for item i, $1 \leq$

$i \leq N$. The family of functions G(x) are the imposed constraints. Together the assignment constraints and decision variable constraints indicate that item i has to be assigned to exactly one resource j. The objective function and inequality constraints do not need to follow a special format, other than being calculatable.

Procedures for Component Selection. The design of efficient strategies for component selection is an active area of research [5, 6, 11, 14]. Outstanding problems include:

1. The design of algorithms to satisfy families of requirements with a minimal number of components (i.e., implying an integrated systems solution) and,

2. The design of algorithms to find approximate, but good, solutions to the component selection problem when exhaustive search of the design space is computationally infeasible.

Our research objective is to devise strategies for component selection during the early stages of design, where the number of components is unlikely to be excessively large. Figure 1.6 illustrates the essential features of a trial-and-error approach to component selection, where components are selected from a database for the implemention of a system architecture. For the purposes of illustration, let us assume that the system architecture requires three types of components: solid rectangles, hashed rectangles, and circles. Within the database, variations in the details of component implementation (e.g., size of the rectangle) will lead to variations in system performance and cost. Key points in the trial-and-error procedure are as follows:

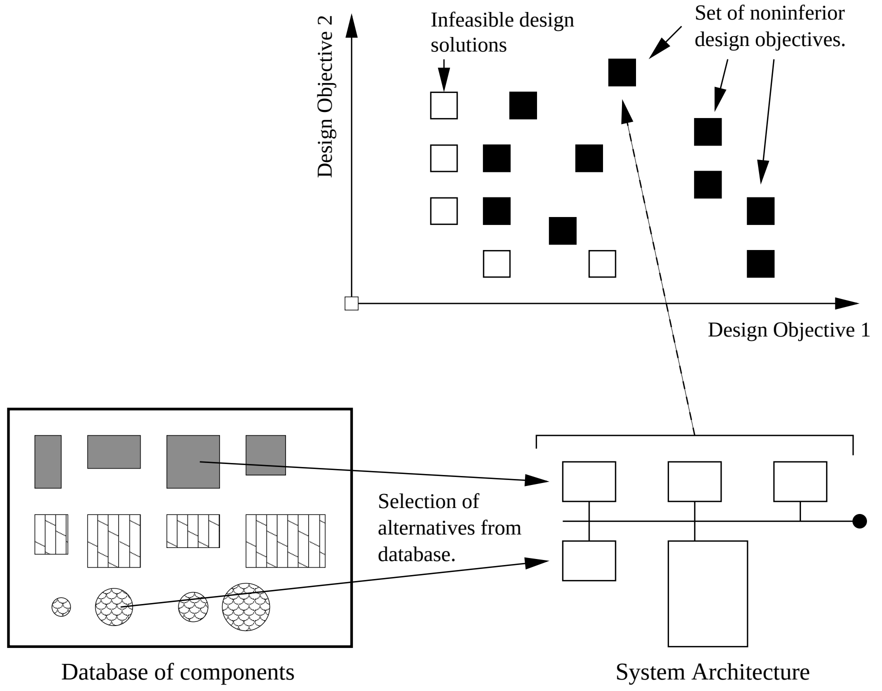

Infeasible design solutions
Set of noninferior design objectives.
Design Objective 2
Design Objective 1
Selection of alternatives from database.
Database of components
System Architecture

Figure 1.6: Trial-and-error approach to component selection.

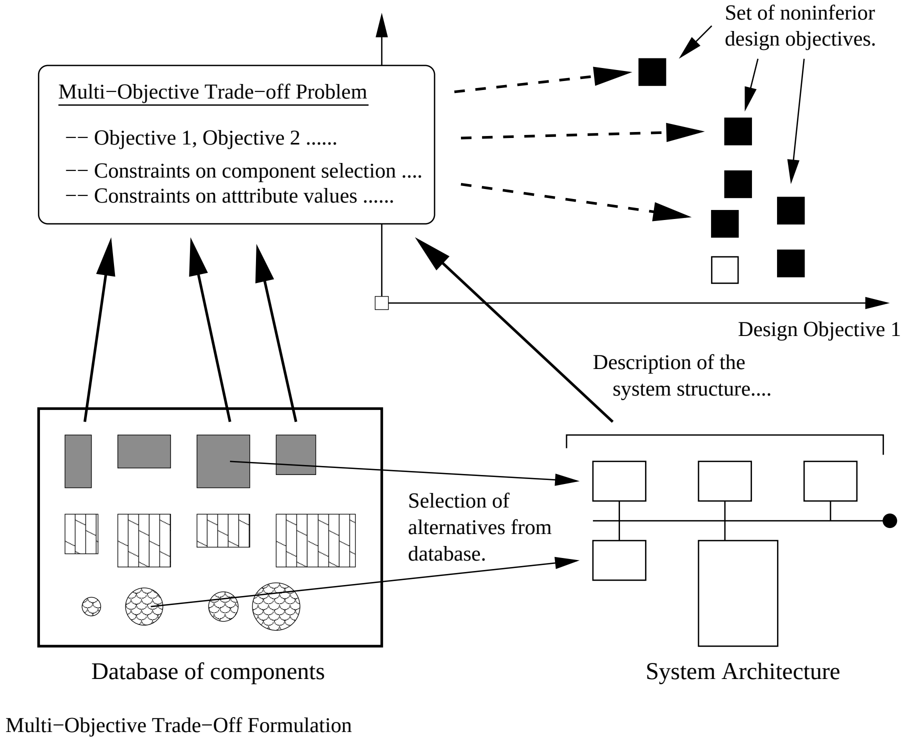

Set of noninferior design objectives.
Multi−Objective Trade−off Problem
−− Objective 1, Objective 2 ......
−− Constraints on component selection ....
−− Constraints on atttribute values ......
Design Objective 1
Description of the system structure....
Selection of alternatives from database.
Database of components
System Architecture
Multi−Objective Trade−Off Formulation

Figure 1.7: Casting the component selection as a multi-objective tradeoff problem.

1. Components are selected from the database and assigned to required elements of the system architecture.

2. The system is evaluated in terms of objectives 1 and 2. Each combination of components will result in one data point shown in the plot of design objectives.

3. We assume that while each component has good component-level performance and satisfies all component-level requirements, the integration of components into a system architecture will be subject to additional system-level constraints. Some of the design solutions will satisfy all of the system-level constraints and some will not. The former group are called feasible designs. The latter group are called infeasible designs.

4. We seek designs (possibly a family of designs) that are feasible, and maximize to the extent possible, one or more design objective values.

For problems containing many constraints and design parameters, the trial-and-error approach may be very inefficient, requiring treatment for an unnecessarily large number of trial solutions. One way of overcoming this problem, as illustrated in Figure 1.7, is to cast the component selection problem as a multi-objective tradeoff problem involving design objectives, and equality and inequality constraints on the attribute values, compatibility of components, and component- and system-level performance. Sequences of point solutions to the trade-off formulation can be programmed and computed with the ILOG solver [17].

A second solution approach comes about by the viewing design process as sequence of decision making problems – the challenge is to determine a sequence the decisions to efficiently resolve the value of free variables in the overall design problem while satisfying the design constraints. Rather than use ILOG, we explore the feasibility of solving the design problem through the representation of requirements and components in an RDF format, followed by their processing through sequences of inference rule application and graph queries.

1.2.3 Graph-Based Modeling and Design of Systems

Graph theory dates back to the Swiss mathematician Leonard Euler (1707-1783). From a mathematical standpoint, we denote a graph G by G(V, E) where V is a set of vertices and E is a set of edges. The edges have no points in common except those contained in V. A directed graph is one in which the edges have direction – directed edges are called arcs (e.g., transitions in statechart diagrams). An edge sequence between vertices v_1 and v_2 is a finite set of adjacent and not necessarily distinct edges that are traversed in going from vertex v_1 to vertex v_2. A large number of mathematical algorithms have been developed to traverse graphs and trees and answer specific questions about their contents (e.g., find a specific node in the tree/graph; determine if the graph is circuit-free; find all of the paths, including the shortest path; compute flows of traffic between two vertices; compute the intersection and union of graphs).

Graph Structures in Systems Engineering. When requirements are organized into levels for team development, graph structures are needed to describe the comply and define relationships among requirements (terminology such as incoming and outgoing requirements is sometime used). A parent requirement may have “N” derived children requirements, and a derived child requirement may have “M” parents. Individual requirements are linked together using graph structures. Depending on the requirements context, they need to support the allocation of requirements onto a number of other system modeling entities like parts (components), functions and interfaces.

Sometimes instances of requirements that come from diverse members of the design team and/or external influences (see Figure 1.1) will combine to over-constrain certain variables, causing inconsistency. Theories have been developed to analyze graph topologies, identifying systems that are possibly over- or under-constrained (i.e., instances of over-constraint occur within segments of a graph containing circuits). Analysis of computational properties can lead to the identification of variables having the freedom to participate in tradeoff studies [12].

1.3 Systems Engineering and the Semantic Web

The Semantic Web is important to the Systems Engineering community because it provides formalisms (i.e., models and tools) for sharing and reasoning with data on the Web. As companies move toward the team-based development of projects and products, having Web access to design specifications and component specifications adds value to business operations.

1.3.1 Semantic Web Vision

In his original vision for the World Wide Web, Tim Berners-Lee described two key objectives [7]: (1) To make the Web a collaborative medium, and (2) To make the Web understandable and, thus, processable by machines. During the past twenty years the first part of this vision has come to pass – today's Web provides a medium for presentation of data/content to humans. Machines are used primarily to retrieve and render information. Humans are expected to interpret and understand the meaning of the content. The Semantic Web aims to produce a semantic data structure which allows machines to access and share information, thus constituting a communication of knowledge between machines, and automated discovery of new knowledge [13, 26]. Realization of this goal will require mechanisms (i.e., markup languages) that will enable the introduction, coordination, and sharing of the formal semantics of data, as well as an ability to reason and draw conclusions (i.e., inference) from semantic data obtained by following hyperlinks to definitions of problem domains (i.e., so-called ontologies).

1.3.2 Technical Infrastructure

Figure 1.8 illustrates the technical infrastructure that supports the Semantic Web vision. Each new layer builds on the layers of technology below it. The bottom layer is constructed of Universal Resource Identifiers (URI) and Unicode. URIs are a generalized mechanism for specifying a unique address for an item on the web. The eXtensible Markup Language (XML) provides the fundamental layer for

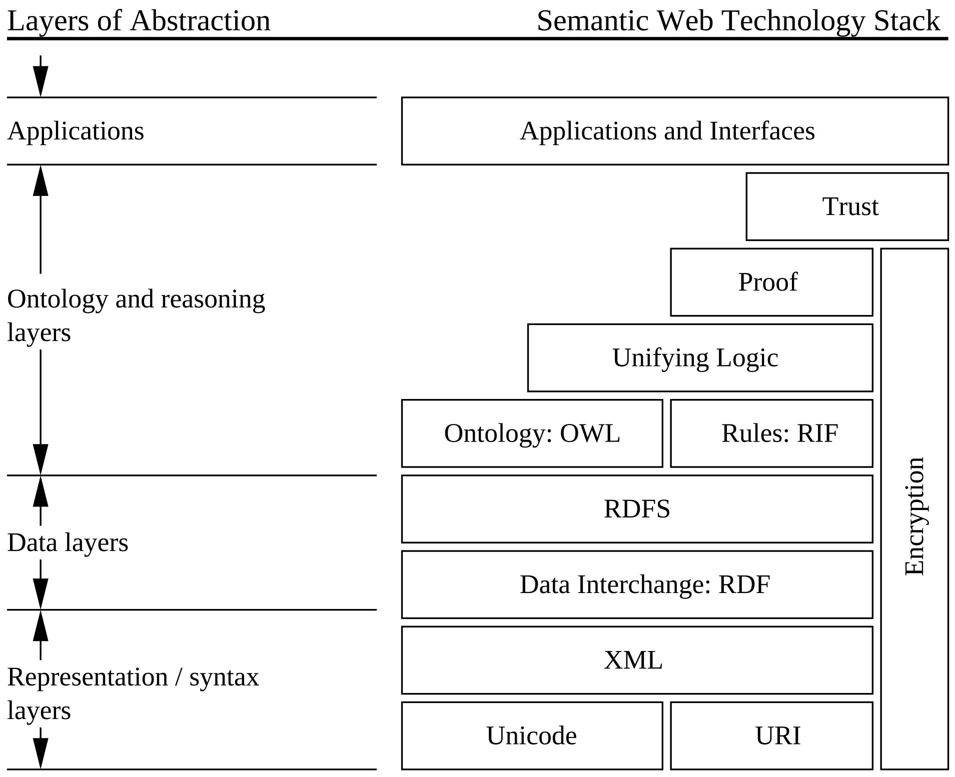


Figure 1.8: Layers of abstraction and technology in the Semantic Web.

representation and management of data on the Web. XML technology has two aspects. First, it is an open standard which describes how to declare and use simple tree-based data structures within a plain text file (human readable format). XML is a meta-language (or set of rules) for defining domain- or industry-specific markup languages. Within the systems engineering community, for example, XML is being used in the implementation of AP233, a standard for exchange of systems engineering data among tools [24]. A second key benefit in representing data in XML is that we can filter, sort and re-purpose the data for different devices using the Extensible Stylesheet Language Transformation (XSLT) [27, 29].

Limitations of XML. Need for the RDF Layer. While XML provides support for the portable encoding of data, it is limited to information that can organized within hierarchical relationships. As illustrated in Figure 1.1, a common engineering task is the synthesis information from multiple data sources. This can be a problematic situation for XML as a synthesized object may or may not fit into a hierarchical (tree) model. A graph, however, can, and thus we introduce the Resource Description Framework (RDF).

RDF is a graph-based assertional data model for describing the relationships between objects and classes (i.e., data and metadata) in a general but simple way, and for designating at least one understanding of a schema that is sharable and understandable. The graph-based nature of RDF means that it can resolve circular references, an inherent problem of the hierarchical structure of XML. An assertion is the smallest expression of useful information. RDF captures assertions made in simple sentences by connecting a subject to an object and a verb, as shown in Figure 1.9.

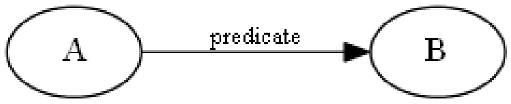


Figure 1.9: Example of RDF triple where node A is a subject, predicate is a verb, and node B is an object.

In practical terms, English statements are transformed into RDF triples consisting of a subject (this is the entity the statement is about), a predicate (this is the named attribute, or property, of the subject) and an object (the value of the named attribute). Subjects are denoted by a URI. Each property will have a specific meaning and may define its permitted values, the types of resources it can describe, and its

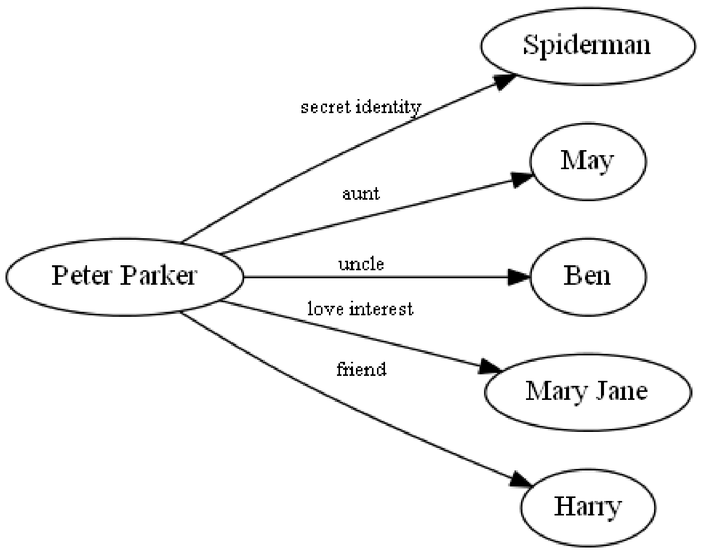


Figure 1.10: An RDF graph of relationships important to Spiderman.

relationship with other properties. Objects are denoted by a "string" or URI. The latter can be web resources such as requirements documents, other Web pages or, more generally, any resource that can be referenced using a URI (e.g., an application program or service program).

A set of related statements constitute an RDF graph. RDF graphs can be used to model relationships among friends, location data, business data, and show information about a restaurant and a movie [26]. Figure 1.10 illustrates, for example, a graph model of relationships relevant to Spiderman.

Ontology, Logic, Proof and Trust Layers. Hendler [15] describes an ontology as "a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic." Ontologies are needed to facilitate communication among people, among machines, and between humans and machines. To provide a formal conceptualization within a particular domain, ontologies need to accomplish three things: (1) Provide a semantic representation of each entity and its relationships to other entities; (2) Provide constraints and rules that permit reasoning within the ontology, and (3) Describes behavior associated with stated or inferred facts. Ontologies that will enable application interoperability by resolving semantic clashes between application domains and standards/design codes are currently in development [10, 19].

The ontology, logic, proof and trust layers introduce vocabularies, logical reasoning, establishment of consistency and correctness, and evidence of trustworthiness into the Semantic Web framework. Class relationships and statements about a problem domain can be expressed in the Web Ontology Language (OWL) [28]. Recently, rule languages such as SWRL (Semantic Web Rule Language) have been developed to provide designers with mechanisms to reason with data and class relationships [16].

1.3.3 Framework for Ontology-Enabled Development

Figure 1.11 shows two pathways for ontology-to-reasoning capability. In the upper schematic, Jena is a Java framework for the development of applications for the the Semantic Web. It provides interfaces and classes for the manipulation of RDf repositories and OWL-based ontologies. And because OWL is an extension of RDF, OWL related classes and interfaces are extensions of those used in the RDF API. One way of implementing the Jena framework is as a plugin to the Protege Ontology Editor. The lower schematic shows a graph of dependencies for reasoning

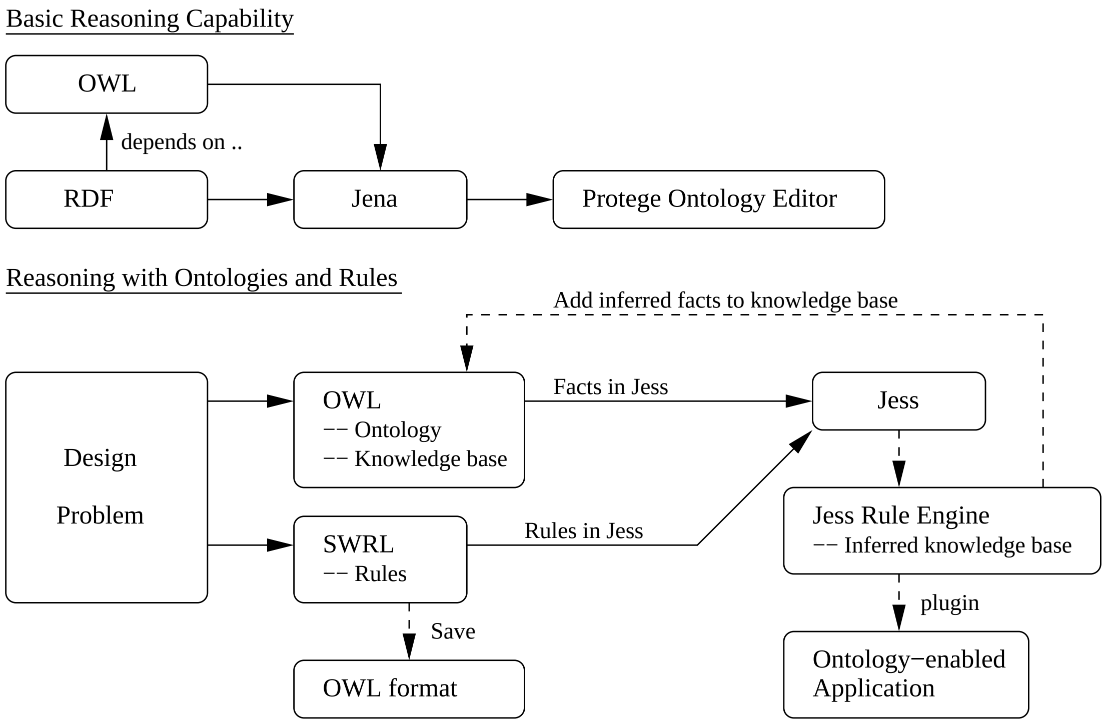


Figure 1.11: Framework for ontology-enabled development.

with ontologies and rules, enabled by the translation of an OWL knowledge base and SWRL rules into a format that can be handled by the Jess Reasoning Engine [18]. The result is a rules-based system that uses rules to reach conclusions from a set of premises about a system.

1.4 Preliminary Work at UMCP

In preliminary work conducted by researchers at the University of Maryland during 2005-2008, ontology models and models of component connetivity were formulated for the architectural synthesis of home stereo systems from pre-defined electronic components [3, 22]. Figure 1.12 shows a simplified port-connectivity-cable model. And Figure 1.13 shows an ontology for the stereo system modeled in the Pro-

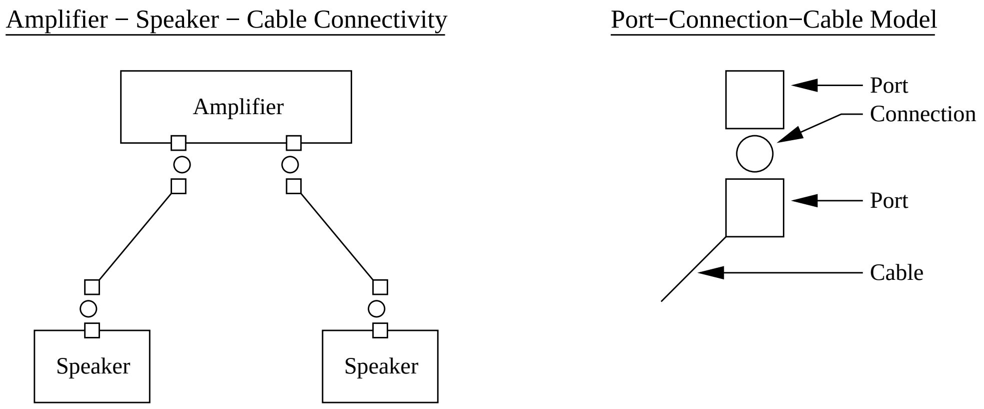

Amplifier − Speaker − Cable Connectivity
Amplifier
Speaker
Speaker
Port−Connection−Cable Model
Port
Connection
Port
Cable

Figure 1.12: Simplified model for connectivity of stereo components.

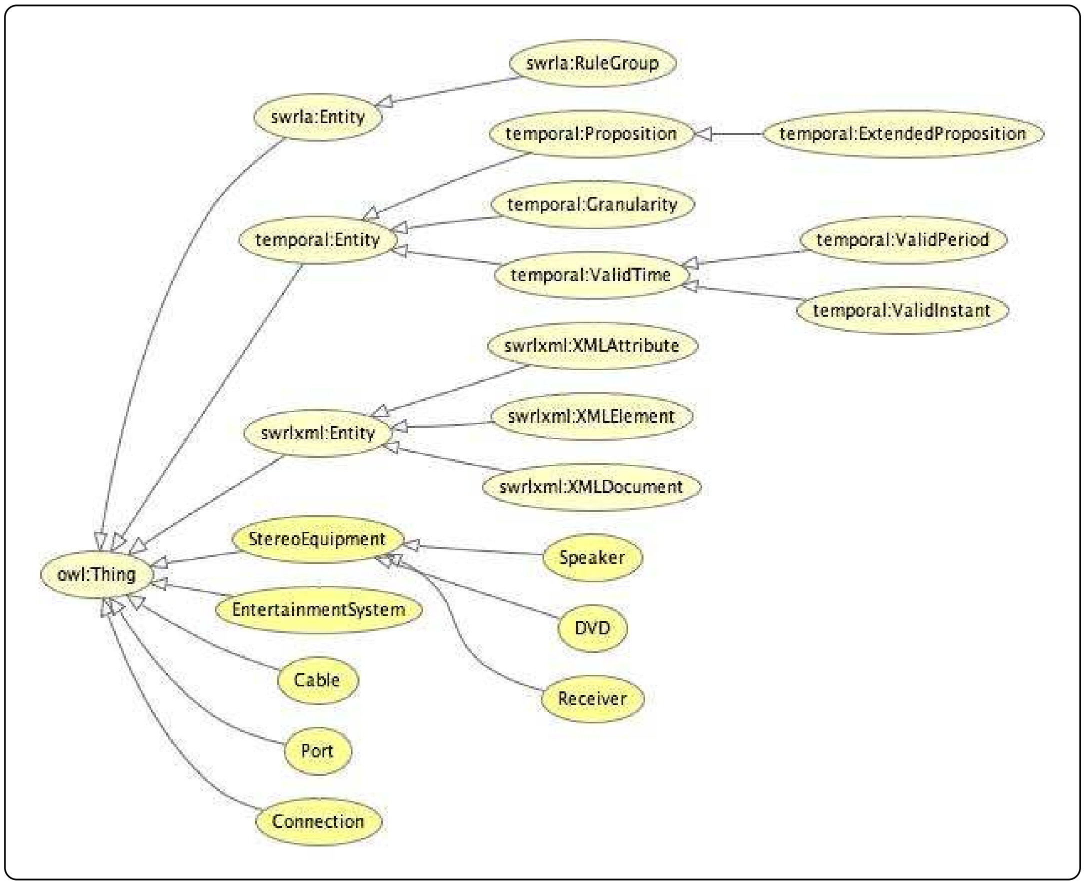


Figure 1.13: Stereo system ontology modeling in Protege

tege Ontology Editor [25]. Class relationships and the domain restriction between the Port and Connection specify what kinds of connections are permitted and what kinds of connections are prohibited. The automatic generation of design alternatives is established through configuration modeling, and the translation of ontologies and SWRL rules into a format that can be handled by the Jess Reasoning Engine [18]. To see how this works in practice, consider the fragment of code written in SWRL:

```
AMP(?a) ^ hasPrice(?a, ?p1)  ^ DVD(?d) ^  hasPrice(?d, ?p2)  ^
swrlb:add(?s, ?p1, ?p2)  ^ swrlb:lessThan(?s, 600.0) -> hasCheapSystem(?a, ?d)
```

The problem definition in English is as follows: (1) I have m amps and n dvds, (2) My system should include a dvd and a amp and a price has to be less than $600, (3) Do I have such a system? The Jess reasoner searches through the design space and prints out all combinations of suitable dvd/amp/price. Since new (or derived) information will be fed back into the design problem description, design is inherently an evolutionary and iterative process.

1.5 Scope and Objectives

Looking forward, many opportunities for improvements to the way in which computers are used to provide assistance in system development seem possible. We observe, for example, that today the vast majority of modern computer-based tools for engineering analysis focus on simulation-based performance assessment of detailed designs. This practice ignores the fact that by the time the detailed design is obtained, most of the important decisions (and commitments to resources) have already been made. This deficiency points to a strong need for computer assistance during the conceptual/early stages of development where the formulation and management of design constraints and identification of potentially good design solutions and trade-offs is of paramount importance.

Mitigating this deficiency will require formal representations for requirements and their associated constraints, as well as new algorithms and computational support for the automated synthesis, assessment and ranking of design alternatives. As a starting point, one approach would be to consider solutions based upon the collection of technologies described in Sections 1.3.3 and 1.4. This idea is less than ideal because any implementation would be complicated by the need to work with a multitude of languages (RDF, OWL, SWRL), transformations (SWRL to Jess; OWL to Jess), and tools (Protege, Jess, SWRL bridge, Jena). These tools are also in various stages of development and working order, but need to work together if we are to move forward.

In an effort to streamline and simplify computational support for design, the purposes of this research project are to explore whether or not it is possible to use RDF and Python as a replacement for RDF, OWL, Jess, Jena, Protege, and SWRL. We will develop RDF graph representations for requirements and their properties (no ontologies), and use Python for the implementation of logical reasoning and inferencing mechanisms. From the outset it is evident that models of requirements can cover a range of abstractions (i.e., levels of detail). This, in turn, will affect the types of inferencing mechanisms that can be developed in Python and the types

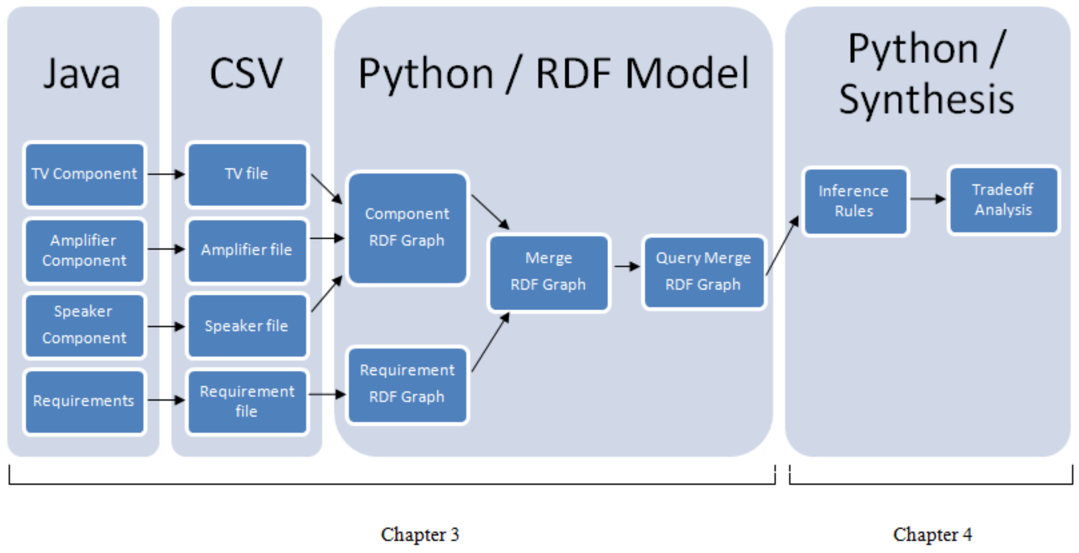


Figure 1.14: Flowchart for systems modeling with Java and Python.

of systems validation that will be possible. At this time we do not understand the model-inferencing-validation trade-space. Longer-term concerns of interest include: What are we giving up by throwing OWL/SWRL out of the picture? To what extent can a Python implementation of inference be fully automated? Can you do things in Python that are very difficult achieve in OWL and SWRL?

Since we are aiming for simplicity we do not use URIs – instead, we use strings to label each node within the RDF graph. For larger applications URIs would be neccessary to identify each node for distinctiveness [26]. We focus on merging RDF graphs to develop relationships between design requirements and design components and the design of sequences of inference rules that will systematically transform and filter the RDF graph into representations for ensembles of Pareto optimal design options. We hope to identify good design solutions of a system, upon which the application of tradeoff analysis will determine the best feasible design solution.

The scope of work and procedure of investigation is as illustrated in Figure 1.14. In Chapter 2 we develop requirements and propose a multi-level framework for design of a home theater system cast as a component selection design problem. Chapter 3 focuses on development of a methodology for systems modeling and trade space-analysis with RDF, Python, and Java. Important tasks include: (1) the recognition and formulation constraints. (2) the generation of plausible design alternatives, (3) identification and exploration of the boundaries of a design space, and (4) preliminary evaluation of system performance and cost in order to identify the most promising candidates for detailed design, further analysis and refinement. An algorithm and software for the computation Pareto-Optimal designs is developed in Chapter 4. In Chapter 5, we apply this framework and methodology to the home theater design and tradeoff analysis – the objective is to select television, speaker, and amplifier compatible combinations that cost less than USD $2,100, and then through the application of tradeoff analysis identify the best feasible home theater design combination. Chapter 6 presents a summary of the work along with conclusions and suggestions for future research.

Chapter 2

The Home Theater Design Problem

The aim of this chapter is to develop requirements and propose a multi-level framework for design of a home theater system cast as a component selection design problem.

Working in conjunction with Vimal Mayank, Natasha Kositsyna and Mark Austin from the University of Maryland [3, 4, 20, 21, 23], the Home Theater Design Problem was first posed by David Everett at NASA Goddard in 2003 as an exercise in understanding how requirements should be written and organized for the team-based development of engineering systems. The exercise was posed as a "return to first principles of requirements engineering" with the goal of trying to understand: What kinds of requirements need to be written? How should they be organized so that they mirror the process of deciding upon and buying a home theater system? Who will be involved in the decision making process? And when will they be involved? Back in 2003 answers to these questions were viewed as a first step toward representing requirements, components, and system descriptions in a model-based format amenable to formal analysis and representation on the Web. During the past decade, the technical capability and economics of consumer electronics has evolved through several generations. Today, components cost a fraction of their price in 2001-2003. As such, the problem presented the home theater design problem with specifications revised to 2012-2013 expectations and capabilities.

2.1 Design Requirements

Development of the design requirements begins with a statement of need (and intial requirements reflecting the statement of need) and evolves into three levels of requirements. The Level 1 requirements are the initial requirements. The Level 2 requirements serve the purpose of defining a detailed agreement between the customer and supplier. The Level 3 represents are the component-level requirements and are cast as inequality constraints written in terms of the component attribute values. In other words, they contain quantitative elements that can constrain the selection of components from a database.

The problem formation addressed here is simplified in the sense that only the Level 3 requirements are evaluated quantitatively. Evaluation of the Level 1 and Level 2 requirements occurs through the logical satisfaction of the sets of lower-level requirements.

Statement of Need. My wishes are very modest:

1. I simply want to watch movies on a large size theater screen that is connected to a high fidelity audio system.

Initial Requirements.

1. I need a home theater system.

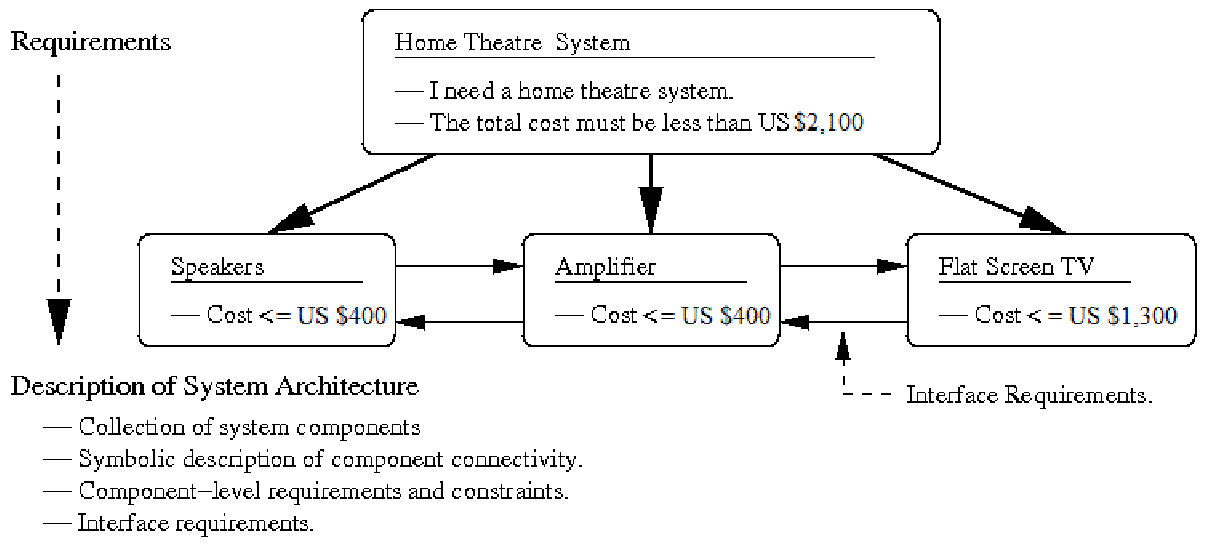

Requirements
Home Theatre System
— I need a home theatre system.
— The total cost must be less than US $2,100
Speakers
— Cost <= US $400
Amplifier
— Cost <= US $400
Flat Screen TV
— Cost < = US $1,300
Interface Requirements.
Description of System Architecture
— Collection of system components
— Symbolic description of component connectivity.
— Component–level requirements and constraints.
— Interface requirements.

Figure 2.1: Flowdown of requirements to a detailed system architecture description.

2. The total cost must be less than USD $2,100.

Now let us assume that the flowdown of requirements to a detailed system architecture proceeds as shown in Figure 2.1, with the requirements are organized into three layers.

Level 1. Summary of Initial Requirements

Table 2.1 summarizes the two initial requirements, a reflection of the high-level statement of need plus an overall budgetary constraint.

Level 2. Agreement between the Customer and Supplier

Table 2.2 outlines the essential features of a detailed agreement between the customer and supplier. Point to note:

1. The statement "home theater" is refined into a visual display system plus an audio system, along with preliminary requirements for where the visual

Level 1 Requirements
INITIAL CUSTOMER REQUIREMENTS
R1 I need a home theater system.
R2 The total cost must be less than $2,100.

Table 2.1: Initial customer requirements for a Home Theater System.

Level 2 Requirements
DETAILED AGREEMENT BETWEEN CUSTOMER AND SUPPLIER
Visual Display
R3 The theater system shall have a large display screen.
R4 The display must be thin enough to be mounted on a wall.
R5 Cost of the visual display shall not exceed US $1,300.
Audio System
R6 The system shall have a high fidelity audio system.
R7 Cost of the ”audio system” shall not exceed US $800.

Table 2.2: Detailed agreement between the customer and the builder of the Home Theater System.

Level 3 Requirements
COMPONENT REQUIREMENTS
Flatscreen TV
R8 The width of the screen shall be at least 3 ft.
R9 The height of the screen shall be at least 2 ft.
R10 The screen thickness shall be no more than 6 inches.
R11 Weight of the screen shall be no more than 60 lbs.
R12 Cost of the flatscreen TV system <= US $1300.00
Amplifier System
R13 The price of the amplifier system <= US $400.00
Speaker System
R14 Cost of the speaker system <= US $400.00.
R15 Capacity of the speaker system output shall be at least 150 watts.

Table 2.3: Component-level requirements for the Home Theatre System.

display will be positioned (i.e., on the wall) and a qualitative statement for its dimensions (i.e., large).

2. Overall cost of the system ($2,100) is apportioned to allowable budgets for the visual display and audio systems.

3. Implicit requirements emanate from the tight budgetary constraints. if we only have $2,100 to spend, then custom-building a home theater system is financially infeasible. The only practical alternative is use of reliable commercial off-the-shelf (COTS) components. Use of these components will required access to a standard A/C power supply in the house.

Level 3. Component-Level Requirements

The component-level requirements are summarized in Table 2.3 and are written in such a way that they can be easily converted into mathematical inequality constraints expressed in terms of the component atttributes. The matching component options will have specific values of component atttributes plus overall estimates of component performance and reliability.

Generation of Requirement-Specifications

Requirement-specifications for the home theatre system are generated by identifying the "quantitative element" for each Level 3 requirement. From Table 2.4 (and the level 3 requirements table), we see that the flatscreen TV objects need to have the following attributes:

Requirement	**Structure**	**Behavior**
	Objects, Attributes	Time, Performance, Sequence
TV Req. 8. Screen width.	at least 3ft.	
TV Req. 9. Screen height	at least 2ft.	
....	....	.....
Speaker Req. 15. Speaker output	....	at least 150 watts
....	....	.....

Table 2.4: Abbreviated generation of requirement-specifications from Level 3 requirements.

```
Screen height, screen width, thickness, weight,
```

and the speakers will have the performance metric

```
Output (watts).
```

2.2 Selection of Components for the Home Theater System

The systematic selection of components for the home theater system (with trade-off) can be partitioned into three sub-problems: (1) Selection of the Flatscreen TV, (2) Selection of the speakers, and (3) Selection of the amplifier.

Figures 2.2 and 2.3 provide a snapshot for how the selection of components and evolution of a design space can occur. We assume that each sub-problem can be solved by searching a database for a component that satisfies the component-level specifications. These subproblems are not uncoupled, however. The tentative selection of an audio component will impose (or fill in) additional interface constraints on acceptable TV and speaker components. These constraints, in turn, will refine the description of the design space.

As a case in point, Figure 2.2 shows the situation where details on the amplifier, speaker and television components remain to be filled in. Notice that all three of the cost requiremnts are expressed as inequality constraints. Figure 2.3 illustrates the next step in a hypothetical design process – by tentatively selecting an amplifier component for \$350, the remaining budget can be reapportioned to the television and speaker components.

If the requirement-specifications are too stringent then the result may be zero feasible design options. Conversely, if the requirement-specifications are too easily satisfied and/or the design problem is underconstrained then the number of potentially acceptable design solutions may be too large to be useful. To fully understand the tradeoffs that are possible in cost versus system functionality and performance, while also solving the selection problem in an efficient manner, manual selection procedures need to be replaced by formal approaches to multi-objective tradeoff analysis.

2.3 Requirement Attributes

We use examples of attribute tags for requirements taken from Austin, Mayank and Shmunis [4], which identify the name of the attribute and the value of

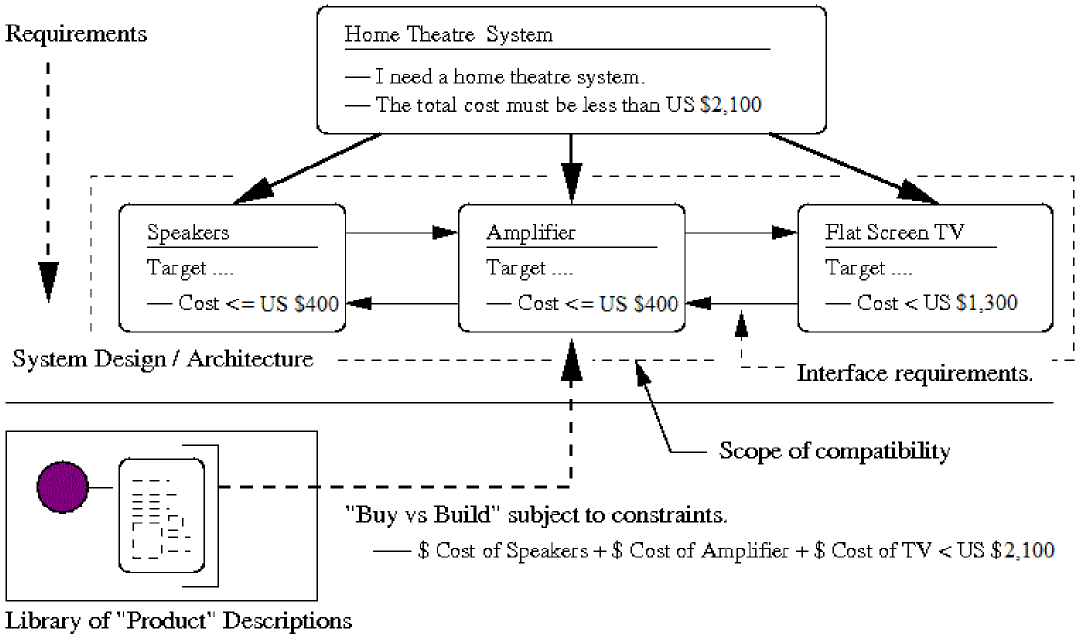

Requirements
Home Theatre System
— I need a home theatre system.
— The total cost must be less than US $2,100
Speakers
Target ....
— Cost <= US $400
Amplifier
Target ....
— Cost <= US $400
Flat Screen TV
Target ....
— Cost < US $1,300
System Design / Architecture
Interface requirements.
Scope of compatibility
"Buy vs Build" subject to constraints.
— $ Cost of Speakers + $ Cost of Amplifier + $ Cost of TV < US $2,100
Library of "Product" Descriptions

Figure 2.2: Assembly of the system architecture. Choosing the amplifier.

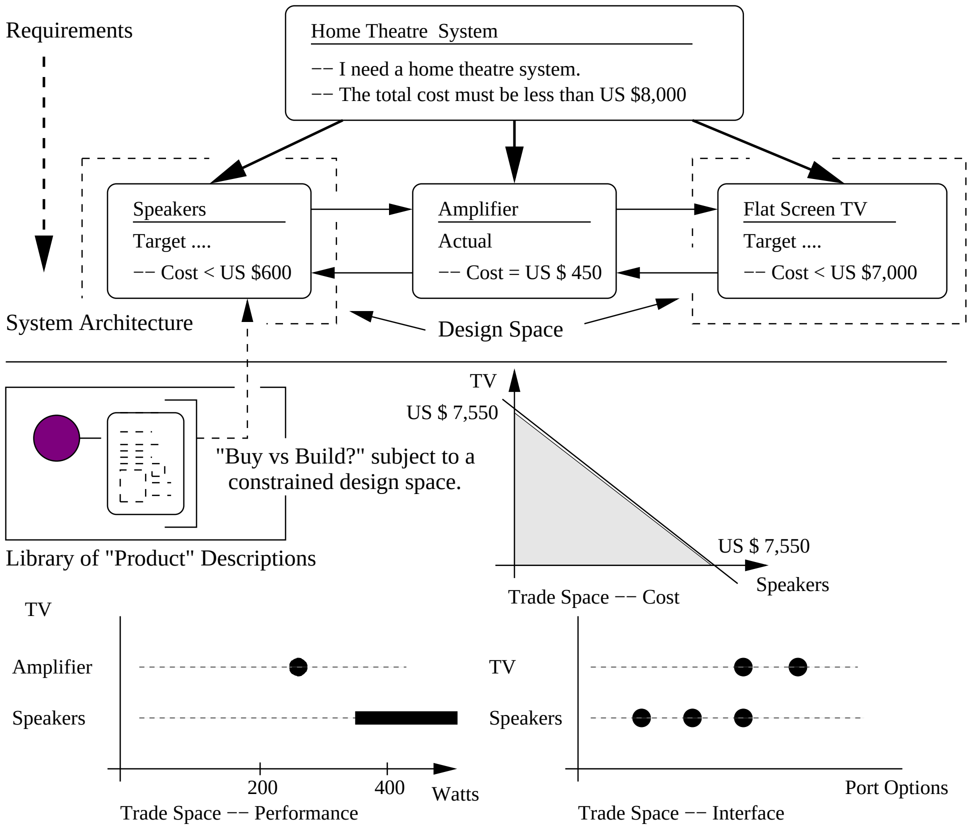


Figure 2.3: Assembly of the system architecture. Choosing the speakers.

the attribute. Such attribute tags include *ID* and *requirement title*. *ID* establishes the identity of a requirement and *requirement title* is the written description of a requirement. The attribute tags described in [4], lead to the development of new attribute tags implemented in this paper. New attribute tags include: *category*, *requirement level*, *derived by*, and *depends on*. *Category* identifies a requirement as a requirement. This attribute is essential when querying RDF graphs, which we will discuss in Chapter 3. *Requirement level* assigns a requirement to its position within the requirement hierarchy. *Derived by* states the requirement source, which a requirement is derived from. *Depends on* states the particular requirement(s) which the status of a requirement is dependent on.

2.4 Design Component Library

The home theater system comprises three component libraries:

1. Television library.

2. Amplifier library.

3. Speaker library.

with the television, amplifier, and speaker properties obtained bestbuy.com and polkaudio.com [1, 2]. As illustrated in Tables 2.5 - 2.7, the television library includes Sony, LG, and Samsung televisions, the amplifier library includes Bose, Klipsch, and Polk amplifiers, and the speaker library includes Bose, Klipsch, and Polk speakers.

TV	Cost	P	R	Height	Width	Thickness	Weight	Inputs	Outputs
LG	$1300	5	0.7	30.8 in	50.6 in	1.2 in	48.7 lbs	AC Power, HDMI, Video, Audio-L, Audio-R, Antenna/Cable, LAN, USB	Audio-L, Audio-R, Headphones
Samsung	$1650	8	0.8	29.0 in	49.3 in	1.2 in	35.7 lbs	AC Power, HDMI, Video, Audio-L, Audio-R, Antenna/Cable, LAN, USB, Ex-Link	Audio-L, Audio-R, Headphones
Sony	$1200	10	0.9	30.4 in	50.0 in	1.6 in	44.5 lbs	AC Power, HDMI, Video, Audio-L, Audio-R, Antenna/Cable, LAN, USB	Audio-L, Audio-R, Headphones

Table 2.5: Generation of television design components for the television library. Legend: P = performance, R = reliability.

Amplifier	**Cost**	**Performance**	**Reliability**	**Power Handling**	**Inputs**	**Outputs**
Bose	\$300	10	0.8	100 watts	AC Power, Audio-L, Audio-R	Speaker-L, Speaker-R
Polk	\$350	8	0.9	175 watts	AC Power, Audio-L, Audio-R	Speaker-L, Speaker-R
Klipsch	\$370	5	0.7	70 watts	AC Power, Audio-L, Audio-R	Speaker-L, Speaker-R

Table 2.6: Generation of amplifier design components for the amplifier library.

Speaker	**Cost**	**Performance**	**Reliability**	**Power Handling**	**Inputs**	**Outputs**
Polk	$400	8	0.8	10 - 150 watts	Speaker-R, Speaker-L	Sound
Klipsch	$300	5	0.7	5 - 85 watts	Speaker-R, Speaker-L	Sound
Bose	$328	10	0.9	50 - 200 watts	Speaker-R, Speaker-L	Sound

Table 2.7: Generation of speaker design components for the speaker library.

Chapter 3

Design Methodology and Implementation

3.1 Methodology

With the Home Theater Design Problem in place, the purpose of this chapter is to lay a foundation for a design methodology that includes representation of requirements and design components as RDF graph models, followed by the automatic synthesis of compatible component pairs and filtering of design options to satisfy requirements. To see how the implementation works in Java and Python, details are provided for one requirement and one component, a television. The television, amplifier, and speaker properties were obtained bestbuy.com and polkaudio.com [1, 2].

Step-by-Step Procedure. The procedure for design space exploration of the framework consists of the following steps:

1. Assemble RDF graphs of design requirements and design components. This involves:

- **a.** Develop a Java object representation for each design requirement;
- **b.** Assign specification values to each design requirement;
- **c.** Develop a Java object representation for each design component;

d. Assign specification values to each design component;

e. Generate a comma separated value (CSV) file for both the design requirements and design components models;

f. Import both the design requirements and design components CSV files to a RDF model implemented in Python;

g. Merge the design requirements and design components graphs;

2. Develop inference rules. This involves:

a. Design inference rules for level 3 requirements;

b. Design inference rules for level 1 and level 2 requirements;

c. Design inference rules for component connectivity;

d. Design inference rules for component compatibility;

e. Design an inference rule, feasible system, that establish relationships between design components;

f. Design an inference rule, system design, that establish relationships between design requirements and design components;

3. Apply the inference rules to the merged RDF graph of design requirements and design components.

4. Query the merged RDF graph of design requirements and design components for design components which satisfy design requirements.

5. Generate a Python graphical user interface (GUI) for the trade space visualization. This involves:

  a. Construct trade-off analysis plot;

  b. Display trade-off analysis for each home theater system combination generated by querying the merged RDF graph;

6. Find the family of Pareto optimal points in the Trade-Off Analysis Plots. In a typical design problem, we will want to minimize cost, and maximize system performance and reliability.

3.2 Modeling and Visualization of RDF Graphs with Python and PyDot

For systems engineering, RDF graphs are good for modeling engineering designs by merging RDF graphs and developing inference rules which lead to good design solutions. But for large systems and when we are merging multiple RDF graphs, it can become challenging to ensure consistent and unparallel identifiers for each node [26].

3.2.1 Modeling RDF Graphs with Python

We model RDF Graphs with Python through the Simple Graph class. Specifically, this class is a triplestore that store RDFs [26]. The Simple Graph class include several methods. We wish to highlight the following methods, which we think are important: *add*, *load*, *triples*, *query*, and *apply inference*.

***Add* Method:** This example illustrates the process of adding a triple to the graph. The method adds a subject, predicate, and object to the index of the graph.

source code

```
def add(self, (sub, pred, obj)):
    self._addToIndex(self._spo, sub, pred, obj)
    self._addToIndex(self._pos, pred, obj, sub)
    self._addToIndex(self._osp, obj, sub, pred)
```

***Load* Method:** This example illustrates the process of loading a CSV file into the graph. The method opens a CSV file, reads the subjects, predicates, and objects and then add them to the graph.

source code

```
def load(self, filename):
    f = open(filename, "rb")
    reader = csv.reader(f)
    for sub, pred, obj in reader:
        sub  = unicode(sub, "UTF-8")
        pred = unicode(pred, "UTF-8")
        obj  = unicode(obj, "UTF-8")
        self.add((sub, pred, obj))
    f.close()
```

***Triples* Method:** This example illustrates the process of returning all triples within the graph which correspond to a triple pattern. The method checks which subject, predicate, or object terms are present in the triple pattern in order to return the the correct index from the graph.

source code

```
def triples(self, (sub, pred, obj)):
    # check which terms are present in order to use the correct index:
    try:
        if sub != None:
```

```
    if pred != None:
       # sub pred obj
         if obj != None:
            if obj in self._spo[sub][pred]: yield (sub, pred, obj)
       # sub pred None
            else:
                for retObj in self._spo[sub][pred]: yield (sub, pred, retObj)
    else:
         # sub None obj
         if obj != None:
            for retPred in self._osp[obj][sub]: yield (sub, retPred, obj)
         # sub None None
         else:
            for retPred, objSet in self._spo[sub].items():
                for retObj in objSet:
                    yield (sub, retPred, retObj)
 else:
    if pred != None:
       # None pred obj
       if obj != None:
          for retSub in self._pos[pred][obj]:
              yield (retSub, pred, obj)
       # None pred None
       else:
           for retObj, subSet in self._pos[pred].items():
               for retSub in subSet:
                   yield (retSub, pred, retObj)
  else:
       # None None obj
       if obj != None:
          for retSub, predSet in self._osp[obj].items():
              for retPred in predSet:
                  yield (retSub, retPred, obj)
       # None None None
  else:
       for retSub, predSet in self._spo.items():
           for retPred, objSet in predSet.items():
               for retObj in objSet:
                   yield (retSub, retPred, retObj)

# KeyErrors occur if a query term wasn't in the index, so we yield nothing:

except KeyError:
    pass
```

***Query* Method:** This example illustrates the process for variable binding within the graph. The method queries the entire graph of relationships to check which subject, predicate, or object terms are present in multiple triple patterns in order to return the the correct indices from the graph.

source code

```
def query(self,clauses):
    bindings=None

    for clause in clauses:
        bpos={}
        qc=[]
        for x,pos in zip(clause,range(3)):
            if x.startswith('?'):
               qc.append(None)
               bpos[x[1:]]=pos
            else:
               qc.append(x)
        rows=list(self.triples((qc[0],qc[1],qc[2])))

        if bindings==None:
           bindings=[]
           for row in rows:
               binding={}
               for var,pos in bpos.items():
                   binding[var]=row[pos]
               bindings.append(binding)
        else:
           newb=[]
           for binding in bindings:
               for row in rows:
                   validmatch=True
                   tempbinding=binding.copy()
                   for var,pos in bpos.items():
                       if var in tempbinding:
                          if tempbinding[var]!=row[pos]:
                             validmatch=False
                          else:
                             tempbinding[var]=row[pos]
                   if validmatch: newb.append(tempbinding)
                bindings=newb

        return bindings
```

***Apply Inference* Method:** This example illustrates the process for getting an inference rule and then applying the rule to the graph. The method gets a given set of queries from an inference rule and then apply the set of queries to the graph.

source code

```
    def applyinference(self,rule):
        queries=rule.getqueries()

        bindings=[]

        for query in queries:
            bindings+=self.query(query)


        for b in bindings:
            new_triples=rule.maketriples(b)

            for triple in new_triples:
                self.add(triple)
```

3.2.2 Visualing RDF Graphs with PyDot

RDF graphs are visualized through the application of PyDot. PyDot is a Python software application that illustrates directed graphs [9]. RDF graphs are considered directed graphs because edges of the nodes are directed away from the node [26]. Essentially, the graphical representation of RDF graphs is visualized through PyDot to illustrate the relationship between entities within a directed graph.

When modeling systems as RDF graphs, PyDot is useful to capture relationships between design components and design requirements. At first, we aim to illustrate a RDF. In PyDot, we create a directed graph. Then, we create a node for subject and object of RDF, followed by adding each node to the directed graph. RDF predicates are represented by directed edges from the subject to object. Next, the directed graph is written to a .png file and then the code is ran to visualize the formulate RDF graph.

A Simple Example: This simple example illustrates the process of creating a RDF model.

source code

```
import pydot

# Create directed graph

graph = pydot.Dot(graph_type='digraph')

# Create nodes

node_a = pydot.Node("A")
node_b = pydot.Node("B")
node_c = pydot.Node("C")
node_d = pydot.Node("D")

# Add nodes to graph

graph.add_node(node_a)
graph.add_node(node_b)
graph.add_node(node_c)
graph.add_node(node_d)

# Add edges to graph

graph.add_edge(pydot.Edge(node_a, node_b, label="predicate", labelfontcolor="#009933",
fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_c, label="predicate", labelfontcolor="#009933",
fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_d, label="predicate", labelfontcolor="#009933",
fontsize="10.0", color="black"))
```

Points to note:

1. A directed graph is specified.

2. Four nodes are created and added to the directed graph.

3. An edge is created between the nodes to create a relationship between node A and node B, node C, and node D.

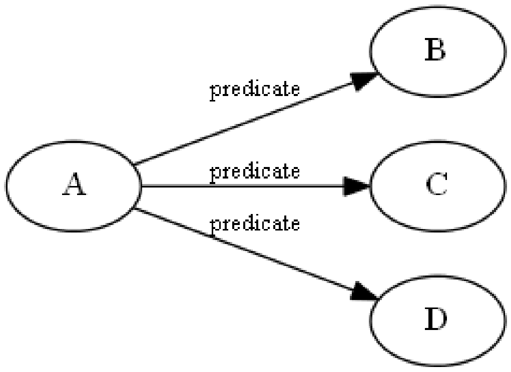


Figure 3.1: Example of a PyDot visualization of a RDF Graph.

3.3 Modeling Requirements with RDF

As described in Chapter 2, the home theater system will have three levels of design requirements, level 1, level 2, and level 3.

Definition of Home Theater Requirements. A Java class for the home theater requirements is created. Figure 3.2 shows the system architecture of the requirements class and associated driver class, both implemented in Java.

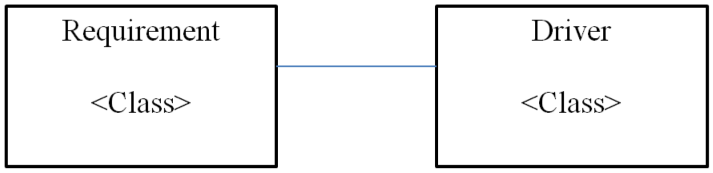


Figure 3.2: System architecture of the requirements class and associated driver.

The abbreviated details for the Requirement class are as follows:

source code

```
package javaBackend;

public class Requirement {
   private String name;
   private String title;
   private int level;
   private String [] dependOn;
   private String derivedBy;

   // Constructor methods ...

   public Requirement(String name, String title, int level, String [] dependOn,
      String derivedBy){
      this.name = name;
      this.title = title;
      this.level = level;
      this.dependOn = dependOn;
      this.derivedBy = derivedBy;
   }

   public void setName(String name){
      this.name = name;
   }

   public String getName(){
      return name;
   }

   ... details of code removed ...
}
```

Points to note:

1. A Requirement class is created.

2. Each design requirement will have a name, title, level, dependOn, and a derivedBy property.

Driver for Assembly of Requirements: The Driver class systematically defines requirement options for each requirement, and creates comma separated value representations for each requirement. The abbreviated details for the definition of a single requirement object is as follows:

source code

```
package javaBackend;

import java.util.ArrayList;

public class Driver {
   public Requirement R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,
                      R12, R13, R14, R15;

   public Driver() {
      System.out.println(" --- D e s i g n  R e q u i r e m e n t s --- ");

      // Create a list of requirements

      ArrayList<Requirement> reqList = new ArrayList<Requirement>();

      // Requirement 1

      String[] R1DependOn = new String[] { "Requirement 3",
                                           "Requirement 4", "Requirement 6"};
      Requirement R1 = new Requirement( "Requirement 1",
                                        "I need a home theater system", 1,
                                        R1DependOn,"Requirement 1");

      R1.printTitle();
      R1.printLevel();
      R1.printDependOn();
      R1.printDerivedBy();

      // Add Requirement 1 to requirement list

      reqList.add(R1);

      ... details of other requirements removed ...
   }
}
```

Points to note:

1. An array list is created to store the requirements.

2. The Requirement.java constructor method is use to create the requirement objects in Driver.java.

Graphs of requirements can be modeled with RDF and visualized with PyDot. As described in section 1.3, the subject and object of RDF are represented by a node – in this case, subject is the design requirement and object is the value of predicate. Predicate of RDF, which is a directed edge from subject to object, is the attribute of the design requirement. This, in turn, allows for the formulation of a directed graph of requirements.

3.3.1 Modeling Level 1 and 2 Requirements

Modeling a Level 1 Requirement: This example illustrates the process of creating a RDF model of the Level 1, Requirement 1 in PyDot.

source code

```
import pydot

# Create directed graph
graph = pydot.Dot(graph_type='digraph', rankdir='LR', ranksep=0.75)

# Requirement
node_a = pydot.Node("Requirement 1")

# Attributes
node_b = pydot.Node("Requirement")
node_c = pydot.Node("I need a home theater system")
node_d = pydot.Node("1")
node_e = pydot.Node("Requirement 1")
node_f = pydot.Node("Requirement 3")
node_g = pydot.Node("Requirement 4")
node_h = pydot.Node("Requirement 6")

# Add nodes to graph
graph.add_node(node_a)
graph.add_node(node_b)
graph.add_node(node_c)
graph.add_node(node_d)
graph.add_node(node_e)
graph.add_node(node_f)
graph.add_node(node_g)
graph.add_node(node_h)
```

```
# Add edges to graph
graph.add_edge(pydot.Edge(node_a, node_b, label="category", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_c, label="title", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_d, label="level", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_e, label="derived by", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_f, label="depends on", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_g, label="depends on", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_h, label="depends on", labelfontcolor="#009933",
     fontsize="10.0", color="black"))

# Create Image File
graph.write_png('Requirement1.png')
```

Points to note:

1. A directed graph is specified.

2. Nodes are created and added to the directed graph.

3. Edges are created between the nodes to develop relationships between each node.

3.3.2 Modeling Level 3 Requirements

The Level 3 requirements represent the home theater system constraints that can be quantitatively evaluated. These inequality constraints can be modeled two ways. First, the simple approach – in this case we model the inequality constraint within the title of the requirement and later implement the details of the constraint

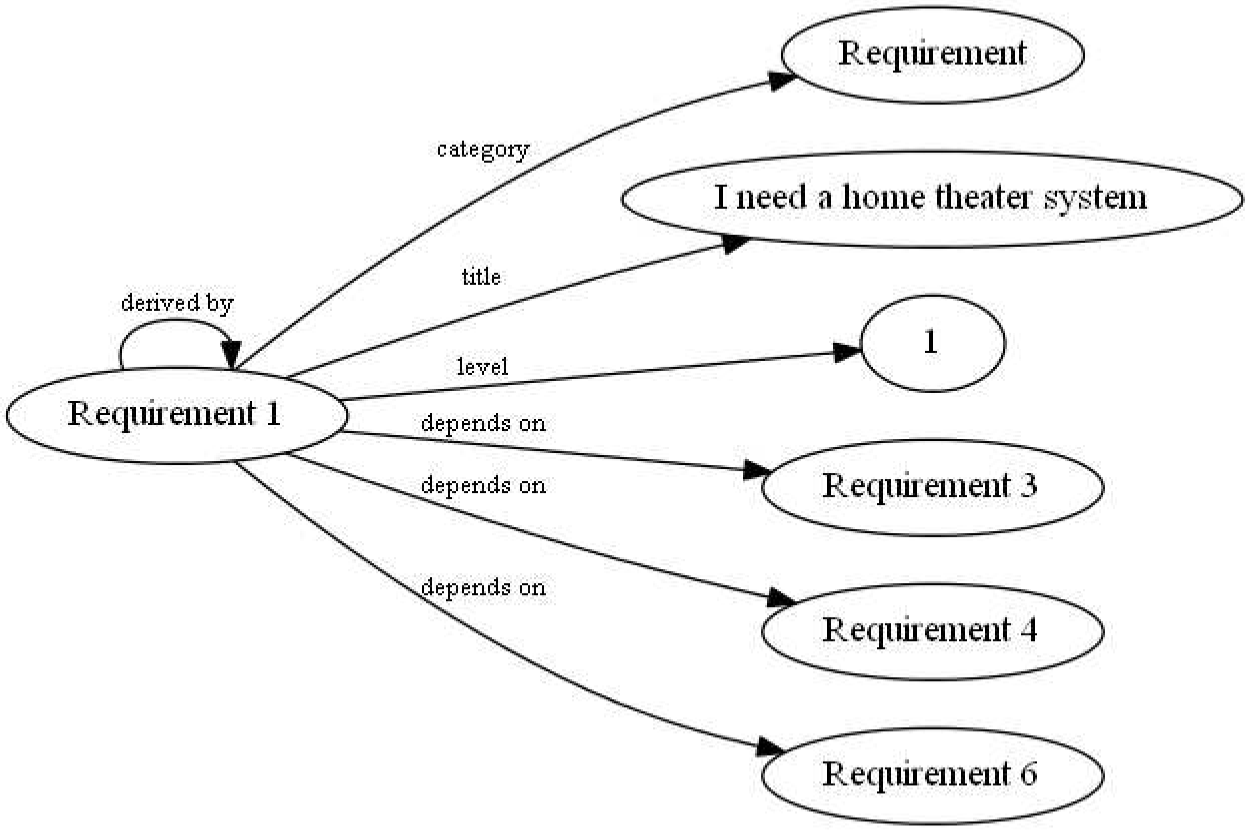


Figure 3.3: A RDF graph model for Requirement 1. Visualized using PyDot.

when developing inference rules for level 3 requirements. In the second complex approach, we model the variables of the inequality constraint as seen in Figure 3.5. For this research project, we will take the simple approach.

Like the level 1 requirement, we undergo the process of creating a RDF graph model for a level 3 requirement using PyDot. The generated RDF requirement graph is illustrated in Figure 3.4.

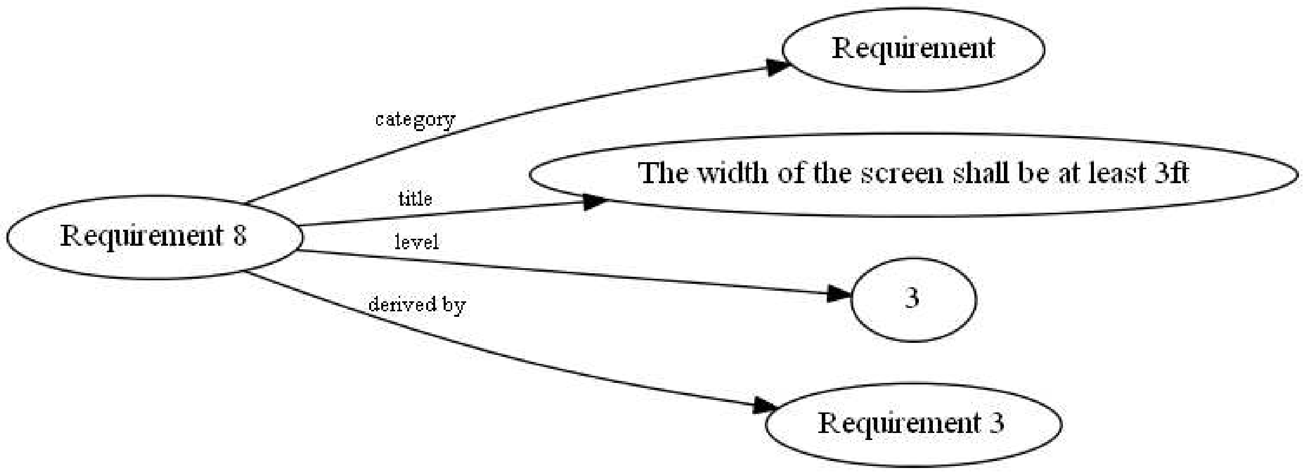


Figure 3.4: A RDF graph model for Requirement 8 using the simple approach. Visualized using PyDot.

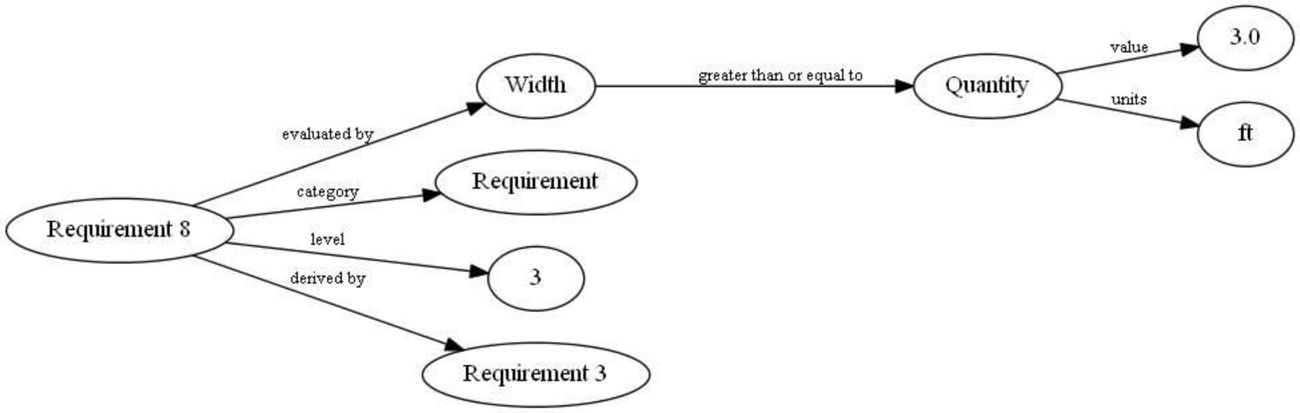


Figure 3.5: A RDF graph model for Requirement 8 using the complex approach. Visualized using PyDot.

3.4 Modeling Design Components with RDF

As described in Chapter 2, the home theater system will have three types of design components, a television, an amplifier, and two speakers. Each type of design component will be selected from a library of predefined components.

Figure 3.6 illustrates the procedure employed in this study for implementing the television, amplifier, and speaker components as Java object representations of the design components.

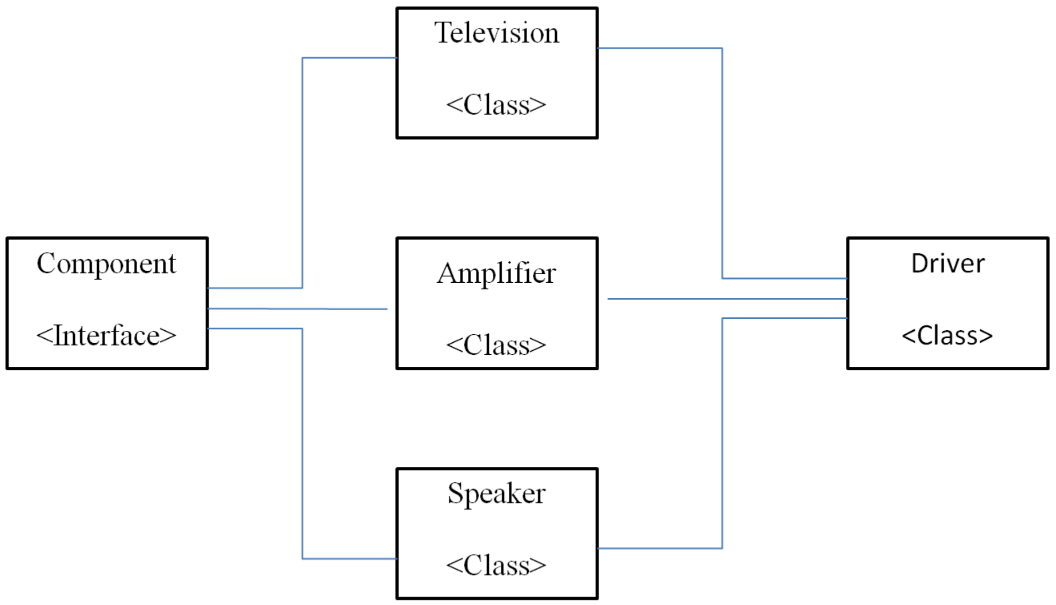


Figure 3.6: Component software implementation for Driver.Java .

Our central concern is creating a design component, and then assigning the design component to its appropriate library. First, we aim to develop a television library of design components. This is done by creating a Hashset television library object in Java.

Component Interface and Library. The television, amplifier and speaker component representations implement the general-purpose Component interface specification shown below:

```
package javaBackend;

public interface Component {
   public String getName();
   public void   setName(String name);
   public void      setInput (String [] input);
   public String [] getInput();
   public void      setOutput(String [] output);
   public String [] getOutput();
   public void   setCost(int cost);
   public int    getCost();
   public void   setPerformance(int performance);
   public int    getPerformance();
   public void   setReliability(double reliability);
   public double getReliability();
}
```

Functional support is provided for setting/getting the component name, string arrays of input and output signals, the component cost, performance and reliability.

A Library components is defined as follows:

source code

```
package javaBackend;

import java.util.*;

public class Library <Component> {
   private HashSet<Component> library = new HashSet<Component>();

   // Methods to set/get the library

   public void setLibrary(HashSet<Component> lib){
      library  = lib;
   }

   public HashSet<Component> getLibrary(){
      return library;
   }
   public void insert(Component object){
      library.add(object);
   }

   public void delete (Component object){
      library.remove(object);
   }

   public boolean isPresent(Component object){
      return library.contains(object);
   }

   public int getSize(){
      return library.size();
   }
}
```

Collections of home theater components are stored as hashsets. As such, support is provided to test the presense of a coponent in the library.

Definition of Home Theater Components. Java classes for the television, amplifier and speaker home theater components implement the Component interface specification. The abbreviated details for the Television class are as follows:

source code

```
package javaBackend;

public class Television implements Component {
   private String [] input;
   private String [] output;
   private String name;

   // Need to put this in some kind of data structure

   private double     width;
   private double    height;
   private double    weight;
   private double thickness;
   private int         cost;
   private int  performance;
   private double reliability;
   // Construtor methods ...

   public Television(String name){
      this.name = name;
   }

   public Television(String name, String [] input, String [] output,
                     double width, double height, double weight,
                     double thickness, int cost, int failure,
                     int performance, double reliability){
      this.name   = name;
      this.input  = input;
      this.output = output;
      this.width  = width;
      this.height = height;
      this.weight = weight;
      this.cost   = cost;
      this.thickness   = thickness;
      this.performance = performance;
      this.reliability = reliability;
   }

   public void setName(String name){
      this.name = name;
   }

   public String getName(){
      return name;
   }

   ... details of code removed ...

}
```

Points to note:

1. Sony, LG, and Samsung Television design components are created.

2. Inputs and outputs are assigned to each design component.

3. Parameters are assigned to each design component.

Then, we insert each television object into the television library, thus developing the television library of design components for the home theater design.

Driver for Assembly of Component Libraries: The Driver class systematically defines component options for each component type, inserts them into the libraries, and creates comma separated variable represenations for each component type. As a case in point, the abbreviated details for the definition of a single television object and assembly into the component library is as follows:

source code

```
package javaBackend;

import java.util.ArrayList;

public class Driver {
   public Television sonyTelevision, lgTelevision, samsungTelevision;
   public Library            tvLib,       ampLib,       speakerLib;

   public Driver() {
      System.out.println(" --- D e s i g n  C o m p o n e n t s --- ");

      // Create a new television

      sonyTelevision = new Television("Sony Television");

      String[] sonyTelevisionInput = { "AC Power",          "HDMI",
                                        "Video",       "Audio-L",
                                      "Audio-R", "Antenna/Cable",
                                          "LAN",           "USB" };

      String[] sonyTelevisionOutput = { "Audio-L", "Audio-R", "Headphones" };

      // Assign properties to the television object ...

      sonyTelevision.setInput(  sonyTelevisionInput );
      sonyTelevision.setOutput( sonyTelevisionOutput );

      // Print sony inputs and outputs ...

      sonyTelevision.printInput();
      sonyTelevision.printOutput();

      // Add parameters to the Sony television ...

      sonyTelevision.setWidth(50);
      sonyTelevision.setHeight(30.4);
      sonyTelevision.setThickness(1.6);
      sonyTelevision.setWeight(44.5);
      sonyTelevision.setCost(1200);
      sonyTelevision.setPerformance(10);
      sonyTelevision.setReliability(0.90);

      ... details of other televisions, amplifiers and speakers removed ...

      // Make a HashSet Library for TVs, Speakers, Amps

      Library<Television> tvLib = new Library<Television>();
      tvLib.insert(sonyTelevision);

      ... details of code removed ....
   }
}
```

Points to note:

1. The television library is created.

2. Sony, Samsung, and LG Television design components are inserted into the television library.

Like the television design component, an amplifier and speaker library of design components are developed in Java, as well as amplifier and speaker objects. Then, like the television design component, each amplifier and speaker object is inserted into its respective library. Thus, developing the amplifier and speaker library of design components for the home theater design.

3.4.1 Modeling TV Components

Graphs of design components can be modeled with RDF and visualized with PyDot. As described in section 1.3, the subject and object of RDF are represented by a node – in this case, subject is the design component and object is the value of predicate. Predicate of RDF, which is a directed edge from subject to object, is the attribute of the design component. This, in turn, allows for the formulation of a directed graph of components. Figure 3.7 models the RDF graph of LG television using PyDot.

Television Component: This example illustrates the process of creating a RDF model of the LG Television component in PyDot.

source code

```
import pydot

# Create directed graph
graph = pydot.Dot(graph_type='digraph', rankdir='LR', ranksep=0.75)

# Component
node_a = pydot.Node("LG Television")

# Input
node_b = pydot.Node("Ac Power")
node_c = pydot.Node("HDMI")
node_d = pydot.Node("Video")
node_e = pydot.Node("Audio-L")
node_f = pydot.Node("Audio-R")
node_g = pydot.Node("Antenna/Cable")
node_h = pydot.Node("LAN")
node_i = pydot.Node("USB")

# Output
node_j = pydot.Node("Audio-L")
node_k = pydot.Node("Audio-R")
node_l = pydot.Node("Headphones")

# Parameters
node_m = pydot.Node("50.6")
node_n = pydot.Node("30.8")
node_o = pydot.Node("1300")

# Add nodes to graph
# Inputs
graph.add_node(node_a)
graph.add_node(node_b)
graph.add_node(node_c)
graph.add_node(node_d)
graph.add_node(node_e)
graph.add_node(node_f)
graph.add_node(node_g)
graph.add_node(node_h)
graph.add_node(node_i)
# Outputs
graph.add_node(node_j)
graph.add_node(node_k)
graph.add_node(node_l)
# Parameters
graph.add_node(node_m)
graph.add_node(node_n)
graph.add_node(node_o)

# Add edges to graph
# Inputs
graph.add_edge(pydot.Edge(node_a, node_b, label="Input[0]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_c, label="Input[1]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_d, label="Input[2]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_e, label="Input[3]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_f, label="Input[4]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_g, label="Input[5]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_h, label="Input[6]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_i, label="Input[7]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))

# Outputs
graph.add_edge(pydot.Edge(node_a, node_j, label="Output[0]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_k, label="Output[1]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_l, label="Output[2]", labelfontcolor="#009933",
     fontsize="10.0", color="black"))

# Parameters
graph.add_edge(pydot.Edge(node_a, node_m, label="Width", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_n, label="Height", labelfontcolor="#009933",
     fontsize="10.0", color="black"))
graph.add_edge(pydot.Edge(node_a, node_o, label="Cost", labelfontcolor="#009933",
     fontsize="10.0", color="black"))

# Create Image File
graph.write_png('lgTelevision.png')
```

Points to note:

1. A directed graph is specified.

2. Nodes are created and added to the directed graph.

3. Edges are created between the nodes to develop relationships between each node.

3.4.2 Modeling Amplifier and Speaker Components

Like the LG Television component in subsection 3.4.1, we undergo the process of creating a RDF model of the Bose Amplifier and Polk Speaker in PyDot. These generated RDF component graphs are illustrated in Figure 3.8 and 3.9 respectively.

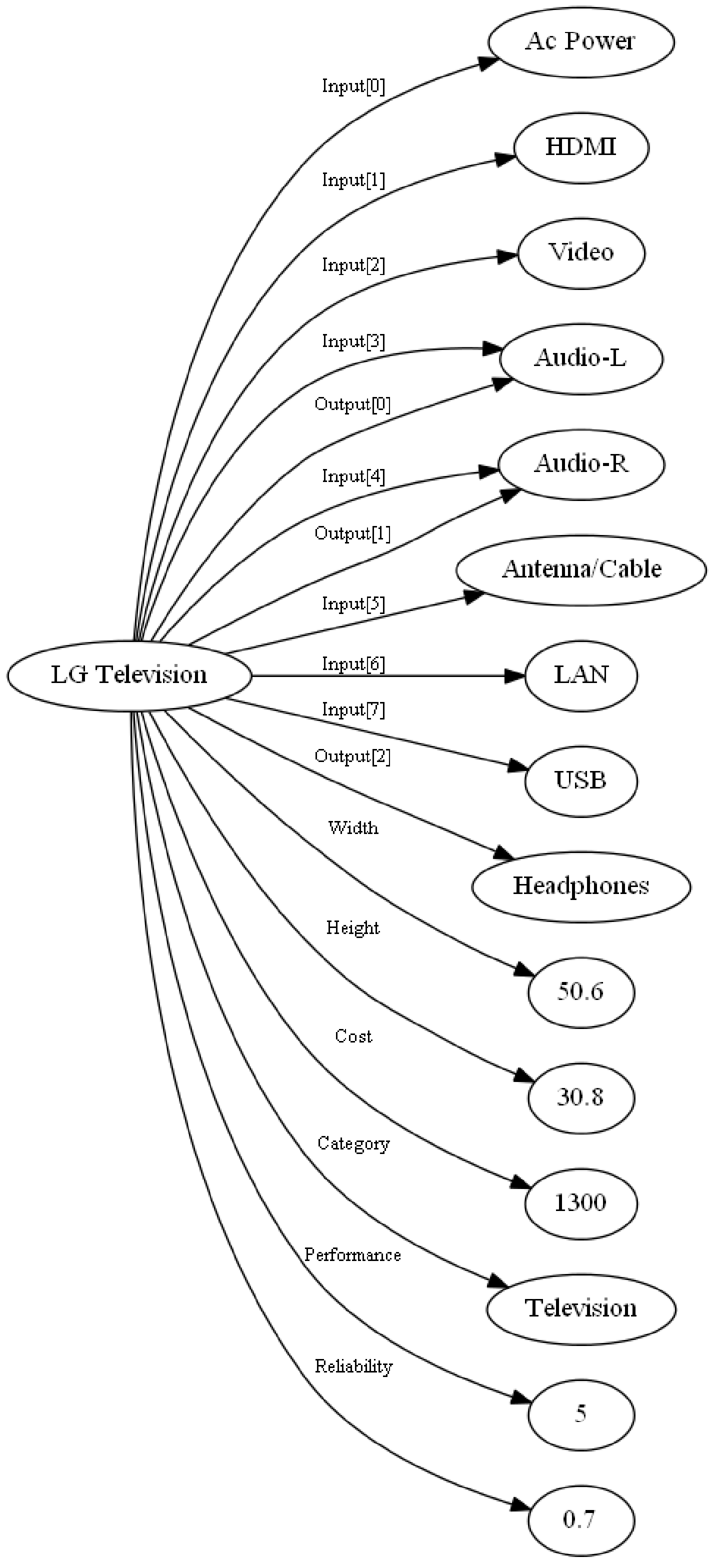


Figure 3.7: Modeling the RDF graph of LG Television using PyDot.

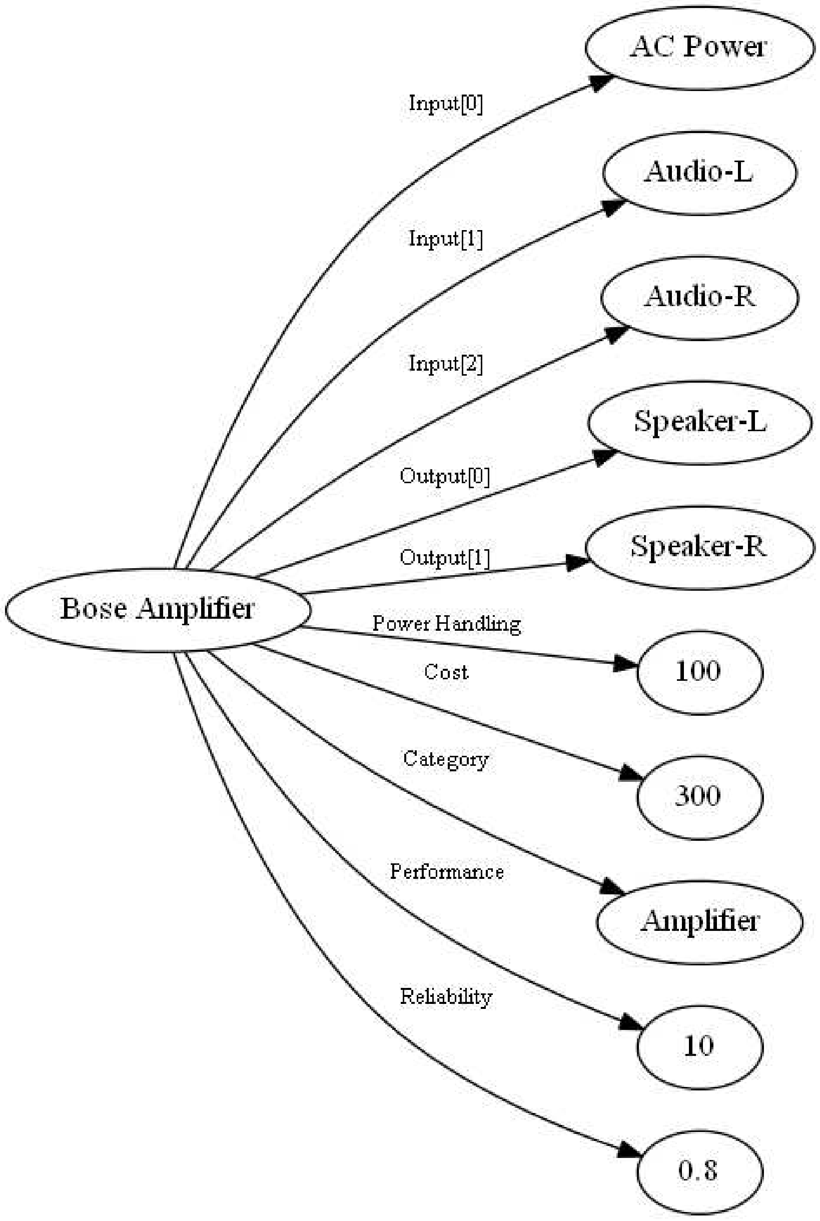


Figure 3.8: Modeling the RDF graph of Bose Amplifier using PyDot.

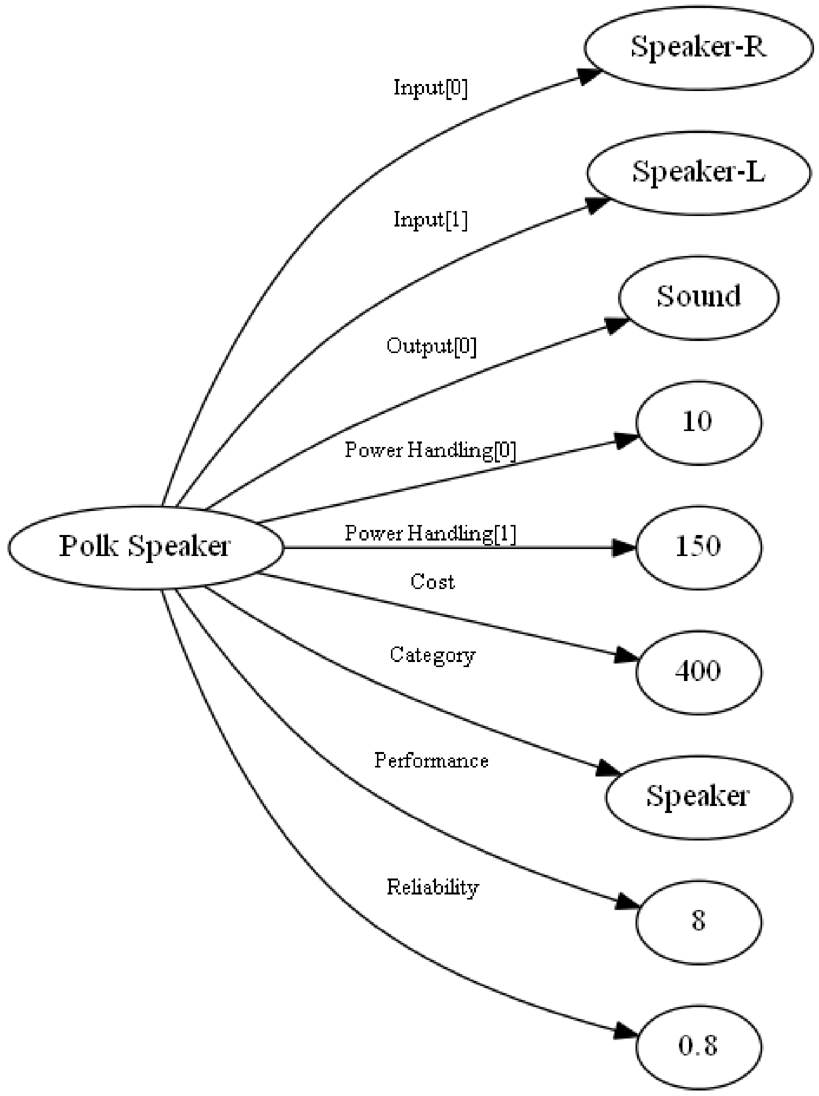


Figure 3.9: Modeling the RDF graph of Polk Speaker using PyDot.

3.5 Transformation of Objects to RDF Graphs

Figure 3.10 illustrates the procedure employed in this study for transforming Java object representations of the requirements and design components into RDF graphs for the requirements and components.

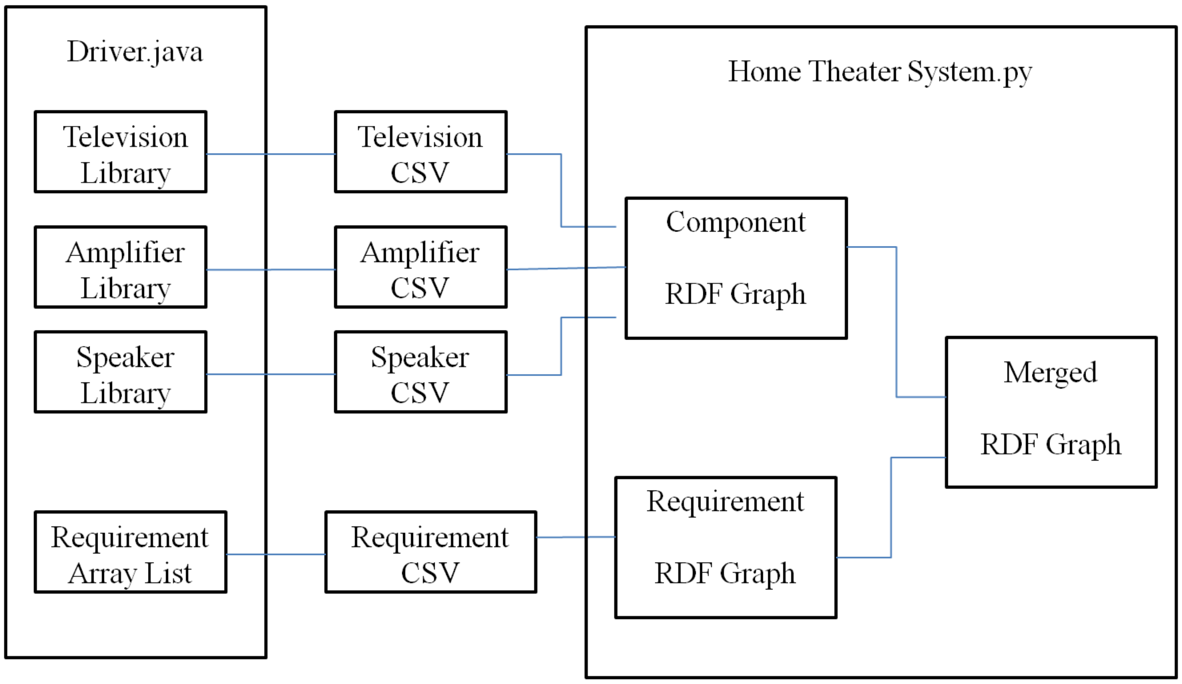


Figure 3.10: Software transformation from Java objects to Python RDF graphs .

The process of assembling the RDF graphs of design components to model the home theater system begins with developing a component interface in Java. The component interface describes methods that all design components are to encompass. Using Java, we create a television, amplifier, and speaker class that implement the component interface. Each Java class includes methods as well as parameters that describe the design component. In the Java driver class, we aim to create television objects then add parameters to the television objects, and assign a value to each parameter. As well, in the Java driver class, amplifier and speaker objects are also created. Further, a Hashset library is created for television, amplifier, and speaker to contain each design component object created in the driver class.

RDF structure is that of a Comma Separated Value (CSV) format [26]. Accordingly, subject and predicate is separated by a comma, as well predicate and object is separated by a comma. We seek, in particular, to generate a CSV file in Java for television, amplifier, and speaker. At first, we create a component CSV class in Java. Then, we compose a file in the class, followed by writing to the file for each television object within the television library. Specifically, we write to the file, the name of the television object concatenated with a comma and a parameter of television, concatenated with a comma and the value of the parameter. As well, in the component CSV class, we write to the file for each amplifier and speaker object within the amplifier and speaker library. Further, we generate a television, amplifier, and speaker CSV file in the driver class.

Television CSV file: This example illustrates the process of creating a television CSV file.

source code

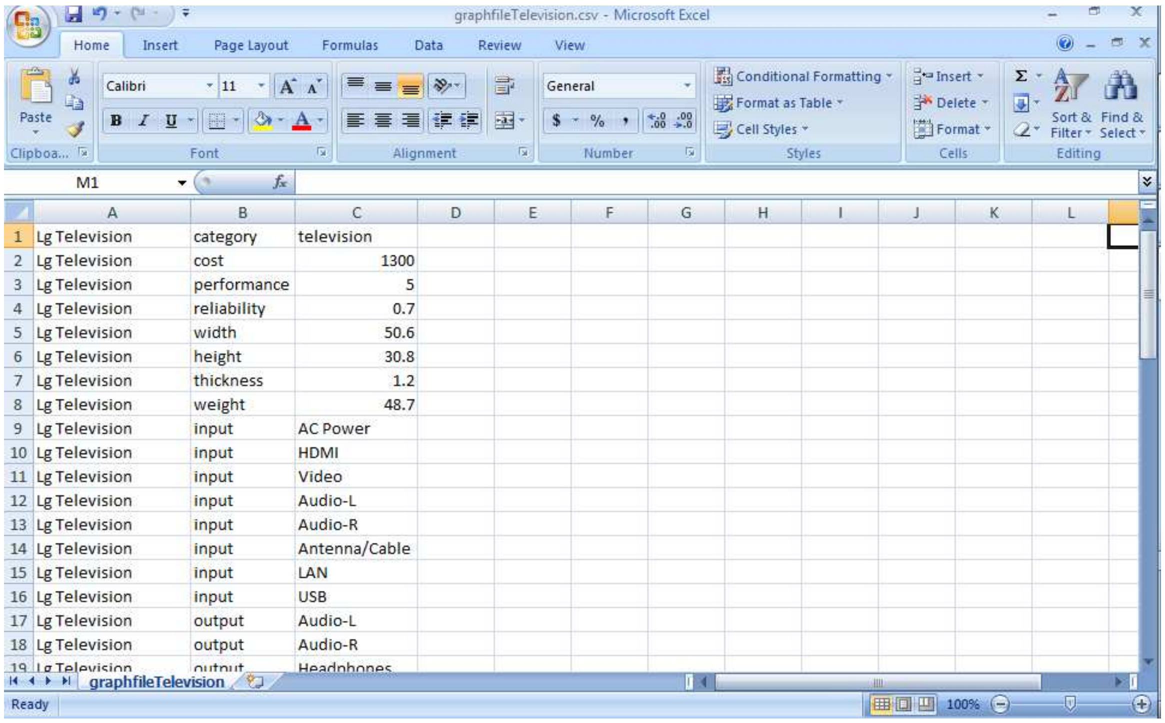
```
package javaBackend;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class GenerateComponentCsv {
   private File       rdf;
```

```
 private FileWriter cmp;

 // Make csv file

 public GenerateComponentCsv(String file){
    rdf = new File(file);
 }

 private File getRdf(){
    return rdf;
 }

 // Write to file for Tvs

 public void writeToFileTelevision(Library<Television> lib){

    // Check that the libary contains contents ...

    if(lib == null){
       System.out.print("Library is null");
       System.exit(0);
    }

    try {
       FileWriter tv = new FileWriter(getRdf());
       for(Television i : lib.getLibrary()) {
           tv.write(i.getName() + ",category," + "television" + "\n");
           tv.write(i.getName() + ",cost," + i.getCost()+ "\n");
           tv.write(i.getName() + ",performance," + i.getPerformance()+ "\n");
           tv.write(i.getName() + ",reliability," + i.getReliability()+ "\n");
           tv.write(i.getName() + ",width," + i.getWidth()+ "\n");
           tv.write(i.getName() + ",height," + i.getHeight()+ "\n");
           tv.write(i.getName() + ",thickness," + i.getThickness()+ "\n");
           tv.write(i.getName() + ",weight," + i.getWeight()+ "\n");

           for(int x = 0; x < i.getInput().length; x++){
               tv.write(i.getName() + ",input"+ "," + i.getInput()[x] + "\n");
           }

           for(int x = 0; x < i.getOutput().length; x++){
               tv.write(i.getName() + ",output"+ "," + i.getOutput()[x] + "\n");
           }
       }

       tv.close();
    } catch (IOException e) {
       e.printStackTrace();
    }
}
```

Points to note:

1. The GenerateComponentCsv class creates a CSV file.
2. The parameters of television design components are formatted into CSV.
3. The FileWriter tv object writes the television design components to a CSV file.

Figure 3.11 illustrates the structure of an Excel CSV file for a design component.



	A	B	C
1	Lg Television	category	television
2	Lg Television	cost	1300
3	Lg Television	performance	5
4	Lg Television	reliability	0.7
5	Lg Television	width	50.6
6	Lg Television	height	30.8
7	Lg Television	thickness	1.2
8	Lg Television	weight	48.7
9	Lg Television	input	AC Power
10	Lg Television	input	HDMI
11	Lg Television	input	Video
12	Lg Television	input	Audio-L
13	Lg Television	input	Audio-R
14	Lg Television	input	Antenna/Cable
15	Lg Television	input	LAN
16	Lg Television	input	USB
17	Lg Television	output	Audio-L
18	Lg Television	output	Audio-R
19	Lg Television	output	Headphones

Figure 3.11: Snapshot of the television CSV file in Excel.

The process of assembling the RDF graph of design requirements to model the home theater system begins with developing a requirement class in Java. The requirement class includes a constructor and methods as well as parameters that describe the design requirement. In the Java driver class, we aim to create requirement objects using the requirement constructor formulated in the requirement class. Further, an Arraylist is created to contain each design requirement object created in the driver class.

We seek, in particular, to generate a CSV file in Java for requirements. At first, we create a requirement CSV class in Java. Then, we compose a file in the class, followed by writing to the file for each requirement object within the requirement list. Specifically, we write to the file, the name of the requirement object concatenated with a comma and a parameter of requirement, concatenated with a comma and the value of the parameter. Further, we generate a requirement CSV file in the driver class.

Requirement CSV file: This example illustrates the process of creating a requirement CSV file.

source code

```
package javaBackend;

package javaBackend;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class GenerateRequirementCsv {
  private File rdf;
  private FileWriter cmp;

  // Make csv file
  public GenerateRequirementCsv(String file){
  rdf = new File(file);
   }
  private File getRdf(){
  return rdf;
  }

  // Write to file for Requirements

  public void writeToFileRequirement(ArrayList<Requirement> list){
  try {
  FileWriter req = new FileWriter(getRdf());
  int dependencyLength = 0;

  for(Requirement i: list){
  req.write(i.getName() + ",category," + "requirement" + "\n");
  req.write(i.getName() + ",id," + i.getName()+ "\n");
  req.write(i.getName() + ",title," + i.getTitle()+ "\n");
  req.write(i.getName() + ",level," + i.getLevel()+ "\n");
  req.write(i.getName() + ",derived by," + i.getDerivedBy()+ "\n");

  if(i.getDependOn() != null){
  dependencyLength= i.getDependOn().length;
  for(int x = 0; x < dependencyLength; x++){
  req.write(i.getName() + ",depends on"+"," + i.getDependOn()[x] + "\n");
     }
    }

}

req.close();

           } catch (IOException e) {
e.printStackTrace();
}
}
}
```

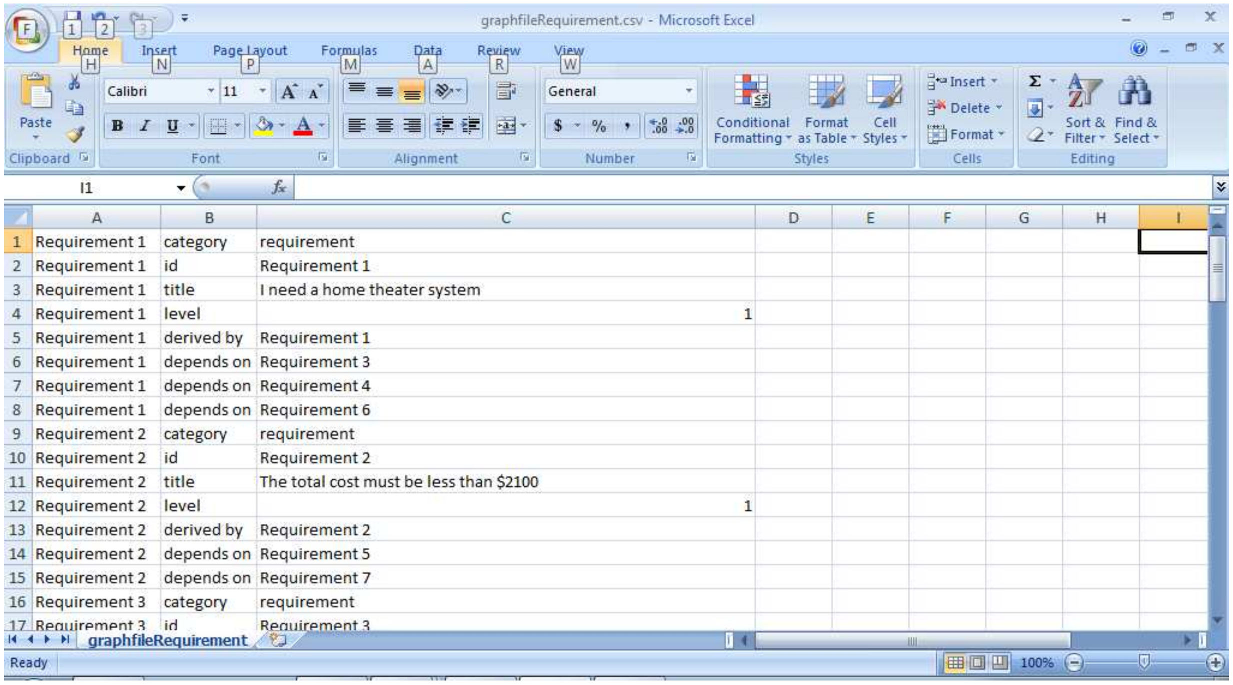
Points to note:

1. The GenerateRequirementCsv class creates a CSV file.

2. The properties of the design requirements are formatted into CSV.

3. The FileWriter tv object writes the design requirements to a CSV file.

Figure 3.12 illustrates the structure of an Excel CSV file for a design requirement.



	A	B	C
1	Requirement 1	category	requirement
2	Requirement 1	id	Requirement 1
3	Requirement 1	title	I need a home theater system
4	Requirement 1	level	1
5	Requirement 1	derived by	Requirement 1
6	Requirement 1	depends on	Requirement 3
7	Requirement 1	depends on	Requirement 4
8	Requirement 1	depends on	Requirement 6
9	Requirement 2	category	requirement
10	Requirement 2	id	Requirement 2
11	Requirement 2	title	The total cost must be less than $2100
12	Requirement 2	level	1
13	Requirement 2	derived by	Requirement 2
14	Requirement 2	depends on	Requirement 5
15	Requirement 2	depends on	Requirement 7
16	Requirement 3	category	requirement
17	Requirement 3	id	Requirement 3

Figure 3.12: Snapshot of the requirement CSV file in Excel.

The collection and arrangement of RDFs form a RDF graph. RDF graphs are useful to model systems. For instance, RDF graphs can model design components of a system. Design components of a system are represented by each subject of the RDF graph. Attributes of design components of a system are represented by each subjects predicate of the RDF graph. Values of attributes of design components of a system are represented by each predicates object of the RDF graph. As well, RDF graphs can also model design requirements of a system. Further, RDF graphs can be merged to form a RDF graph that models a system which detail design components and design requirements.

The central concern is assemblying a RDF graph in Python. This is done by calling the SimpleGraph function that store RDFs [26]. Here, we create a design component RDF graph and load the television, amplifier, and speaker CSV files generated in Java to the Python RDF graph. As well, a design requirement RDF graph is created and the requirement CSV file generated in Java is loaded to the Python RDF graph. Then, we merge the design component and the design requirement RDF graphs to produce a RDF graph, which models the home theater system.

Merged RDF Graph: This example illustrates the process of loading CSV files into RDF graphs and then merging the graphs together.

source code

```
Home Theater System.py.

from simplegraph_chapter3 import SimpleGraph

# *******R D F  G R A P H S*******

# Build component graph
componentRdf=SimpleGraph()
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileTelevision.csv')
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileAmplifier.csv')
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileSpeaker.csv')

# Build requirement graph
requirementRdf=SimpleGraph()
requirementRdf.load('C:\Users\Queen\workspace\HomeTheatreSystem\graphfileRequirement.csv')

#Merge Component and Requirement Graphs
mergegraph = SimpleGraph()
for sub, pred, obj in componentRdf.triples((None, None, None)):
    mergegraph.add((sub, pred, obj))
for sub, pred, obj in requirementRdf.triples((None, None, None)):
    mergegraph.add((sub, pred, obj))
```

Points to note:

1. The television, amplifier, and speaker CSV files are loaded into the component RDF graph.

2. The requirement CSV file is loaded into the requirement RDF graph.

3. The component and requirement RDF graphs are merged, forming the merge-graph of design components and design requirements.

3.6 Querying RDF Graphs

RDF graphs are queried with the intent of finding information within the graph. When modeling systems using RDF graphs, querying the RDF graph results in finding design components that satisfy design requirements. Essentially, querying systems through the application of RDF graphs, aim at verifying and validating system architecture.

Querying RDF graphs searches for subjects and objects that satisfy a RDF pattern [26]. This process begins with a RDF pattern. The structure of a RDF pattern is that of the assembly of RDF, discussed in section 3.2, yet different. The subject and/or object of a RDF pattern is *None*. If only the subject of a RDF pattern is *None*, then the RDF graph is queried to find subjects that are coupled with the predicate and object specified within the RDF pattern. If only the object of a RDF pattern is *None*, then the RDF graph is queried to find objects that are coupled with the subject and predicate specified within the RDF pattern. If the subject and object of a RDF pattern are *None*, then the RDF graph is queried to find subjects and objects that are coupled with the predicate specified within the RDF pattern. In addition, syntax such as *?x*, *?y*, or *?z* can substitute *None*, when querying RDF graphs [26].

Querying RDF graphs return subjects and objects that satisfy a RDF pattern [26]. The returned subjects and objects can also be queried to retrieve further information about the RDF graph. This is done by specifying another RDF pattern following the initial RDF pattern. We seek, in particular, to model systems as RDF graphs with intent to query the system to explore feasible design solutions.

Querying RDF Graph: This example illustrates the process of querying a television component RDF graph.

source code

```
from simplegraph_chapter3 import SimpleGraph

# Build tv component graph
tvComponentRdf=SimpleGraph()
tvComponentRdf.load('C:\Users\Nassar\workspace\Thesis Code\graphfileTelevision.csv')

#Query graph
print list(tvComponentRdf.triples((None, 'category', 'television')))
```

Points to note:

1. The SimpleGraph function creates a RDF graph.

2. The television design component CSV is loaded into the RDF graph.

3. The RDF graph is queried to find subjects which have predicate cateogry and object television.

3.7 Inference Rules for Design

An inference is an inferred conclusion derived from factual information. We aim to develop inference rules that further establish relationships between design components and design requirements to model systems. We expect that relationships between individual design components and relationships between individual design requirements will be achieved by applying inference rules to RDF graphs.

An inference rule is a class in Python that contain two functions. The first function of inference rule class is *getqueries*. The *getqueries* function queries RDF graphs for the RDFs specified within the function, similar to querying RDF graphs discussed in section 3.3. The second function of inference class is *maketriples*. Here, the parameters returned from *getqueries* are passed to the *maketriples* function. This, in turn, allows for the assembly of a RDF. Through the *maketriples* method, we seek, in particular, to introduce an inferred relationship between subjects of RDFs within the RDF graph. Accordingly, the developed inference rule class is applied to the RDF graph, followed by querying the RDF graph [26].

Inference rules are useful to infer relationships of design requirements. This is appropriate for modeling systems because inference rules establish an implicit correlation. As well, inference rules can also be useful to infer relationships of design components. Furthermore, inference rules are most valuable to infer relationships between design requirements and design components. This is effective for modeling systems because this particular inference rule generate feasible design solutions.

Inference Rule: This example illustrates the process of creating an inference rule for connecting a television to an amplifier.

source code

```
InferenceRule.py.

class InferenceRule:
    def getqueries(self):
        return None

    def maketriples(self,binding):
        return self._maketriples(**binding)

# Rule: television connects to amplifier
class televisionToAmplifier(InferenceRule):
    def getqueries(self):
        tvToAmp=[('?x','category','television'),
                      ('?y','category','amplifier')]

        return [tvToAmp]

    def _maketriples(self,x,y):
        return [ ( x, 'connects to', y ) ]


Home Theater System.py.

# Apply Tv to Amp Rule to RDF Graph...

tvamp = televisionToAmplifier()
componentRdf.applyinference(tvamp)
```

Points to note:

1. The *getqueries* function searches for two RDF patterns.

2. The *maketriples* function creates a new RDF, which establish a relationship between television and amplifier.

3. The developed inference rule is applied to the RDF graph.

Chapter 4

Synthesis of Pareto Optimal Design Alternatives

This chapter presents methodology, algorithms, and numerical experiments for the automated synthesis of Pareto Optimal Design Alternatives from families of potentially good design solutions.

4.1 Methodology and Definition

Engineering systems are typically designed to satisfy the needs of multiple stakeholders. Each stakeholder will have: (1) A set of design concerns/functional requirements, (2) Levels of performance that need to be met, and (3) A budget. The challenge that we face is finding ways to systematically and efficiently find designs that balance the attributes of economy, performance, reliability/quality, and use of resources, subject to physical, regulatory, design and implementation constraints imposed by the participating stakeholders and domains.

Purpose of a Trade Study. The purpose of a trade study is to examine the relative value and sensitivity of attributes associated with the design's measure of effectiveness. This information is then used to guide decision making relating to the selection and treatment of design alternatives. In some applications (e.g., design of a building), multiple objectives are resolved and constraints are satisfied to find a singular one-off design. However, in cases where a point customer does not exist (e.g., customers for the purchase of electronic/computer equipment), objectives and constraints are resolved and satisfied to find a family of good design solutions.

The so-called "sweet spot in design" occurs when improvements in one aspect of system effectiveness can only occur when it is traded against decreases in one or more other aspects of system effectiveness. Understanding the trades that a particular design problem offers can be an exceedingly difficult problem – generally speaking, more functionality usually means less economy (i.e., you can expect to pay more for additional system functionality). More functionality can also mean slower time-to-market (i.e., it takes time develop and test a new technology). Improved performance usually means less economy (e.g., a system might cost more because components with high-end performance are expensive). And shorter time-to-market means less economy (e.g., due to increased costs in shipping). To determine which combination of design options is best for a particular situation, designers need to resolve conflicts by indicating their preferences. Instead of trying to compute a single optimal solution, we begin the problem solving procedure by partitioning the feasible design space into regions of high technical efficiency and regions of inferior performance. The preferred design is one that is technically efficient (or non-inferior). The inferior solutions are removed from further consideration.

Mathematical Definition of Non-Inferior Solutions. Given a set of feasible solutions X, the set of non-inferior (or non-dominated) solutions is denoted S and defined as follows:

S = x : x $\in$ X, there exists no other $x^* \in$ X such that $f_q(x^*) > f_q(x)$ for some $q \in \{1 \cdots p\}$ and $f_k(x^*) \geq f_k(x)$ for all $k \neq q$.

The plain English interpretation of this definition is as follows: Let S be the set of solutions x for which we can demonstrate no better solutions exist. As one moves from one non-dominated solution to another and one objective function improves, then one or more of the other objective functions **must decrease in value.** Regardless of how the design objectives are prioritized, the non-dominated design solutions (also called pareto optimal design solutions and/or efficient frontier solutions) will lie along the boundary of the feasible domain.

4.2 Computation of Pareto-Optimal Design Alternativess

The step-by-step procedure for the computation of Pareto-Optimal design alternatives is as follows:

Case 1: Minimize X-Axis and Minimize Y-Axis.

1. Consider all coordinate points of the scatter plot Pareto-Optimal and add the points into a two-dimensional Pareto-Optimal data array. The first columm will store objective values along the x- axis. The second columm will store objective values along the y- axis.

2. Find the minimum coordinate point of the x-axis on the scatter plot.

3. *Pop* all coordinate points from the Pareto-Optimal array which are greater than and equal to the y-axis value of the minimum coordinate point.

4. For coordinate points remaining in the Pareto-Optimal array with the same x-axis value, the coordinate point with the minimum y-axis value should remain and others must be *popped* from the Pareto-Optimal array.

5. For coordinate points remaining in the Pareto-Optimal array with the same y-axis value, the coordinate point with the minimum x-axis value should remain and others must be *popped* from the Pareto-Optimal array.

6. The remaining coordinate points in the Pareto-Optimal array are considered Pareto-Optimal.

Case 2: Minimize X-Axis and Maximize Y-Axis.

1. Consider all coordinate points of the scatter plot Pareto-Optimal and add the points into a Pareto-Optimal array.

2. Find the minimum coordinate point of the x-axis on the scatter plot.

3. *Pop* all coordinate points from the Pareto-Optimal array which are less than and equal to the y-axis value of the minimum coordinate point.

4. For coordinate points remaining in the Pareto-Optimal array with the same x-axis value, the coordinate point with the maximum y-axis value should remain and others must be *popped* from the Pareto-Optimal array.

5. For coordinate points remaining in the Pareto-Optimal array with the same y-axis value, the coordinate point with the minimum x-axis value should remain and others must be *popped* from the Pareto-Optimal array.

6. The remaining coordinate points in the Pareto-Optimal array are considered Pareto-Optimal.

Case 3: Maximize X-Axis and Maximize Y-Axis.

1. Consider all coordinate points of the scatter plot Pareto-Optimal and add the points into a Pareto-Optimal array.

2. Find the maximum coordinate point of the x-axis on the scatter plot.

3. *Pop* all coordinate points from the Pareto-Optimal array which are less than and equal to the y-axis value of the minimum coordinate point.

4. For coordinate points remaining in the Pareto-Optimal array with the same x-axis value, the coordinate point with the maximum y-axis value should remain and others must be *popped* from the Pareto-Optimal array.

5. For coordinate points remaining in the Pareto-Optimal array with the same y-axis value, the coordinate point with the maximum x-axis value should remain and others must be *popped* from the Pareto-Optimal array.

6. The remaining coordinate points in the Pareto-Optimal array are considered Pareto-Optimal.

Case 4: Maximize X-Axis and Minimize Y-Axis.

1. Consider all coordinate points of the scatter plot Pareto-Optimal and add the points into a Pareto-Optimal array.

2. Find the maximum coordinate point of the x-axis on the scatter plot.

3. *Pop* all coordinate points from the Pareto-Optimal array which are greater than and equal to the y-axis value of the minimum coordinate point.

4. For coordinate points remaining in the Pareto-Optimal array with the same x-axis value, the coordinate point with the minimum y-axis value should remain and others must be *popped* from the Pareto-Optimal array.

5. For coordinate points remaining in the Pareto-Optimal array with the same y-axis value, the coordinate point with the maximum x-axis value should remain and others must be *popped* from the Pareto-Optimal array.

6. The remaining coordinate points in the Pareto-Optimal array are considered Pareto-Optimal.

Implementation of Pareto Optimal Computations in Python: The following code illustrates the process of finding the Pareto optimal point of a trade-off analysis plot. Pareto optimal design alternatives can be computed for four cases;

1. Minimize both objectives of the trade-off analysis plot.

2. Minimize the x-axis objective, maximize the y-axis objective.

3. Minimize both the x- and y-axis objectives.

4. Maximize the x-axis objective and minimize the y-axis objective.

The Pareto-Optimal code is a structure of if-else statements used to find an array of Pareto-Optimal point(s) of a graph. Each if statement include the procedures for either minimizing or maximizing the x-axis and either minimizing or maximizing the y-axis of a graph. First, the user provides both the x and y-axis objectives of a graph in the form of an array. Then, the code prompts the user to input whether the x-axis objective is minimized or maximized and whether the y-axis objective is minimized or maximized. Next, the if-else statements are executed according to the user input. As a result, a graph is shown illustrating the user's provided x and y-axis objectives, in addition to the computed Pareto-Optimal point(s) of the graph.

source code

```
import matplotlib.pyplot as plt
import numpy as np
from pylab import *

x_objective = [ ]
y_objective = [ ]

# ********** P A R E T O  O P T I M A L **********

# Prompt User

# Ask user for the intent of objective 1 and store

input1 = int(raw_input("Is the x-axis objective min[0] or max[1] : "))

# Ask user for the intent of objective 2 and store

input2 = int(raw_input("Is the y-axis objective min[0] or max[1] : "))

if input1 == 0 and input2 == 0:
    Data   = np.array((x_objective,y_objective))
    points = zip(*Data)
    temp   = points[:]

    # temp holds all the points, lets call this Pareto

    for t in range(0,len(points)):
        x, y = zip(*temp)
        minPairs = [temp[i] for i,a in enumerate(x) if a == min(x)]
        returnPair = minPairs
```

```
if len(returnPair) == 1:
    m,n = zip(*returnPair)

    test_a = [temp[i] for i, b in enumerate(y) if b >= n]
    popPairs = test_a

    for i in popPairs:
        if i == returnPair[0]:
            popPairs.remove(i)

    for i in popPairs:
        temp.remove(i)

    if len(temp) == 2:
        break

if len(returnPair) > 1:
    returnPair = returnPair
    m,n = zip(*returnPair)
    test_a = [returnPair[i] for i, b in enumerate(n) if b == min(n)]
    popPairs = test_a

    m,n = zip(*popPairs)
    test_a1 = [temp[i] for i, b in enumerate(y) if b >= n]
    popPairs_a1 = test_a1

    for i in popPairs_a1:
        if i == popPairs[0]:
            popPairs_a1.remove(i)

    for i in popPairs_a1:
        temp.remove(i)

if len(temp) == 1:
    break

x,y = zip(*temp)

space = {}
for i in set(y):
    space[i] = y.count(i)
    while y.count >1:

        test_b = [temp[i] for i, c in enumerate(y) if y.count(c)>1]

        popPairs_b = test_b

        if len(popPairs_b) == 0:
            test_b = [temp[i] for i, c in enumerate(x) if x.count(c)>1]
            popPairs_b = test_b
            a,b = zip(*popPairs_b)
            test_b1 = [popPairs_b[i] for i,d in enumerate(b) if d == max(b)]
            popPairs_b1 = test_b1
```

```
                  for i in popPairs_b1:
                      temp.remove(i)

                  break


              a,b = zip(*popPairs_b)
              test_b1 = [popPairs_b[i] for i,d in enumerate(a) if d == max(a)]
              popPairs_b1 = test_b1

              for i in popPairs_b1:
                  temp.remove(i)

              if len(popPairs_b) ==2:
                  break

              x,y = zip(*temp)
          break
      break

elif input1 == 0 and input2 == 1:
  Data   = np.array(( x_objective, y_objective ))
  points = zip(*Data)
  temp   = points[:]

  # temp holds all the points, lets call this Pareto

  for t in range(0,len(points)):
      x, y = zip(*temp)
      minPairs   = [temp[i] for i,a in enumerate(x) if a == min(x)]
      returnPair = minPairs

      if len(returnPair) == 1:
          m,n = zip(*returnPair)

          test_a = [temp[i] for i, b in enumerate(y) if b <= n]
          popPairs = test_a

          for i in popPairs:
              if i == returnPair[0]:
                  popPairs.remove(i)

          for i in popPairs:
              temp.remove(i)

          if len(temp) == 2:
              break

      if len(returnPair) > 1:
          returnPair = returnPair
          m,n = zip(*returnPair)
          test_a = [returnPair[i] for i, b in enumerate(n) if b == max(n)]
          popPairs = test_a
          m,n = zip(*popPairs)
          test_a1 = [temp[i] for i, b in enumerate(y) if b <= n]
          popPairs_a1 = test_a1

          for i in popPairs_a1:
              if i == popPairs[0]:
                  popPairs_a1.remove(i)

          for i in popPairs_a1:
              temp.remove(i)

      if len(temp) == 1:
          break

      x,y = zip(*temp)

      space = {}
      for i in set(y):
          space[i] = y.count(i)
          while y.count >1:
              test_b = [temp[i] for i, c in enumerate(y) if y.count(c)>1]

              popPairs_b = test_b

              if len(popPairs_b) == 0:
                  test_b = [temp[i] for i, c in enumerate(x) if x.count(c)>1]
                  popPairs_b = test_b
                  a,b = zip(*popPairs_b)
                  test_b1 = [popPairs_b[i] for i,d in enumerate(b) if d == min(b)]
                  popPairs_b1 = test_b1

                  for i in popPairs_b1:
                      temp.remove(i)

                  break

              a,b = zip(*popPairs_b)
              test_b1 = [popPairs_b[i] for i,d in enumerate(a) if d == max(a)]
              popPairs_b1 = test_b1

              for i in popPairs_b1:
                  temp.remove(i)

              if len(popPairs_b) ==2:
                  break

              x,y = zip(*temp)
          break
      break

elif input1 == 1 and input2 == 1:
  Data   = np.array((x_objective,y_objective))
  points = zip(*Data)
```

```
temp   = points[:]

# temp holds all the points, lets call this Pareto

for t in range(0,len(points)):
    x, y = zip(*temp)
    minPairs = [temp[i] for i,a in enumerate(x) if a == max(x)]
    returnPair = minPairs

    if len(returnPair) == 1:
        m,n = zip(*returnPair)

        test_a = [temp[i] for i, b in enumerate(y) if b <= n]
        popPairs = test_a

        for i in popPairs:
            if i == returnPair[0]:
                popPairs.remove(i)

        for i in popPairs:
            temp.remove(i)

        if len(temp) == 2:
            break

    if len(returnPair) > 1:
        returnPair = returnPair
        m,n = zip(*returnPair)
        test_a = [returnPair[i] for i, b in enumerate(n) if b == max(n)]
        popPairs = test_a

        m,n = zip(*popPairs)
        test_a1 = [temp[i] for i, b in enumerate(y) if b <= n]
        popPairs_a1 = test_a1

        for i in popPairs_a1:
            if i == popPairs[0]:
                popPairs_a1.remove(i)

        for i in popPairs_a1:
            temp.remove(i)

    if len(temp) == 1:
        break

    x,y = zip(*temp)

    space = {}
    for i in set(y):
        space[i] = y.count(i)
        while y.count >1:

            test_b = [temp[i] for i, c in enumerate(y) if y.count(c)>1]
```

```
              popPairs_b = test_b

              if len(popPairs_b) == 0:
                  test_b = [temp[i] for i, c in enumerate(x) if x.count(c)>1]
                  popPairs_b = test_b
                  a,b = zip(*popPairs_b)
                  test_b1 = [popPairs_b[i] for i,d in enumerate(b) if d == min(b)]
                  popPairs_b1 = test_b1

                  for i in popPairs_b1:
                      temp.remove(i)

                  break

              a,b = zip(*popPairs_b)
              test_b1 = [popPairs_b[i] for i,d in enumerate(a) if d == min(a)]
              popPairs_b1 = test_b1

              for i in popPairs_b1:
                  temp.remove(i)

              if len(popPairs_b) ==2:
                  break

              x,y = zip(*temp)
          break
      break

elif input1 == 1 and input2 == 0:
  Data   = np.array((x_objective,y_objective))
  points = zip(*Data)
  temp   = points[:]

  # temp holds all the points, lets call this Pareto

  for t in range(0,len(points)):
      x, y = zip(*temp)
      minPairs = [temp[i] for i,a in enumerate(x) if a == max(x)]
      returnPair = minPairs

      if len(returnPair) == 1:
          m,n = zip(*returnPair)

          test_a = [temp[i] for i, b in enumerate(y) if b >= n]
          popPairs = test_a

          for i in popPairs:
              if i == returnPair[0]:
                  popPairs.remove(i)

          for i in popPairs:
              temp.remove(i)

          if len(temp) == 2:
```

```
        break

if len(returnPair) > 1:
    returnPair = returnPair
    m,n = zip(*returnPair)
    test_a = [returnPair[i] for i, b in enumerate(n) if b == min(n)]
    popPairs = test_a

    m,n = zip(*popPairs)
    test_a1 = [temp[i] for i, b in enumerate(y) if b >= n]
    popPairs_a1 = test_a1

    for i in popPairs_a1:
        if i == popPairs[0]:
            popPairs_a1.remove(i)

    for i in popPairs_a1:
        temp.remove(i)

if len(temp) == 1:
    break

x,y = zip(*temp)

space = {}
for i in set(y):
    space[i] = y.count(i)
    while y.count >1:

        test_b = [temp[i] for i, c in enumerate(y) if y.count(c)>1]

        popPairs_b = test_b

        if len(popPairs_b) == 0:
            test_b = [temp[i] for i, c in enumerate(x) if x.count(c)>1]
            popPairs_b = test_b
            a,b = zip(*popPairs_b)
            test_b1 = [popPairs_b[i] for i,d in enumerate(b) if d == max(b)]
            popPairs_b1 = test_b1

            for i in popPairs_b1:
                temp.remove(i)

            break

        a,b = zip(*popPairs_b)
        test_b1 = [popPairs_b[i] for i,d in enumerate(a) if d == min(a)]
        popPairs_b1 = test_b1

        for i in popPairs_b1:
            temp.remove(i)

        if len(popPairs_b) ==2:
            break
```

```
                x,y = zip(*temp)
            break
        break

else:
    print("You failed to min or max either one or both objectives")

# ******* P L O T  T R A D E - O F F  A N A L Y S I S *******
#                          AND
#      ******* P A R E T O  O P T I M A L *******

print ''
print "========================="
print "Pareto Optimal Points"
print "========================="
print ''


print "Pareto Point(s):",temp

if input1 == 0 and input2 == 0:
    # Min x-axis & Min y-axis
    fig, (ax0) = plt.subplots(ncols=1)
    x = x_objective
    y = y_objective
    a,b = zip(*temp)

    ax0.plot(x, y, marker ='.',markersize= 8, linewidth=0)
    ax0.plot(a, b, marker ='*',markersize= 18, linewidth=0)
    ax0.axis([min(x)-1, max(x)+1, min(y)-1, max(y)+1])
    ax0.set_xlabel('Objective 1')
    ax0.set_ylabel('Objective 2')

    fig.suptitle('Trade-off Analysis: Minimize Objective 1 & Minimize Objective 2')
    plt.tight_layout(pad=2)
    plt.show()

elif input1 == 0 and input2 == 1:
    # Min x-axis & Max y-axis
    fig, (ax1) = plt.subplots(ncols=1)
    x = x_objective
    y = y_objective
    a,b = zip(*temp)

    ax1.plot(x, y, marker ='.',markersize= 8, linewidth=0)
    ax1.plot(a, b, marker ='*',markersize= 18, linewidth=0)
    ax1.axis([min(x)-1, max(x)+1, min(y)-1, max(y)+1])
    ax1.set_xlabel('Objective 1')
    ax1.set_ylabel('Objective 2')

    fig.suptitle('Trade-off Analysis: Minimize Objective 1 & Maximize Objective 2')
    plt.tight_layout(pad=2)
    plt.show()
```

```
elif input1 == 1 and input2 == 1:
    # Max x-axis & Max y-axis
    fig, (ax2) = plt.subplots(ncols=1)
    x = x_objective
    y = y_objective
    a,b = zip(*temp)

    ax2.plot(x, y, marker ='.',markersize= 8, linewidth=0)
    ax2.plot(a, b, marker ='*',markersize= 18, linewidth=0)
    ax2.axis([min(x)-1, max(x)+1, min(y)-1, max(y)+1])
    ax2.set_xlabel('Objective 1')
    ax2.set_ylabel('Objective 2')

    fig.suptitle('Trade-off Analysis: Maximize Objective 1 & Maximize Objective 2')
    plt.tight_layout(pad=2)
    plt.show()

elif input1 == 1 and input2 == 0:
    # Max x-axis & Min y-axis
    fig, (ax3) = plt.subplots(ncols=1)
    x = x_objective
    y = y_objective
    a,b = zip(*temp)

    ax3.plot(x, y, marker ='.',markersize= 8, linewidth=0)
    ax3.plot(a, b, marker ='*',markersize= 18, linewidth=0)
    ax3.axis([min(x)-1, max(x)+1, min(y)-1, max(y)+1])
    ax3.set_xlabel('Objective 1')
    ax3.set_ylabel('Objective 2')

    fig.suptitle('Trade-off Analysis: Maximize Objective 1 & Minimize Objective 2')

    plt.tight_layout(pad=2)
    plt.show()

print ""
print ""
print "======================== Finished ========================"
print ""
```

In this script of Python code, all four cases are handled. If else statements are used to find the minimum and maximum of the objectives.

4.3 Trade-Space Visualization with Python

A Graphical User Interface (GUI) is an application that allows the user to interact with the software environment. Python GUI is useful to display valuable information developed within Python classes.

Python GUI is appropriate for modeling systems to permit the user to visually interact with the design requirements and design components of the system. We seek, in particular, to develop a Python GUI that visualizes the trade-space of feasible design solutions.

The trade-space visualization of the Python GUI illustrates three interactive trade-off plots for each feasible system design combination generated. The first plot displays the trade-off analysis of cost verse performance. The second plot displays the trade-off analysis of cost verse reliability. The third plot displays the trade-off analysis of performance verse reliability. There are three criteria, cost, performance, and reliability, which each feasible system design combination is evaluated against. Cost is calculated by summing the cost of individual design components. For the purpose of this project, performance is calculated by summing the performance of individual design components. Assuming the design components are connected in series, reliability is calculated by multiplying the reliability of each design component.

4.4 Numerical Experiments

The Python code for computing sets of Pareto-Optimal design solutions was exercised by computing design solutions for two test cases:

Test Shape A: Square

```
x_objective = [5,5,5,6,6,7,7,7]
y_objective = [5,6,7,5,7,5,6,7]
```

Test Shape B: Diamond

```
x_objective = [1,3,3,5,5,7,7,9]
y_objective = [5,7,3,9,1,7,3,5]
```

The results for Test Shape A are shown in Figures 4.1 - 4.4. The results for Test Shape B are shown in Figures 4.5 - 4.8. In each case the Pareto-Optimal design points are displayed as an enlarged two-dimensional diamond.

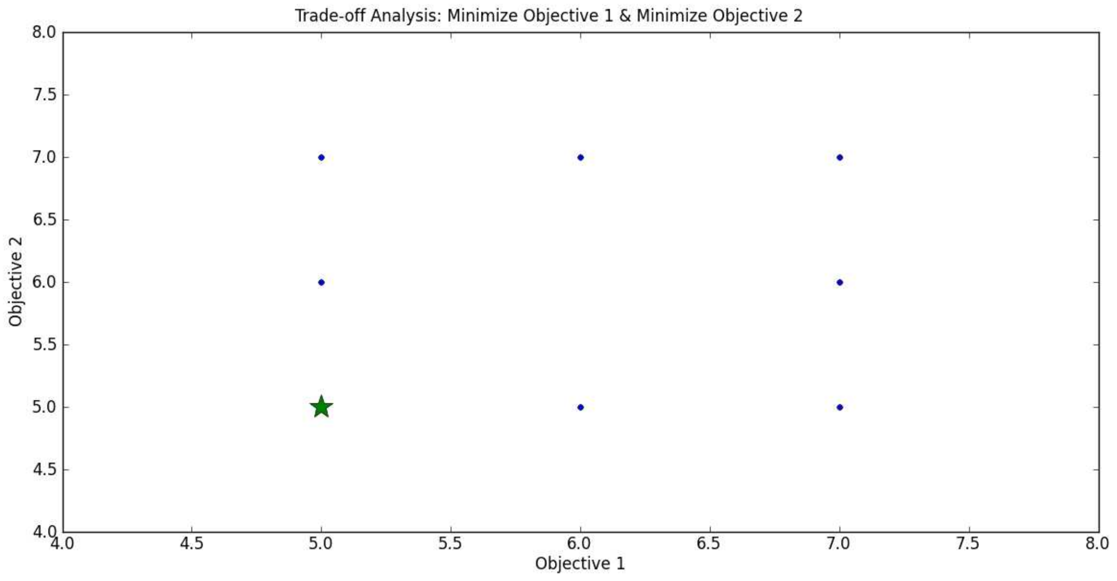


Figure 4.1: Test shape A trade-off analysis GUI of minimizing Objective 1 and minimizing Objective 2 with starred Pareto optimal point.

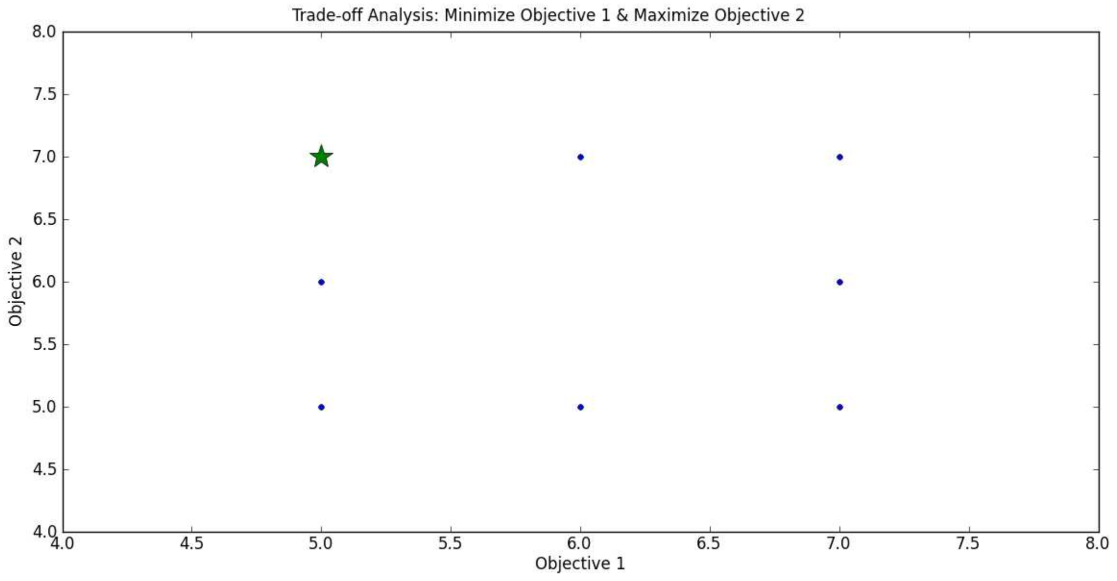


Figure 4.2: Test shape A rade-off analysis GUI of minimizing Objective 1 and maximizing Objective 2 with starred Pareto optimal point.

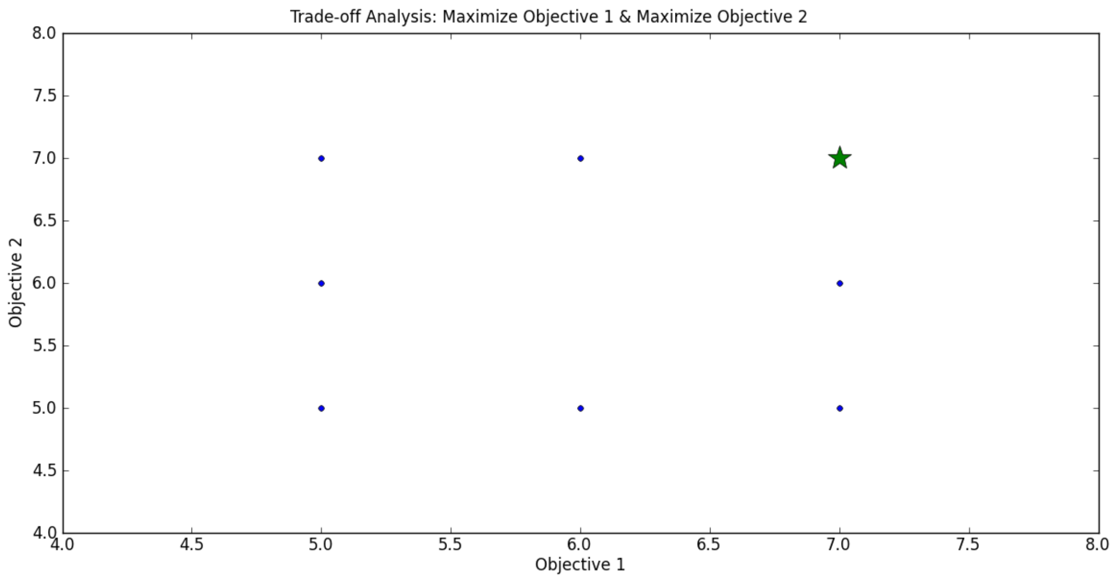


Figure 4.3: Test shape A trade-off analysis GUI of maximizing Objective 1 and maximizing Objective 2 with starred Pareto optimal point.

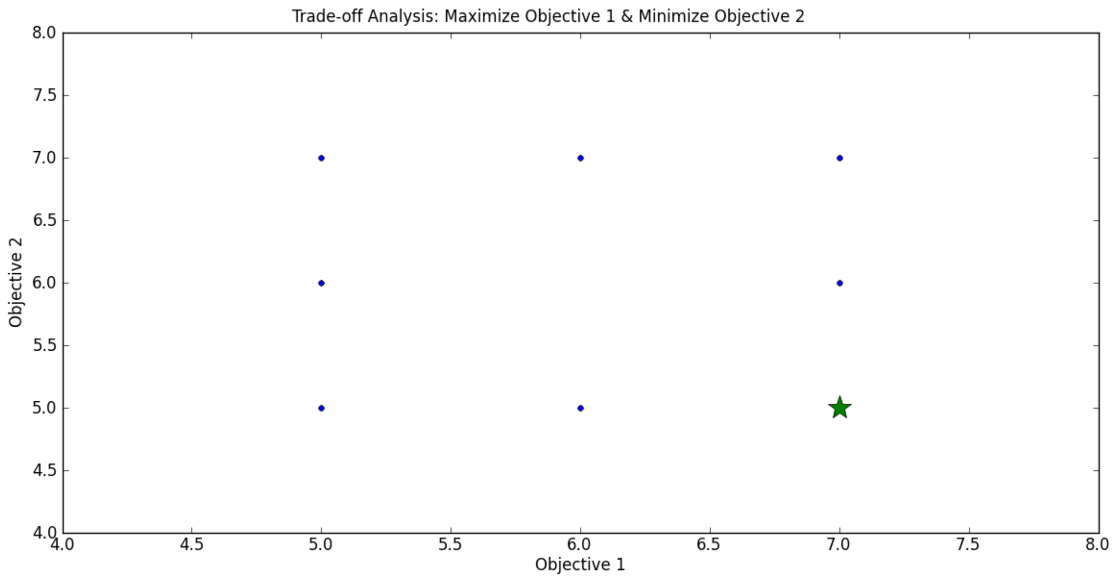


Figure 4.4: Test shape A trade-off analysis GUI of maximizing Objective 1 and minimizing Objective 2 with starred Pareto optimal point.

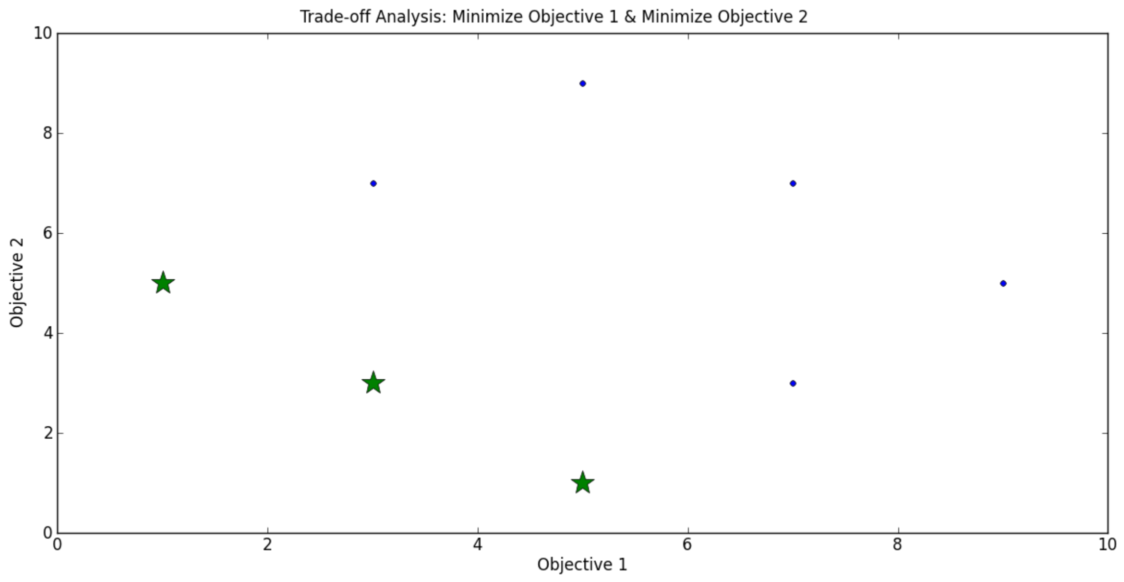


Figure 4.5: Test shape B trade-off analysis GUI of minimizing Objective 1 and minimizing Objective 2 with starred Pareto optimal points.

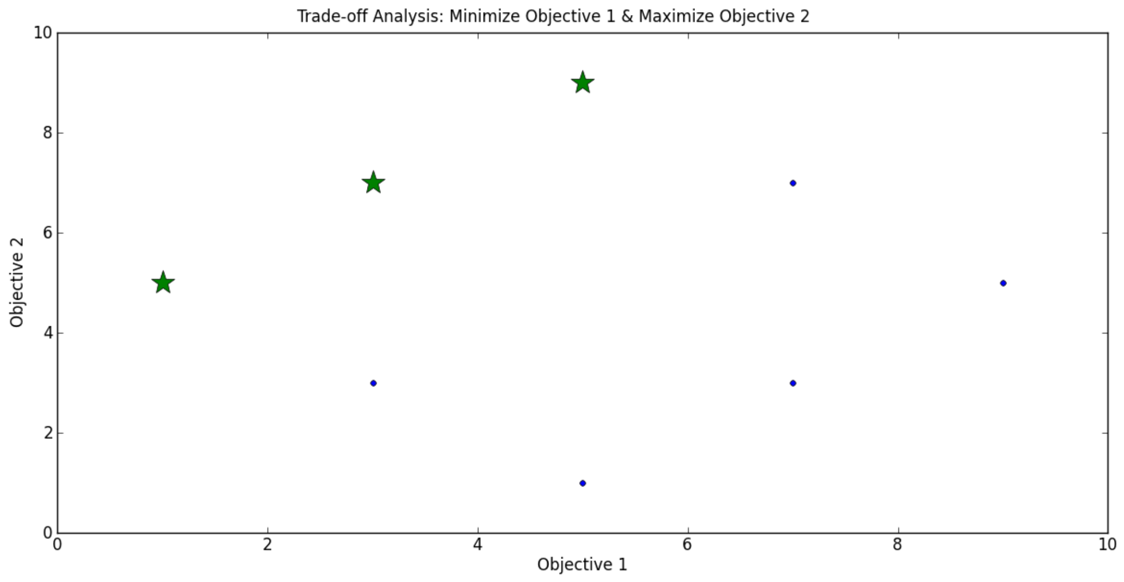


Figure 4.6: Test shape B rade-off analysis GUI of minimizing Objective 1 and maximizing Objective 2 with starred Pareto optimal points.

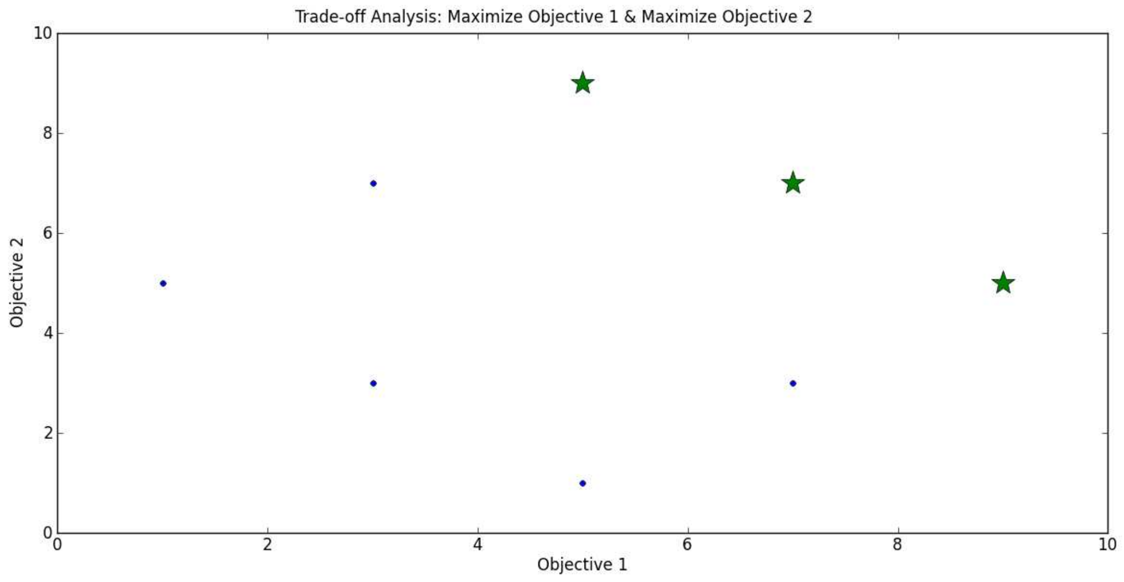


Figure 4.7: Test shape B trade-off analysis GUI of maximizing Objective 1 and maximizing Objective 2 with starred Pareto optimal points.

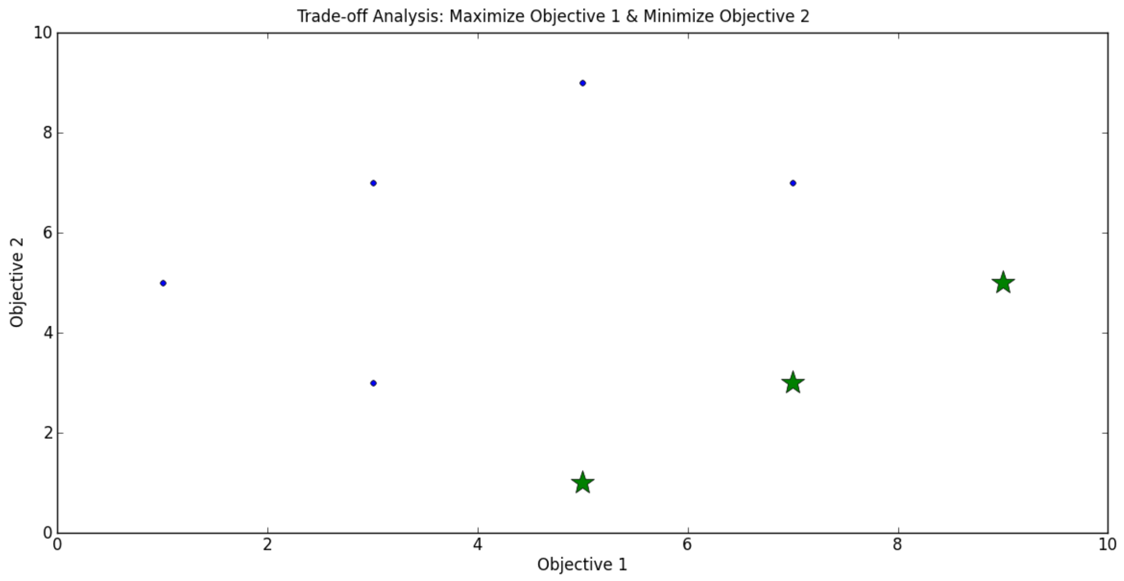


Figure 4.8: Test shape B trade-off analysis GUI of maximizing Objective 1 and minimizing Objective 2 with starred Pareto optimal points.

Chapter 5

Home Theater Design and TradeOff Analysis

5.1 Problem Statement and Solution Procedure

This chapter describes the computational procedure and results for design synthesis and trade-space analysis of the Home Theater Design Problem. The left-hand side of Figure 5.1 summarizes the sequence of inference rule evaluations that will generate ensembles of feasible design solutions. The right-hand side of Figure 5.1 summarizes the three types of trades that will be examined in this study. Figure 5.2 illustrates the level and dependencies of each requirement of the home theater design.

We seek a computationally efficient procedure for the synthesis of design solutions by casting the problem as a sequence of inferences. Inference rules for system-level development will establish premissable relationships between design components as well as design components and design requirements. Each design requirement will be translated into a Python inference rule according to the requirement description, level, and dependencies among requirements.

Solution Procedure. The step-by-step procedure for synthesizing potentially good design solutions and assessing their performance, reliability and cost is as follows:

1. Assemble the graph model of the requirements from a CSV file of requirements

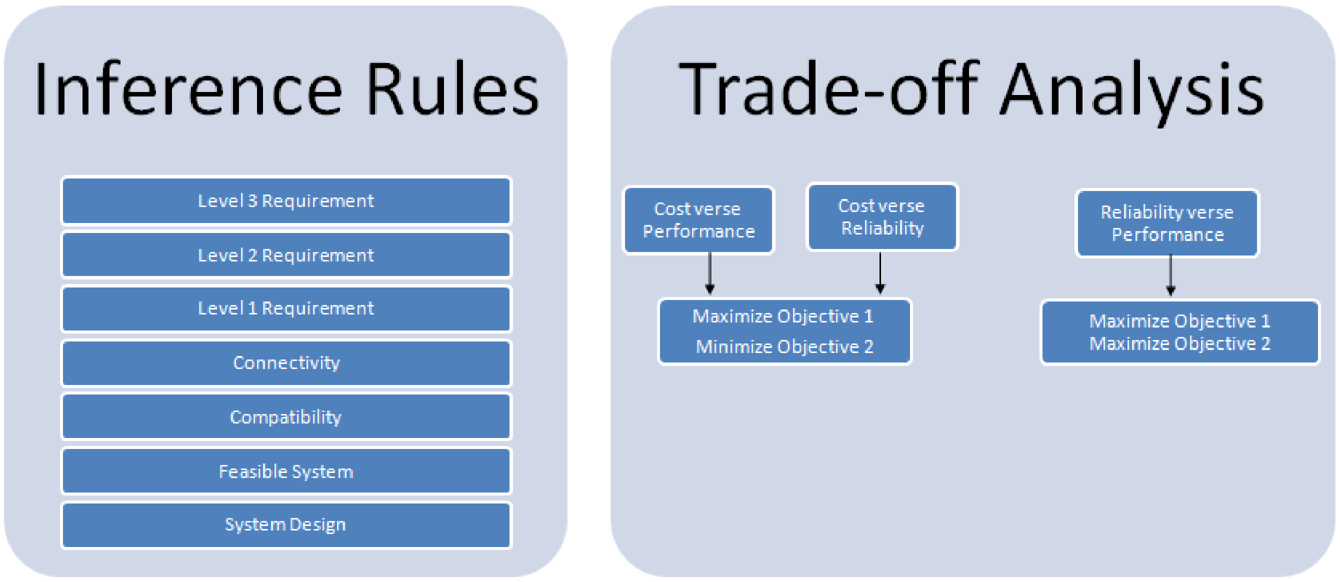


Figure 5.1: Flowchart for home theater design and tradeoff analysis.

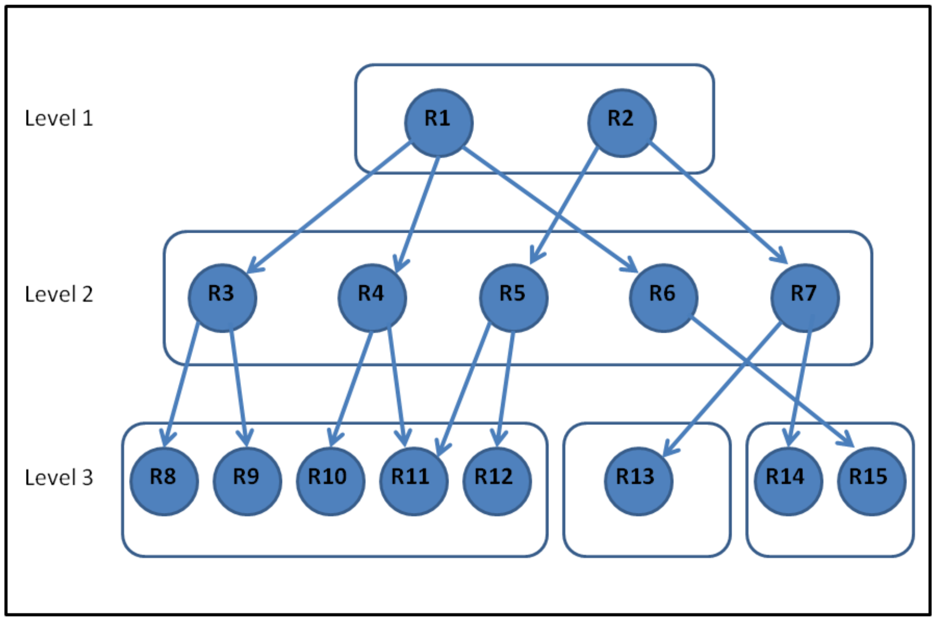


Figure 5.2: Requirement Hierarchy.

data.

2. Assemble the component graph by reading in CSV files for the television, amplifier and speakers.

3. Define and apply the System-Level Architecture Connectivity Rules to the component graph: TVtoAmp Rule and AmptoSpeaker Rule.

4. Define and apply the Component Compatibility Rules to the component graph: TV/Amp Compatibility Rule and Amp/Speaker Compatibility Rule

5. Merge the component and requirement graphs

6. Define and apply the Feasible System Configuration Rule.

7. Define and verify the Level 3 Requirements Rules: Req15 Rule, Req14 Rule, Req13 Rule, Req12 Rule, Req11 Rule, Req10 Rule, Req9 Rule, and and Req8 Rule.

8. Define and verify the Level 2 Requirements Rules: Req7 Rule, Req6 Rule, Req5 Rule, Req4 Rule, and Req3 Rule.

9. Define and verify the Level 1 Requirements Rules: Req2 Rule and Req1 Rule.

10. Define and apply the Feasible System Design Rule.

11. Retrieve all of the Feasible Design Combinations from the inferred model.

12. Create Trade-off Analysis Plots: Performance vs Cost, Reliability vs Cost and Performance vs Reliability.

The feasible system configurations (see Step 6) rule identifies combinations of amplifier, speaker and television components that are compatible, and match the architectural requirements described in Chapter 2. The feasible system design rule (see step 10) prunes the list system cofigurations by removing all combinations of components that fail one or more of the Level 3 requirements.

5.2 Initializing the Design Problem

The home theater design problem is initialized once the requirement and components graphs are assembled. The requirement CSV file is loaded into the SimpleGraph function, which we have identified as the requirements model graph.

source code

```
GUI and Trade-Space Visualization.py.

# Build the requirements model graph ...

requirementRdf=SimpleGraph()
requirementRdf.load('C:\Users\Queen\workspace\HomeTheatreSystem\
                     graphfileRequirement.csv')

# Build the design components graph ...

componentRdf=SimpleGraph()
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileTelevision.csv')
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileAmplifier.csv')
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileSpeaker.csv')
```

The requirements model graph has 44 unique vertices and 130 edges. The television, amplifier, and speaker CSV files are loaded into the SimpleGraph function, which we have identified as the design components graph. The design components graph has 57 unique vertices and 115 edges. The following script of Python code initializes the design problem:

5.3 System-Level Architecture

The system-level architecture rules declare that television components can only connect to amplifiers and amplifiers can only connect to speakers. The following script of Python code establishes the first rule:

source code

```
InferenceRule.py.

# Rule: television connects to amplifier

class televisionToAmplifier(InferenceRule):
    def getqueries(self):
        tvToAmp=[ ('?x','category','television'),
                  ('?y','category','amplifier')]

        return [tvToAmp]

    def _maketriples(self,x,y):
        return [ ( x, 'connects to', y ) ]
```

First, the *getqueries* function searches for two RDF patterns, first with predicate category and object television, second with predicate category and object amplifier. Then, the *maketriples* function creates a new RDF, with subject television, predicate connects to, and object amplifier. The second connectivity rule is implemented in exactly the same way, except we test for,

```
ampToSpeaker=[('?x','category','amplifier'), ('?y','category','speaker') ]
```

Solution Procedure: The fragment of Python code:

source code

```
tvamp  = televisionToAmplifier()
ampspk = amplifierToSpeaker()
componentRDF.applyinference(tvamp)
componentRDF.applyinference(ampspk)
```

applies the televisionToAmplifier() and amplifierToSpeaker() inference rules to the design components graph of design component properties.

Results: The following script of Python code queries the design components graph, in search of new relationships:

source code

```
# Query the design components graph for system-level architecture relationships ...

print "******************************************************************"
print " Find which components 'z' connect to the Lg Television "
print "******************************************************************"
print ""

print componentRdf.query([('Lg Television','connects to','?z')])

print ""
print "******************************************************************"
print " Find which components 'z' connect to the Bose Amplifier  "
print "******************************************************************"
print ""

print componentRdf.query([('Bose Amplifier','connects to','?z')])

... Output of query ....

******************************************************************
 Find which components 'z' connect to the Lg Television
******************************************************************

[{'z': u'Klipsch Amplifier'}, {'z': u'Bose Amplifier'}, {'z': u'Polk Amplifier'}]
```

```
*******************************************************************
 Find which components 'z' connect to the Bose Amplifier
*******************************************************************

[{'z': u'Klipsch Speaker'}, {'z': u'Bose Speaker'}, {'z': u'Polk Speaker'}]
```

After applying the system-level architecture rules to the design components graph, the graph now has 57 unique vertices and 133 edges. The size of the graph in terms of the number of vertices remained the same. However, the order of the graph in terms of the number of edges has increase by a factor of 18 new relationships.

5.4 Synthesis of Feasible System Configurations

The component compatibility rules establish whether or not two different component types are compatible based upon the outputs of the first matched against the inputs of the second. For the synthesis of home theater design solutions, amplifier-to-television and amplifier-to-speaker compatibility are required.

Amplifier-to-Television Compatibility Rule: This rule establishes the conditions for compatibility of inputs and outputs between the television and amplifier components.

source code

```
InferenceRule.py.

# For Television & Amplifier to be compatible,
# Amplifier inputs are compatible with Television outputs

class Television_Amplifier_Compatibility(InferenceRule):
    def getqueries(self):
        Tv_Amp_Compatibility =[ ('?x','output','?z1'),
                              ('?x','category','television'),
                              ('?y','input','?z2'),
                              ('?y','category','amplifier')]

        return [Tv_Amp_Compatibility]

    def _maketriples(self,x,y,z1,z2 ):
        if z1 == z2:
            return [(x, 'compatible with',y)]
        else:
            return[]
```

The *getqueries* function assembles a search based upon four RDF patterns shown in the script of Python code. Then, the *maketriples* function tests for compability of input and output relations in the television and amplifier components. When the test evaluates to true, a new RDF statement is created declaring that telvision `x` is compatible with amplifier `y`.

Amplifier-to-Speaker Compatibility Rule: The amplifier-to-speaker compatability rule declares that if all the outputs of an amplifier are consistent with all the inputs of the speaker design component, then the two components are compatible. In the fragment of Python code:

source code

```
InferenceRule.py.

# For Amplifier & Speaker to be compatible,
# Speaker inputs are compatible with Amplifier outputs

class Amplifier_Speaker_Compatibility(InferenceRule):
    def getqueries(self):
        Amp_Speaker_Compatibility=[('?x','connects to','?y'),
                        ('?x','category','amplifier'),
                        ('?x','output','?z1'),
                        ('?x','power handling','?m1'),
                        ('?y','category','speaker'),
                        ('?y','input','?z2'),
                       ('?y','power handling','?m2')]

        return [Amp_Speaker_Compatibility]

    def _maketriples(self,x,y,z1,z2, m1,m2):
        list = m2[:]
        if z1 == z2 and int(m1) in range(int (list)):
            return [(x, 'compatible with',y)]
        else:
            return[]
```

amplifier component `x` will be compatible with speaker component `y` when the outputs of the amplifier are equivalent to the inputs of the speaker, and the power handling value of amplifier is within the speaker power handling range.

Solution Procedure: The fragment of Python code:

source code

```
# Apply Component Compatibility Rules ...

tvampCompat      = Television_Amplifier_Compatibility()
ampspeakerCompat = Amplifier_Speaker_Compatibility()
componentRdf.applyinference(tvampCompat)
componentRdf.applyinference(ampspeakerCompat)
```

applies the Television_Amplifier_Compatibility() and Amplifier_Speaker_Compatibility() inference rules to the design components graph of design component properties.

Results: The following script of Python code queries the design components graph, in search of new relationships:

source code

```
# Query the design component graph for compatibility relationships ...

print "********************************************************************"
print " Find which components 'z' are compatible with the Lg Television "
print "********************************************************************"
print ""

print componentRdf.query([('Lg Television','compatible with','?z')])

print ""
print "********************************************************************"
print " Find which components 'z' are compatible with the Bose Amplifier  "
print "********************************************************************"
print ""

print componentRdf.query([('Bose Amplifier','compatible with','?z')])
print ""

... Output of query ....

********************************************************************
 Find which components 'z' are compatible with the Lg Television
********************************************************************

[{'z': u'Klipsch Amplifier'}, {'z': u'Bose Amplifier'}, {'z': u'Polk Amplifier'}]

********************************************************************
 Find which components 'z' are compatible with the Bose Amplifier
********************************************************************

[{'z': u'Bose Speaker'}, {'z': u'Polk Speaker'}]
```

After the compatibility rules have been applied, the design components graph has 57 unique vertices and 148 edges. Notice that these transformations leave the number of graph vertices unchanged. However, 15 new edges are added to the graph structure. Figure 5.3 illustrates the design component graph.

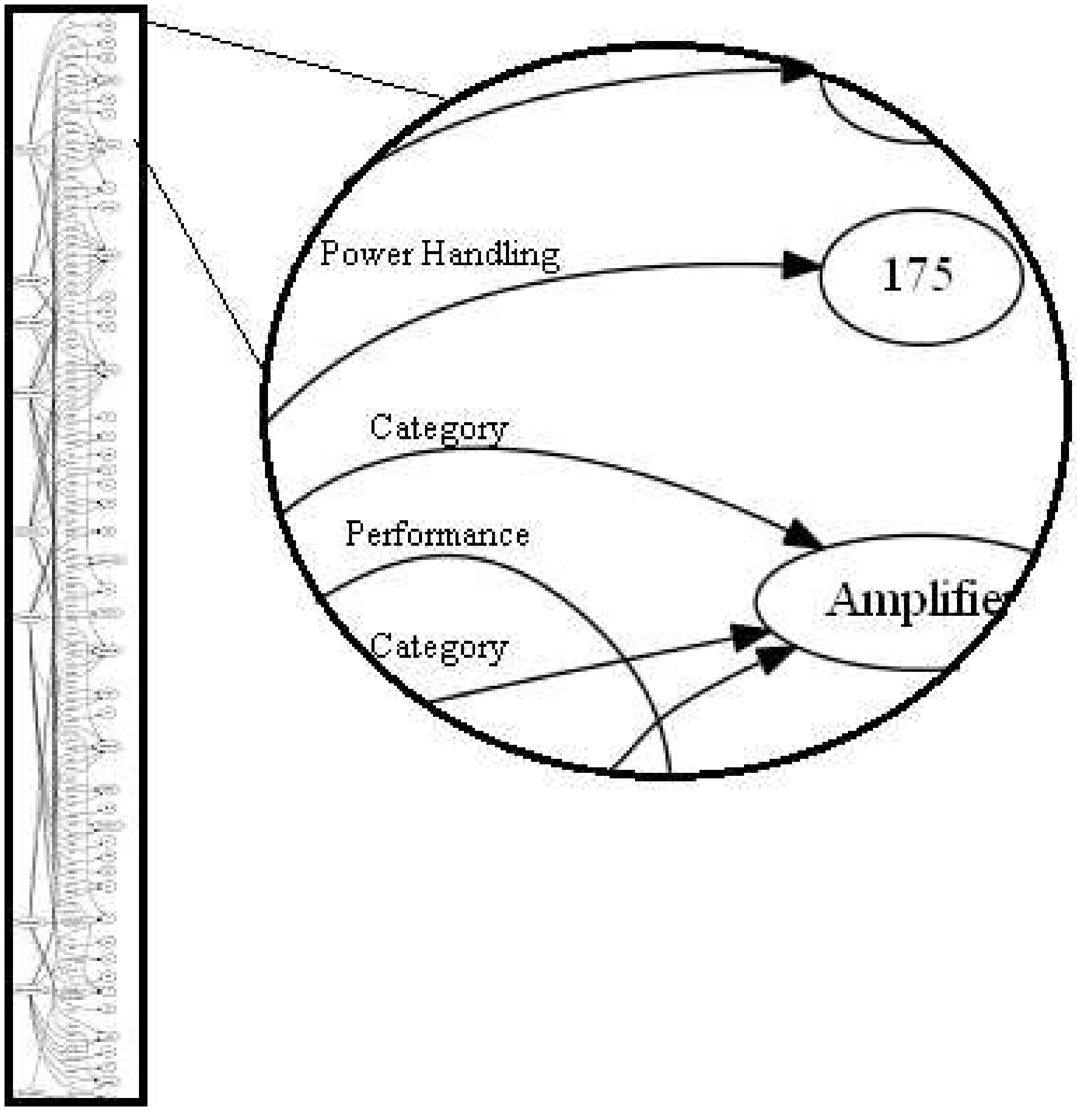


Figure 5.3: Modeling the design component RDF graph using PyDot.

Merging Graphs. The requirements model and design components graphs are merged. The Python code is as follows:

source code

```
# Merge Design Components and Requirements Model Graphs ...

mergegraph = SimpleGraph()
for sub, pred, obj in requirementRdf.triples((None, None, None)):
    mergegraph.add((sub, pred, obj))
for sub, pred, obj in componentRdf.triples((None, None, None)):
    mergegraph.add((sub, pred, obj))
```

Results: The merged graph has 99 unique vertices and 278 edges. The number of vertices for the merged graph is the summation of the unique requirements model and design components graph vertices. The number of edges for the merged graph is the summation of the requirements model and design components graph edges.

Synthesis of Feasible System Configurations. The feasible system configurations rule identifies permutations of compatible amplifier, television and speaker components, and for each compatible arrangement, computes and prints the total system cost. The Python code is as follows:

source code

```
InferenceRule.py.

class feasibleSystemConfigurations(InferenceRule):
    def getqueries(self):
        System=[ ('?x','connects to','?y'),
                 ('?x','cost','?x1'),
                 ('?y','compatible with','?z'),
                 ('?y','cost','?y1'),
                 ('?z','category','speaker'),
                 ('?z','cost','?z1') ]

        return [System]

    def _maketriples(self,x,y,z,x1,y1,z1):
        global counter
        count   = counter
        counter = counter+1

        combo = [[] for i in range(20)]
        combo[counter].append(x)
        combo[counter].append(y)
        combo[counter].append(z)

        total_cost = int(x1[0:])+int(y1[0:])+int(z1[0:])
        print 'System Design', counter, ': ', combo[counter]
        print 'System Design Cost = $', total_cost
        print ' '
        return []
```

It is important to notice that this rule merely identifies sets of compatible components and makes no reference to the requirements.

Solution Procedure. The fragment of Python code:

source code

```
# Apply Feasible System Configurations Rule...

feasibleSystemConfigurations = feasibleSystemConfigurations()
mergegraph.applyinference(feasibleSystemConfigurations)
```

The step-by-step procedure for synthesizing feasible system configurations and assessing their cost is as follows:

1. Assemble a search through the *getqueries* function based upon six RDF patterns.

a. Query the graph to find sets of two components that can be connected. This query is based on the connectivity Rules.

```
('?x','connects to',?y')
```

b. Query the graph to find the assigned cost of the first component of the sets of two components, returned from the connectivity query.

```
('?x','cost',?x1')
```

c. Query the graph to find components that are compatible with the second component of the sets of two components, returned from the connectivity query. In fact, this query is based on the Compatibility Rules.

```
('?y','compatible with',?z')
```

d. Query the graph to find the assigned cost of the second component of the sets of two components, returned from the connectivity query.

```
('?y','cost',?y1')
```

e. Query the graph to ensure that the third component, which is the compatible component of the second component returned from the connectivity query, is indeed a speaker component.

```
('?z','category','speaker')
```

f. Query the graph to find the assigned cost of the third component.

```
('?z','cost',?z1')
```

2. Create an array through the *maketriples* function and append each system combination of components as a result from the *getqueries* function to the array.

3. Calculate the cost of each system combination by summing the cost of each component.

Results: There exist eighteen compatible system configurations, each one containing a television, an amplifier, and speakers.

source code

```
:::: C O M P A T I B L E  S Y S T E M  D E S I G N S ::::

System Design 1 :  [u'Lg Television', u'Bose Amplifier', u'Polk Speaker']
System Design Cost = $ 2000

System Design 2 :  [u'Lg Television', u'Bose Amplifier', u'Bose Speaker']
System Design Cost = $ 1928

System Design 3 :  [u'Samsung Television', u'Bose Amplifier', u'Polk Speaker']
System Design Cost = $ 2350

System Design 4 :  [u'Samsung Television', u'Bose Amplifier', u'Bose Speaker']
System Design Cost = $ 2278

System Design 5 :  [u'Sony Television', u'Bose Amplifier', u'Polk Speaker']
System Design Cost = $ 1900

System Design 6 :  [u'Sony Television', u'Bose Amplifier', u'Bose Speaker']
System Design Cost = $ 1828

System Design 7 :  [u'Lg Television', u'Klipsch Amplifier', u'Klipsch Speaker']
System Design Cost = $ 1970

System Design 8 :  [u'Lg Television', u'Klipsch Amplifier', u'Polk Speaker']
System Design Cost = $ 2070

System Design 9 :  [u'Lg Television', u'Klipsch Amplifier', u'Bose Speaker']
System Design Cost = $ 1998

System Design 10 :  [u'Samsung Television', u'Klipsch Amplifier', u'Klipsch Speaker']
System Design Cost = $ 2320

System Design 11 :  [u'Samsung Television', u'Klipsch Amplifier', u'Polk Speaker']
System Design Cost = $ 2420

System Design 12 :  [u'Samsung Television', u'Klipsch Amplifier', u'Bose Speaker']
System Design Cost = $ 2348

System Design 13 :  [u'Sony Television', u'Klipsch Amplifier', u'Klipsch Speaker']
System Design Cost = $ 1870

System Design 14 :  [u'Sony Television', u'Klipsch Amplifier', u'Polk Speaker']
System Design Cost = $ 1970

System Design 15 :  [u'Sony Television', u'Klipsch Amplifier', u'Bose Speaker']
System Design Cost = $ 1898

System Design 16 :  [u'Lg Television', u'Polk Amplifier', u'Bose Speaker']
System Design Cost = $ 1978

System Design 17 :  [u'Samsung Television', u'Polk Amplifier', u'Bose Speaker']
System Design Cost = $ 2328

System Design 18 :  [u'Sony Television', u'Polk Amplifier', u'Bose Speaker']
System Design Cost = $ 1878
```

Compatible component pairs are compatible design components which do not necessarily satisfy all design requirements. The least expensive (potentially good) cofiguration costs USD \$1,828; it consists of a Sony television, Bose amplifier, and Bose speakers. The most expensive configuration costs USD \$2,420, and consists of a Samsung television, a Klipsch amplifier, and a Polk speaker.

5.5 Quantitative Evaluation of Requirements

The quantitative evaluation of requirements involves development of inference rules for evaluation of the level 3, level 2 and level 1 requirements, followed by their application to the merged RDF graph of requirements and design components.

Level 3 Requirements. Inference rules for the Level 3 component-level requirements are derived directly from the requirement description. They evaluate one or more inequality constraints and create new attribute relationships when the constraints evaluate to true. Consider, for example, the inference rule for quantitative evaluation of Requirement 15.

source code

```
InferenceRule.py.

class InferenceRule:
    def getqueries(self):
        return None

    def maketriples(self,binding):
        return self._maketriples(**binding)

# Rule: Requirement 15
class Req15(InferenceRule):
    def getqueries(self):
        setReq15 =[('?x','id','Requirement 15'),
                   ('?x','category','requirement'),
                   ('?y','power handling','?z'),
                   ('?y','category','speaker')]

        return [setReq15]

    def _maketriples(self,x,y,z):
        list = z[:]
        if int (149) in range(int (list)):
            return [(y,'satisfy', x)] + [(x,'status', 'satisfied')]
        else:
            return []
```

The *getqueries* function searches the merged requirements graph for RDF triples having a requirements identification of Requirement 15, and a speaker with power handling parameters. When 150 watts is within the power handling range of a speaker, two new RDF statements are created: (1) Amplifier component “y” satisfies requirement 15, and (2) Requirements 15 status is “satisfied.”

Dedicated requirements evaluation functions have also been developed for Requirements 8 through 14. Functions for Requirements 12, 13 and 14 evaluate constraints associated with cost of the amplifier, speaker and television components. Similarly, functions for Requirements 8 through 11 evaluate constraints relating to the weight and acceptable dimensions of the flatscreen television.

Level 2 Requirements. Next, we develop inference rules for the Level 2 requirements that serve the purpose of a detailed agreement between the customer and supplier. Satisfaction of the Level 2 inference rules occurs through the satisfaction of lower-level (Level 3) dependency requirements. For example, the fragments of Python code:

source code

```
InferenceRule.py.

class InferenceRule:
    def getqueries(self):
        return None

    def maketriples(self,binding):
        return self._maketriples(**binding)

# Rule: Requirement 7

class Req6(InferenceRule):
    def getqueries(self):
        setReq7 = [ ('?x','id','Requirement 7'),
                    ('?x','depends on','?y'),
                    ('?y','category','requirement'),
                    ('?y','status','?z')]

        return [setReq7]

    def _maketriples(self,x,y,z):
        if z == 'satisfied':
            return[(x,'status', 'satisfied')]
        else:
            return[(x,'status', 'unsatisfied')]
```

determines whether or not Requirement 7 is satisfied. Level 2 requirements will be satisfied if and only if all of the lower-level requirements are satisfied. Identical functions have been written for the evaluation of Requirements 3 through 6.

Level 1 Requirements. Last, we develop inference rules for the evaluation of Requirements 1 and 2, the initial requirements. Satisfaction of the Level 1 requirements occurs indirectly through the satisfaction of the lower level (Level 2) requirements. The strategy for evaluating a requirements is identical to the Level 2 requirements, e.g., `setReq2 = [ ('?x','id','Requirement 2') , ('?x','depends on','?y') , ('?y','category','requirement') , ('?y','status','?z') ]`.

Solution Procedure: The fragment of Python code:

source code

```
# Apply Level 3 Requirements Rules ...

requirement15 = Req15()
mergegraph.applyinference(requirement15)
requirement14 = Req14()
mergegraph.applyinference(requirement14)
requirement13 = Req13()
mergegraph.applyinference(requirement13)
requirement12 = Req12()
mergegraph.applyinference(requirement12)
requirement11 = Req11()
mergegraph.applyinference(requirement11)
requirement10 = Req10()
mergegraph.applyinference(requirement10)
requirement9 = Req9()
mergegraph.applyinference(requirement9)
requirement8 = Req8()
mergegraph.applyinference(requirement8)

# Apply Level 2 Requirements Rules ...

requirement7 = Req7()
mergegraph.applyinference(requirement7)
requirement6 = Req6()
mergegraph.applyinference(requirement6)
requirement5 = Req5()
mergegraph.applyinference(requirement5)
requirement4 = Req4()
```

```
mergegraph.applyinference(requirement4)
requirement3 = Req3()
mergegraph.applyinference(requirement3)

# Apply Level 1 Requirements Rules ...

requirement2 = Req2()
mergegraph.applyinference(requirement2)
requirement1 = Req1()
mergegraph.applyinference(requirement1)
```

applies all the level 3, level 2, and level 1 requirement inference rules to the merged graph of requirements and design component properties.

Results: After the inference rules for the level 3 requirements have been applied, the merged graph has 100 unique vertices and 308 edges While only one new satisfied vertex is added to the graph, 30 new edge relationships are added. After applying the level 2 requirement inference rules to the merged graph, the graph has 100 unique vertices and 319 edges. The graph size remains unchanged and 11 new edge relationships are added. Finally, application of the level 1 inference rules transforms the merged graph to 100 unique vertices and 321 edges (i.e., only two new edges are added).

5.6 Synthesis of System-Level Design Alternatives

The system design rule establishes relationships between the design components and the design requirements. The rule declares that a home theater system design can only be comprised of compatible design components which satisfy all design requirements, and is implemented by the script of Python code:

source code

```
InferenceRule.py.

class systemDesign(InferenceRule):
    def getqueries(self):

        sysDesReq = [('?x','connects to','?y'),
                     ('?x','category','television'),
                     ('?x','cost','?x1'),
                     ('?x','performance','?xp'),
                     ('?x','reliability','?xr'),
                     ('?x','satisfy','Requirement 12'),
                     ('?x','satisfy','Requirement 11'),
                     ('?x','satisfy','Requirement 10'),
                     ('?x','satisfy','Requirement 9'),
                     ('?x','satisfy','Requirement 8'),

                     ('?y','compatible with','?z'),
                     ('?y','category','amplifier'),
                     ('?y','cost','?y1'),
                     ('?y','performance','?yp'),
                     ('?y','reliability','?yr'),
                     ('?y','satisfy','Requirement 13'),
                     ('?y','satisfy','Requirement 7'),

                     ('?z','category','speaker'),
                     ('?z','cost','?z1'),
                     ('?z','performance','?zp'),
                     ('?z','reliability','?zr'),
                     ('?z','satisfy','Requirement 15'),
                     ('?z','satisfy','Requirement 14'),
                     ('?z','satisfy','Requirement 7')]

        return [sysDesReq]

    def _maketriples(self,x,y,z,x1,y1,z1,xp,yp,zp,xr,yr,zr):

        global sysCounter
        count = sysCounter
        sysCounter= sysCounter+1
        comb = []
        comb.append(x)
        comb.append(y)
        comb.append(z)

        combo = [[] for i in range(20)]
        combo[sysCounter].append(x)
        combo[sysCounter].append(y)
        combo[sysCounter].append(z)

        total_performance = int(xp)+int(yp)+int(zp)
        comb.append(total_performance)

        total_reliability = float(xr)*float(yr)*float(zr)
        comb.append(total_reliability)

        total_cost = int(x1[0:])+int(y1[0:])+int(z1[0:])
        if total_cost <= int(2000):
            comb.append(total_cost)
            all_comb.append(comb)
            print 'System Design', sysCounter, ': ', combo[sysCounter]
            print 'System Design Cost = $', total_cost
            print 'System Design Performance = ', total_performance
            print 'System Design Reliability = ', total_reliability
            print ' '
            return []
        sysCounter = sysCounter -1
        return []
```

The *maketriples* function assembles and prints sets of compatible television, amplifier, and speaker components that also satisfy the design requirements, along with the measures of effectiveness: cost, performance, and reliability. The total cost computation is easy – we sum the cost of the participating design components. System performance is computed by summing the performance level of each design component. To compute the system-level reliability we assume that the components are connected in a series – hence the system-level reliability is simply the product of the component reliabilities.

Solution Procedure: The following (abbreviated) script of Python code illustrates the process of applying inference rules for the merged graph of home theater requirements:

source code

```
# Apply System Design Rule...

systemDesign = systemDesign()
mergegraph.applyinference(systemDesign)
```

In this procedure, each design requirement inference-rule function is assigned to an object, and then the design requirement inference-rule function is applied to the merged requirements graph.

Results: System-level designs are compatible design components which satisfy all design requirements. As illustrated in the script of program output below, there exist nine system-level designs.

source code

```
:::: S Y S T E M - L E V E L  D E S I G N S ::::

System Design 1 :  [u'Lg Television', u'Bose Amplifier', u'Polk Speaker']
System Design Cost = $ 2000
System Design Performance =  23
System Design Reliability =  0.448

System Design 2 :  [u'Lg Television', u'Bose Amplifier', u'Bose Speaker']
System Design Cost = $ 1928
System Design Performance =  25
System Design Reliability =  0.504

System Design 3 :  [u'Sony Television', u'Bose Amplifier', u'Polk Speaker']
System Design Cost = $ 1900
System Design Performance =  28
System Design Reliability =  0.576

System Design 4 :  [u'Sony Television', u'Bose Amplifier', u'Bose Speaker']
System Design Cost = $ 1828
System Design Performance =  30
System Design Reliability =  0.648

System Design 5 :  [u'Lg Television', u'Klipsch Amplifier', u'Bose Speaker']
System Design Cost = $ 1998
System Design Performance =  20
System Design Reliability =  0.441
System Design 6 :  [u'Sony Television', u'Klipsch Amplifier', u'Polk Speaker']
System Design Cost = $ 1970
System Design Performance =  23
System Design Reliability =  0.504

System Design 7 :  [u'Sony Television', u'Klipsch Amplifier', u'Bose Speaker']
System Design Cost = $ 1898
System Design Performance =  25
System Design Reliability =  0.567

System Design 8 :  [u'Lg Television', u'Polk Amplifier', u'Bose Speaker']
System Design Cost = $ 1978
System Design Performance =  23
System Design Reliability =  0.567

System Design 9 :  [u'Sony Television', u'Polk Amplifier', u'Bose Speaker']
System Design Cost = $ 1878
System Design Performance =  28
System Design Reliability =  0.729
```

The highest performing system-level design has a value of 30 with an associated cost of USD $1,828 and a reliability of 0.648. This system-level design includes a Sony television, Bose amplifier, and a Bose speaker. The most reliable system-level design has a value of 0.729 with an associated cost of USD $1,878 and a performance of 28. This system-level design includes a Sony television, Polk amplifier, and a Bose speaker. The least expensive system-level design cost USD $1,828 with an associated performance of 30 and a reliability of 0.648. This system-level design includes a Sony television, Bose amplifier, and a Bose speaker. A final observation is that the system-level design which has the highest performance level is also the least expensive. The tentative conclusion is as follows: the best system-level design will include a Sony television, Bose amplifier, and a Bose speaker.

5.7 Trade-Space Evaluation and Exploration

The results of applying connectivity and compatibility inference rules, as discussed in section 5.3, have revealed the system-level design alternatives of the home theater system. We aim to explore the best system-level design through a trade-space evaluation.

The trade-space evaluation will be based on three criteria: cost, performance, and reliability. First, we evaluate each system-level design against cost verse performance. Then, we find the Pareto optimal point of cost verse performance for minimizing cost and maximizing performance. Next, we evaluate each system-level design against cost verse reliability. As well, we find the Pareto optimal point of cost verse reliability for for minimizing cost and maximizing performance. Last, we evaluate each system-level design against performance verse reliability. As well, we find the Pareto optimal point of performance verse reliability for maximizing performance and maximizing reliability.

The acquired Pareto optimal points of each trade-space evaluation facilitate in helping us select the best system-level design according to our desirable criteria. When evaluating each system-level design against cost verse performance, we prefer to minimize cost and maximize performance. For that reason, the system-level design which include Sony television, Bose amplifier, and Bose speaker is the best home theater system. When evaluating each system-level design against cost verse reliability, we prefer to minimize cost and maximize reliability. For that reason,

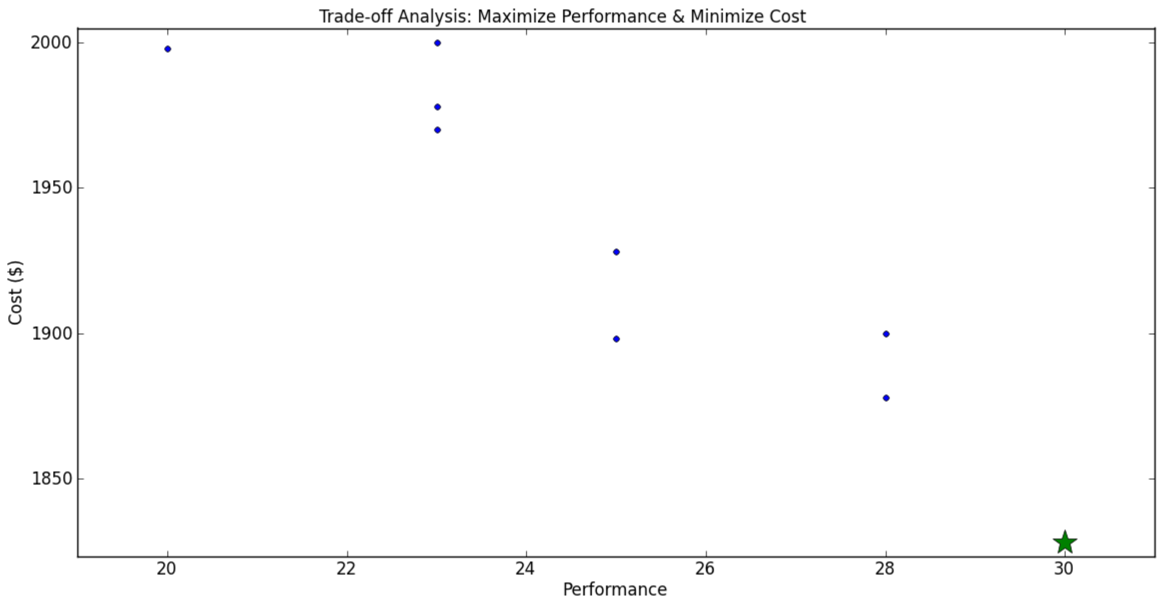


Figure 5.4: Trade-off analysis GUI of minimizing cost and maximizing performance with a starred Pareto optimal point.

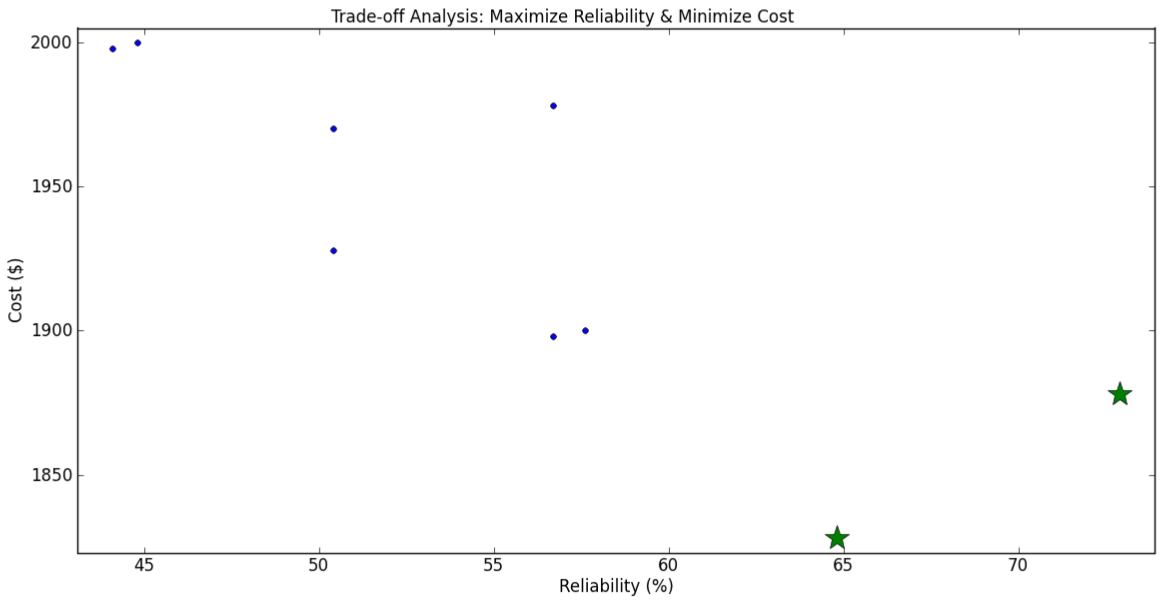


Figure 5.5: Trade-off analysis GUI of minimizing cost and maximizing reliability with a starred Pareto optimal point.

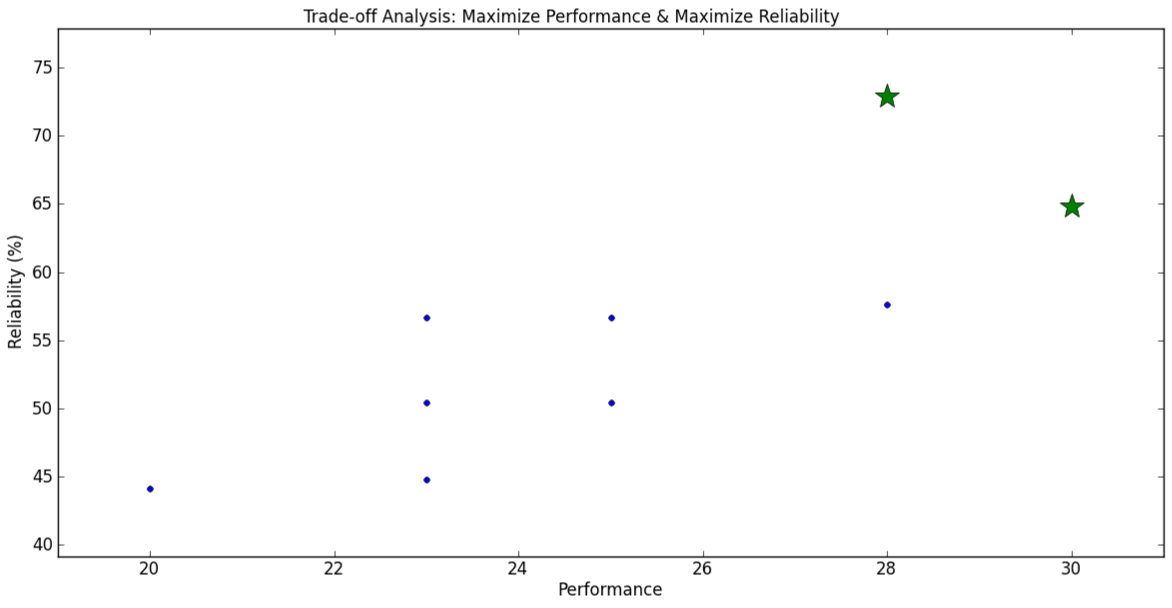


Figure 5.6: Trade-off analysis GUI of maximizing reliability and maximizing performance with a starred Pareto optimal point.

the system-level designs which include Sony television, Bose amplifier, and Bose speaker and Sony television, Polk amplifier, and Bose speaker are the best home theater systems. When evaluating each system-level design against performance verse reliability, we prefer to maximize performance and maximize reliability. For that reason, the system-level designs which include Sony television, Bose amplifier, and Bose speaker and Sony television, Polk amplifier, and Bose speaker are the best home theater system. As a result, Sony television, Bose amplifier, and Bose speaker is the best overall home theater design.

Chapter 6

Conclusions and Future Work

6.1 Summary and Conclusions

In this thesis, we have discovered that RDF and Python can be used in a software pipeline as a replacement for RDF, OWL, Jess, Jena, Protg, and SWRL. We successfully developed a three-level RDF graph representations for requirements and their properties, and used Python for the implementation of logical reasoning and inferencing mechanisms to solve a component-selection design problem. We focused on merging RDF graphs to develop relationships between design requirements and design components, and the design of sequences of inference rules to systematically transform and filter the RDF graph into representations for ensembles of design alternatives. We then used the algorithm for relabeling of Pareto-Optimal design solutions to separate the inferior and non-inferior designs. Tasks that can be accomplished through the use of this pipeline include:

1. Development of system requirement and design component representations (e.g., television, speaker, amplifier) in Java.

2. Transformation of system requirement and design component representations into RDF graphs modeled in Python, and

3. Development and implementation of inference rules modeled in Python for the step-by-step selection of compatible design component combinations that satisfy all system requirements.

4. Computation Pareto-Optimal designs to identify non-inferior design combination.

5. Tracking of the size of the RDF graphs at various stages in the software pipeline.

Steps 2, 3 and 4 make up a software pipeline for the synthesis of Pareto-Optimial design solutions from requirements and component options. The software pipeline was linked to visualization of RDF graphs through PyDot.

The computational results of this thesis suggest that for design problems of a modest size, RDF and Python can be used to satisfy system requirements and acquire good design solutions in a straightforward and uncomplicated manner. The size of the RDF graphs, and associated inference rules in Python, are at least an order of magnitude smaller than their counterpart implementation in OWL. At this point the proposed approach may not be scalable to thousands of requirements and libraries containing hundreds of component options, practical design solutions can be obtained with smaller numbers of requirements and component options. We expect, however, that this shortcoming can be mitigated with the development of appropriate algorithms and strategies for using the results of inference computations to carry forward only that information that will be used downstream.

6.2 Future Work

Future work should focus on strategies to overcome the limitations of the proposed method. Looking ahead, computational support is needed for:

- Reasoning with physical quantities (i.e., numerical quantities plus dimensions). In the current pipeline, all quantities are represented as character strings. An improved pipeline would explicitly represent and provide support for reasoning about physical quantities. This is a reasonable expectation given that units and measures are now part of Java 7. For details, see the Java package javax.measure.

- Developing a requirements template that allows for the automatic generation of inference rule functions from the RDF graph. In the current work, each inferencing mechanism for the level-3 system requirements is manually embedded into the software pipeline. All of the system-level requirements are manually applied to the RDF graph. A family of requirement templates would define the syntactic structure of requirements and semi-automation of the inference functions needed for design.

- Developing a way to automate the generation of visualizing RDF graphs in PyDot. The current visualization of RDF graphs using PyDot is implemented through character strings. A method for extracting triples from the RDF graph would assemble each subject, predicate, and object into a format for visualizing RDF graphs.

- Combining the feasible system configurations rule and the system design rule. In the current pipeline, the feasible system configurations rule identifies sets of compatible components and makes no reference to the requirements while the system design rule identifies compatible design components which satisfy all design requirements. Combining these rules would minimize the frequency of querying the RDF graph and further compact the software pipeline.

Bibliography

[1] Best Buy - Computers, Video Games, TVs, Cameras, Appliances, Phones. See http://www.bestbuy.com/.

[2] Polk Audio. See http://www.polkaudio.com/.

[3] Austin M.A., Mayank V., and Shmunis N. Ontology-Based Validation of Connectivity Relationships in a Home Theater System. *International Journal of Intelligent Systems*, 21(10):1111–1125, October 2006.

[4] Austin M.A., Mayank V., and Shmunis N. PaladinRM: Graph-Based Visualization of Requirements Organized for Team-Based Design. *Systems Engineering: The Journal of the International Council on Systems Engineering*, 9(2):129–145, May 2006.

[5] Ball M., Baras J., Bashyam S., Karne R. and Trichur, V.,. On the Selection of Parts and Processes during Design of Printed Circuit Board Assemblies. *1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Automation*, 3:241–248, 1995.

[6] Bartholet R.G, Brogan D.C., and Reynolds Jr. P.F. The Computational Complexity of Component Selection in Simulation Reuse. In *Winter Simulation Conference*, pages 2472–2481, Orlando, Florida, 2005.

[7] Berners-Lee T., Hendler J., Lassa O. The Semantic Web. *Scientific American*, pages 35–43, May 2001.

[8] Butler R. NASA LaRC Formal Methods Program: What Is Formal Methods? 2001. See http://shemesh.larc.nasa.gov/fm/fm-what.html.

[9] Carrera E. Python Package Index. See http://pypi.python.org/pypi/pydot.

[10] Ciocoiu M., Gruninger M., Nau D.S. Ontologies for Integrating Engineering Applications. *Journal of Computing and Information Science in Engineering*, 1(1):12–22, 2001.

[11] Ferland J., Hertz A., and Lavoie A. An Object-Oriented Methodology for Solving Assignment-Type Problems with Neighborhood Search Techniques. *Operations Research*, 44(2):347–359, March-April 1996.

[12] Friedman G. *Constraint Theory: Multidimensional Mathematical Model Management*, volume 23. IFSR International Series on Systems Science and Engineering, Springer, 2005.

[13] Geroimenko V., and Chen C. (Eds). *Visualizing the Semantic Web: XML-based Internet and Information Visualization*. Springer, 2003.

[14] Hamza-Lup G.L., Agarwal A., Shankar R., and Iskander C. Component Selection Strategies based on System Requirements' Dependencies on Component Attibutes. In *SysCon 2008 – IEEE International Systems Conference*, Montreal, Canada, March 7-10 2008.

[15] Hendler J. Agents and the Semantic Web. *IEEE Intelligent Systems*, pages 30–37, March/April 2001.

[16] Horrocks I., Patel-Schneider P.F., Boley H., Tabet S., Grosof B., and Dean M. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. 2004. W3C Member Submission: See http://www.w3.org/Submission/SWRL/.

[17] ILOG Solver. See http://www.ilog.com/products/cp/ (Accessed February 26, 2009). 2009.

[18] Jess – The Expert System Shell for the Java Platform. See http://herzberg.ca.sandia.gov/jess/. 2003.

[19] Kiliccote H., Garrett J.H. Standards Usage Language (SUL). *Journal of Computing in Civil Engineering*, 15(2):118–128, 2001.

[20] Kositsyna N., Mayank V., and Austin M. Paladin Software Toolset. *Institute for Systems Research*, 2003. For more information, see http://www.isr.umd.edu/paladin/.

[21] Mayank V., Austin M.A. Ontology-Enabled Validation of System Architectures. In *Proceedings of Fourteenth Annual International Symposium of The International Council on Systems Engineering (INCOSE)*, Toulouse, France, June 20-24 2004.

[22] Mayank V., Kositsyna N. and Austin M.A. Requirements Engineering and the Semantic Web: Part II. Representation, Management and Validation of Requirements and System-Level Architectures. *ISR Technical Report 2004-14*, 2004. See http://techreports.isr.umd.edu/reports/2004/TR_2004-14.pdf.

[23] Mayank V., Kositsyna N., and Austin M.A. Graph-Based Visualization of System Requirements Organized for Team-Based Development. Technical Research Report TR 2005-92, Institute for Systems Research, College Park, MD 20742, June 2005.

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

[25] Protege Ontology Editor and Knowledge Acquisition System. For details, see http://protege.stanford.edu. 2003.

[26] Segaran T., Taylor J., Evans C. *Programming the Semantic Web*. O'Reilly, Beijing, 2009.

[27] Tidwell D. *XSLT*. O'Reilly and Associates, Sebastopol, California, 2001.

[28] Web Ontology Language (OWL). See http://www.w3.org/TR/owl-ref/. 2003.

[29] XML Stylesheet Transformation Language (XSLT). See http://www.w3.org/Style/XSL. 2002.

Appendix A
Python GUI and Trade-Space Visualization Code

In this appendix we provide the process of creating a Python GUI and trade-space visualization for each of the feasible system designs that are generated.

source code

```
GUI and Trade-Space Visualization.py.

from Engineering_Design_Components import Driver
from simplegraph_chapter3 import SimpleGraph
from Home_Theater_System_InferenceRule import *
import matplotlib.pyplot as plt
import numpy as np
from pylab import *

# *******R D F  G R A P H S*******

# Build requirement graph ...

requirementRdf = SimpleGraph()
requirementRdf.load('C:\Users\Queen\workspace\HomeTheatreSystem\
                    graphfileRequirement.csv')

# Compute the size of the requirements model graph ...

size  = []
aList = []

for sub, pred, obj in requirementRdf.triples((None, None, None)):
    aList.append(sub)
    aList.append(obj)

print "Nodes in the component graph..."
print "List: " ,aList
print "List size: ", len(aList)
print " "
aList = list(set(aList))
print "Eliminate redundant nodes ..."
print "New List: " ,aList
print "New List size: ", len(aList)
print " "

# Initialization of the list of vertices and edges for the size-graph ...

verticesList = list()
edgesList    = list()

# Creation of the vertices for the size-graph ...
```

```
for k in range (len(aList)):
    verticesList.append(aList[k])

# Creation of edges for the size-graph ...

edgesList = requirementRdf.makeEdgesList(verticesList)

# Creation of the size-graph ...

directed_graph = Graph(verticesList,edgesList)

# Compute the graph size ...

print("The vertices within the directed graph: " + str(directed_graph.getVertices()))
print("The number of vertices within the directed graph: " + str(directed_graph.getSize()))
print " "
print("The edges within the directed graph are: ")
directed_graph.printEdges()
print("The number of edges within the directed graph: " + str(directed_graph.getEdgeSize()))
print ""

# Build component graph ...

componentRdf = SimpleGraph()
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileTelevision.csv')
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileAmplifier.csv')
componentRdf.load('C:\Users\Queen\workspace\Thesis Code\graphfileSpeaker.csv')

# Apply System-Level Architecture Rules ...

tvamp  = televisionToAmplifier()
ampspk = amplifierToSpeaker()
componentRdf.applyinference(tvamp)
componentRdf.applyinference(ampspk)

# Compute the size of the design components graph ...

size  = []
aList = []

for sub, pred, obj in componentRdf.triples((None, None, None)):
    aList.append(sub)
    aList.append(obj)

... Details of graph size computation removed ...

# Query the design components graph for system-level architecture relationships ...

print "*******************************************************************"
print " Find which components 'z' connect to the Lg Television "
print "*******************************************************************"
print ""
print componentRdf.query([('Lg Television','connects to','?z')])

print ""
print "****************************************************************"
print " Find which components 'z' connect to the Bose Amplifier  "
print "****************************************************************"
print ""

print componentRdf.query([('Bose Amplifier','connects to','?z')])
print ""

# Apply Component Compatibility Rules ...

tvampCompat      = Television_Amplifier_Compatibility()
ampspeakerCompat = Amplifier_Speaker_Compatibility()
componentRdf.applyinference(tvampCompat)
componentRdf.applyinference(ampspeakerCompat)

# Compute the size of the design components graph ...

size  = []
aList = []

for sub, pred, obj in componentRdf.triples((None, None, None)):
    aList.append(sub)
    aList.append(obj)

... Details of graph size computation removed ...

# Query the design components graph for compatibility relationships ...

print "****************************************************************"
print " Find which components 'z' are compatible with the Lg Television "
print "****************************************************************"
print ""

print componentRdf.query([('Lg Television','compatible with','?z')])

print ""
print "****************************************************************"
print " Find which components 'z' are compatible with the Bose Amplifier  "
print "****************************************************************"
print ""

print componentRdf.query([('Bose Amplifier','compatible with','?z')])
print ""

# Merge Design Components and Requirements Model Graphs ...

mergegraph = SimpleGraph()
for sub, pred, obj in requirementRdf.triples((None, None, None)):
    mergegraph.add((sub, pred, obj))
for sub, pred, obj in componentRdf.triples((None, None, None)):
    mergegraph.add((sub, pred, obj))
```

```
# Compute the size of the design components graph ...

size  = []
aList = []

for sub, pred, obj in mergegraph.triples((None, None, None)):
    aList.append(sub)
    aList.append(obj)

... Details of graph size computation removed ...

# Apply Feasible System Configurations Rule...

feasibleSystemConfigurations = feasibleSystemConfigurations()
mergegraph.applyinference(feasibleSystemConfigurations)

# Apply Level 3 Requirements Rules ...

requirement15 = Req15()
mergegraph.applyinference(requirement15)
requirement14 = Req14()
mergegraph.applyinference(requirement14)
requirement13 = Req13()
mergegraph.applyinference(requirement13)
requirement12 = Req12()
mergegraph.applyinference(requirement12)
requirement11 = Req11()
mergegraph.applyinference(requirement11)
requirement10 = Req10()
mergegraph.applyinference(requirement10)
requirement9  = Req9()
mergegraph.applyinference(requirement9)
requirement8  = Req8()
mergegraph.applyinference(requirement8)

# Compute the size of the merge graph ...

size  = []
aList = []

for sub, pred, obj in mergegraph.triples((None, None, None)):
    aList.append(sub)
    aList.append(obj)

... Details of graph size computation removed ...

# Apply Level 2 Requirements Rules ...

requirement7 = Req7()
mergegraph.applyinference(requirement7)
requirement6 = Req6()
mergegraph.applyinference(requirement6)
requirement5 = Req5()
```

```
mergegraph.applyinference(requirement5)
requirement4 = Req4()
mergegraph.applyinference(requirement4)
requirement3 = Req3()
mergegraph.applyinference(requirement3)

# Compute the size of the merge graph ...

size  = []
aList = []

for sub, pred, obj in mergegraph.triples((None, None, None)):
    aList.append(sub)
    aList.append(obj)

... Details of graph size computation removed ...

# Apply Level 1 Requirements Rules ...

requirement2 = Req2()
mergegraph.applyinference(requirement2)
requirement1 = Req1()
mergegraph.applyinference(requirement1)

# Compute the size of the merge graph ...

... Details of graph size computation removed ...

# Apply System Design Rule...

systemDesign = systemDesign()
mergegraph.applyinference(systemDesign)

## ******* P A R E T O  O P T I M A L  A L G O R I T H M *******

# Get all feasible combinations from inference class
full = return_comb()

performance = []
reliability = []
cost = []

for i in range(len(full)):
    performance.append(full[i][3])
    reliability.append(100*(full[i][4]))
    cost.append(full[i][5])

print "Finding the Pareto-Optimal point for Cost vs Performance ... "

x_objective = performance
y_objective = cost

... Details of Pareto-Optimal Computation removed ...
```

```
# ******* P L O T   T R A D E - O F F   A N A L Y S I S *******
#                          AND
#       ******* P A R E T O   O P T I M A L *******

print ''
print "========================="
print "Pareto Optimal Points"
print "========================="
print ''

print "Pareto Point(s):",temp

if input1 == 0 and input2 == 0:
    # Min x-axis & Min y-axis
    fig, (ax0) = plt.subplots(ncols=1)
    x = x_objective
    y = y_objective
    a,b = zip(*temp)

    ax0.plot(x, y, marker ='.',markersize= 8, linewidth=0)
    ax0.plot(a, b, marker ='*',markersize= 18, linewidth=0)
    ax0.axis([min(x)-1, max(x)+1, min(y)-5, max(y)+5])
    ax0.set_xlabel('Performance')
    ax0.set_ylabel('Cost ($)')

    fig.suptitle('Trade-off Analysis: Minimize Performance & Minimize Cost')
    plt.tight_layout(pad=2)
    plt.show()

elif input1 == 0 and input2 == 1:
    # Min x-axis & Max y-axis
    fig, (ax1) = plt.subplots(ncols=1)
    x = x_objective
    y = y_objective
    a,b = zip(*temp)

    ax1.plot(x, y, marker ='.',markersize= 8, linewidth=0)
    ax1.plot(a, b, marker ='*',markersize= 18, linewidth=0)
    ax1.axis([min(x)-1, max(x)+1, min(y)-5, max(y)+5])
    ax1.set_xlabel('Performance')
    ax1.set_ylabel('Cost ($)')

    fig.suptitle('Trade-off Analysis: Minimize Performance & Maximize Cost')
    plt.tight_layout(pad=2)
    plt.show()

elif input1 == 1 and input2 == 1:
    # Max x-axis & Max y-axis
    fig, (ax2) = plt.subplots(ncols=1)
    x = x_objective
    y = y_objective
    a,b = zip(*temp)

    ax2.plot(x, y, marker ='.',markersize= 8, linewidth=0)
    ax2.plot(a, b, marker ='*',markersize= 18, linewidth=0)
    ax2.axis([min(x)-1, max(x)+1, min(y)-5, max(y)+5])
    ax2.set_xlabel('Performance')
    ax2.set_ylabel('Cost ($)')

    fig.suptitle('Trade-off Analysis: Maximize Performance & Maximize Cost')
    plt.tight_layout(pad=2)
    plt.show()

elif input1 == 1 and input2 == 0:
    # Max x-axis & Min y-axis
    fig, (ax3) = plt.subplots(ncols=1)
    x = x_objective
    y = y_objective
    a,b = zip(*temp)

    ax3.plot(x, y, marker ='.',markersize= 8, linewidth=0)
    ax3.plot(a, b, marker ='*',markersize= 18, linewidth=0)
    ax3.axis([min(x)-1, max(x)+1, min(y)-5, max(y)+5])
    ax3.set_xlabel('Performance')
    ax3.set_ylabel('Cost ($)')

    fig.suptitle('Trade-off Analysis: Maximize Performance & Minimize Cost')

    plt.tight_layout(pad=2)
    plt.show()

x_objective = reliability
y_objective = cost

print "Finding the Pareto-Optimal point for Cost vs Reliability ... "

... Details of Pareto-Optimal computation removed ...

# ******* P L O T   T R A D E - O F F   A N A L Y S I S *******
#                           AND
#       ******* P A R E T O   O P T I M A L *******

... details of reliability versus cost plot removed ....

x_objective = performance
y_objective = reliability

print "Finding the Pareto-Optimal point for Reliability vs Performance ... "

# Prompt User

# Ask user for the intent of objective 1 and store
input1 = int(raw_input("Is the performance objective min[0] or max[1] : "))

# Ask user for the intent of objective 2 and store
input2 = int(raw_input("Is the reliability objective min[0] or max[1] : "))

... Details of Pareto-Optimal computation removed ...
```

```
# ******* P L O T  T R A D E - O F F  A N A L Y S I S *******
#                          AND
#       ******* P A R E T O  O P T I M A L *******

... details of performance versus reliability plot removed ....

print ""
print ""
print "========================= Finished ========================="
print ""
```

Appendix B
Television Component

This appendix contains complete details of Television.java, an implementation of the compoent interface specification for television components.

source code

```
Television.java

package javaBackend;

public class Television implements Component{
   private String [] input;
   private String [] output;
   private String name;
   private double width;
   private double height;
   private double weight;
   private double thickness;
   private int cost;
   private int performance;
   private double reliability;

   // Television constructor methods ...

   public Television(String name){
      this.name = name;
   }

   public Television(String name, String [] input, String [] output, double width,
                     double height, double weight, double thickness, int cost,
                     int performance, double reliability){
      this.name   = name;
      this.input  = input;
      this.output = output;
      this.width  = width;
      this.height = height;
      this.weight = weight;
      this.thickness = thickness;
      this.cost      = cost;
      this.performance = performance;
      this.reliability = reliability;
   }

   public void setName(String name){
      this.name = name;
   }

   public String getName(){
```

```
   return name;
}

public void setInput (String [] input){
   this.input = input;
}

public String [] getInput(){
   return input;
}

public void printInput(){
   System.out.println(name + " Input:");
   for(int index = 0; index< input.length; index++){
       System.out.print(input[index] + ", ");
   }
   System.out.println();
}

public void setOutput(String [] output){
   this.output = output;
}

public String [] getOutput(){
   return output;
}

public void printOutput(){
   System.out.println(name + " Output:");
   for(int index = 0; index< output.length; index++){
      System.out.print(output[index] + ", ");
   }
   System.out.println();
}

public void setWidth(double width){
   this.width = width;
}

public double getWidth(){
   return width;
}

public void printWidth(){
   System.out.print(name + " Width = " + width);
   System.out.println();
}

public void setHeight(double height){
   this.height = height;
}

public double getHeight(){
   return height;
```

```
}

public void printHeight(){
   System.out.print(name + " Height = " + height);
   System.out.println();
}

public void setWeight(double weight){
   this.weight = weight;
}

public double getWeight(){
   return weight;
}

public void printWeight(){
   System.out.print(name + " Weight = " + weight);
   System.out.println();
}

public void setThickness(double thickness){
   this.thickness = thickness;
}

public double getThickness(){
   return thickness;
}

public void printThickness(){
   System.out.print(name + " Thickness = " + thickness);
   System.out.println();
}

public void setCost(int cost){
   this.cost = cost;
}

public int getCost(){
   return cost;
}

public void printCost(){
   System.out.print(name + " Cost = " + cost);
   System.out.println();
}

public void setPerformance(int performance){
   this.performance = performance;
}

public int getPerformance(){
   return performance;
}
```

```
   public void printPerformance(){
      System.out.print(name + " Performace = " + performance);
      System.out.println();
   }

   public void setReliability(double reliability){
      this.reliability = reliability;
   }

   public double getReliability(){
      return reliability;
   }

   public void printReliability(){
      System.out.print(name + " Reliability = " + reliability);
      System.out.println();
   }
}
```

Appendix C
Amplifier Component

This appendix contains complete details of Amplifier.java, an implementation of the compoent interface specification for amplifier components.

source code

```
Amplifier.java

package javaBackend;

public class Amplifier implements Component{
   private String [] input;
   private String [] output;
   private String name;
   private int cost;
   private int powerHandling;
   private int performance;
   private double reliability;

   // Amplifier constructor methods ...

   public Amplifier(String name){
      this.name = name;
   }

   public Amplifier(String name, String [] input, String [] output, int cost,
                    int powerHandling, int performance, double reliability){
      this.input  = input;
      this.output = output;
      this.cost   = cost;
      this.powerHandling = powerHandling;
      this.performance   = performance;
      this.reliability   = reliability;
   }

   public void setName(String name){
      this.name = name;
   }

   public String getName(){
      return name;
   }

   public void setInput(String [] input){
      this.input = input;
   }

   public String [] getInput(){
```

```
   return input;
}

public void printInput(){
   System.out.println(name + " Input:");
   for(int index = 0; index< input.length; index++){
       System.out.print(input[index] + ", ");
   }
   System.out.println();
}

public void setOutput(String [] output){
   this.output = output;
}

public String [] getOutput(){
   return output;
}

public void printOutput(){
   System.out.println(name + " Output:");
   for(int index = 0; index< output.length; index++){
       System.out.print(output[index] + ", ");
   }
   System.out.println();
}

public void setCost(int cost){
   this.cost = cost;
}

public int getCost(){
   return cost;
}

public void printCost(){
   System.out.print(name + " Cost = " + cost);
   System.out.println();
}

public void setPowerHandling(int powerHandling){
   this.powerHandling = powerHandling;
}

public int getPowerHandling(){
   return powerHandling;
}

public void printPowerHandling(){
   System.out.print(name + " Power Handling = " + powerHandling);
   System.out.println();
}

public void setPerformance(int performance){
      this.performance = performance;
   }

   public int getPerformance(){
      return performance;
   }

   public void printPerformance(){
      System.out.print(name + " Performance = " + performance);
      System.out.println();
   }

   public void setReliability(double reliability){
      this.reliability = reliability;
   }

   public double getReliability(){
      return reliability;
   }

   public void printReliability(){
      System.out.print(name + " Reliability = " + reliability);
      System.out.println();
   }
}
```

Appendix D
Speaker Component

This appendix contains complete details of Speaker.java, an implementation of the compoent interface specification for speaker components.

source code

```
Speaker.java

package javaBackend;


public class Speaker implements Component{
   private String [] input;
   private String [] output;
   private String name;
   private int [] powerHandling;
   private int cost;
   private int performance;
   private double reliability;

   // Speaker constructor methods ...

   public Speaker(String name){
      this.name=name;
   }

   public Speaker(String name, String [] input, String [] output, int [] powerHandling,
                  int cost, int performance, double reliability){
      this.input  = input;
      this.output = output;
      this.powerHandling = powerHandling;
      this.cost          = cost;
      this.performance   = performance;
      this.reliability   = reliability;
   }

   public void setName(String name){
      this.name = name;
   }

   public String getName(){
      return name;
   }

   public void setInput(String [] input){
      this.input = input;
   }
```

```
public String [] getInput(){
   return input;
}

public void printInput(){
   System.out.println(name + " Input:");
   for(int index = 0; index< input.length; index++){
       System.out.print(input[index] + ", ");
   }
   System.out.println();
}

public void setOutput(String [] output){
   this.output = output;
}

public String [] getOutput(){
   return output;
}

public void printOutput(){
   System.out.println(name + " Output:");
   for(int index = 0; index< output.length; index++){
       System.out.print(output[index] + ", ");
   }
   System.out.println();
}

public void setPowerHandling(int [] powerHandling ){
   this.powerHandling = powerHandling;
}

public int [] getPowerHandling(){
   return powerHandling;
}

public void printPowerHandling(){
   System.out.print(name + " Power Handling = " + powerHandling);
   System.out.println();
}

public void setCost(int cost){
   this.cost = cost;
}

public int getCost(){
   return cost;
}

public void printCost(){
   System.out.print(name + " Cost = " + cost);
   System.out.println();
}
```

```
   public void setPerformance(int performance){
      this.performance = performance;
   }

   public int getPerformance(){
      return performance;
   }

   public void printPerformance(){
      System.out.print(name + " Performance = " + performance);
      System.out.println();
   }

   public void setReliability(double reliability){
      this.reliability = reliability;
   }

   public double getReliability(){
      return reliability;
   }

   public void printReliability(){
      System.out.print(name + " Reliability = " + reliability);
      System.out.println();
   }
}
```

Appendix E
System Requirement

This appendix contains complete details of Requirement.java, an implementation of the specification for system requirements.

source code

```
Requirement.java

package javaBackend;

public class Requirement {
   private String name;
   private String title;
   private int level;
   private String [] dependOn;
   private String derivedBy;

   // Requirement constructor methods ...

   public Requirement(String name){
      this.name = name;
   }

   public Requirement(String name, String title, int level, String [] dependOn,
                     String derivedBy){
      this.name  = name;
      this.title = title;
      this.level = level;
      this.dependOn  = dependOn;
      this.derivedBy = derivedBy;
   }

   public void setName(String name){
      this.name = name;
   }

   public String getName(){
      return name;
   }

   public void setTitle(String title){
      this.title = title;
   }

   public String getTitle(){
      return title;
   }
   public void printTitle(){
      System.out.print(name + " Title: " + title);
      System.out.println();
   }

   public void setLevel(int level){
      this.level = level;
   }

   public int getLevel(){
      return level;
   }

   public void printLevel(){
      System.out.print(name + " Level = " + level);
      System.out.println();
   }

   public void setDependOn(String [] dependOn){
      this.dependOn = dependOn;
   }

   public String [] getDependOn(){
      return dependOn;
   }

   public void printDependOn(){
      if(dependOn == null){
         System.out.print(name + " Depends on: " + "null");
         System.out.println();
      }
      if(dependOn != null){
         System.out.print(name + " Depends on: ");
         for(int index = 0; index< dependOn.length; index++){
             System.out.print(dependOn[index] + ", ");
         }
         System.out.println();
      }
   }

   public void setDerivedBy(String derivedBy){
      this.derivedBy = derivedBy;
   }

   public String getDerivedBy(){
      return derivedBy;
   }

   public void printDerivedBy(){
      System.out.print(name + " Derived by: " + derivedBy);
      System.out.println();
   }
}
```

Appendix F
Generate Component CSV file

This appendix contains complete details of GenerateComponentCsv.java, an implementation of the specification for generating component CSV files.

source code

```
GenerateComponentCsv.java

package javaBackend;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class GenerateComponentCsv {
   private File rdf;
   private FileWriter cmp;

   // Create CSV file ...

   public GenerateComponentCsv(String file){
      rdf = new File(file);
   }

   private File getRdf(){
      return rdf;
   }

   // Write to file for televisions ...

   public void writeToFileTelevision(Library<Television> lib){
      if(lib == null){
         System.out.print("Library is null");
         System.exit(0);
      }
      try {
         FileWriter tv = new FileWriter(getRdf());
         for(Television i : lib.getLibrary()){
             tv.write(i.getName() + ",category," + "television" + "\n");
             tv.write(i.getName() + ",cost," + i.getCost()+ "\n");
             tv.write(i.getName() + ",performance," + i.getPerformance()+ "\n");
             tv.write(i.getName() + ",reliability," + i.getReliability()+ "\n");
             tv.write(i.getName() + ",width," + i.getWidth()+ "\n");
             tv.write(i.getName() + ",height," + i.getHeight()+ "\n");
             tv.write(i.getName() + ",thickness," + i.getThickness()+ "\n");
             tv.write(i.getName() + ",weight," + i.getWeight()+ "\n");
```

```
            for(int x = 0; x < i.getInput().length; x++){
                tv.write(i.getName() + ",input"+ "," + i.getInput()[x] + "\n");
            }

            for(int x = 0; x < i.getOutput().length; x++){
               tv.write(i.getName() + ",output"+ "," + i.getOutput()[x] + "\n");
            }
        }
        tv.close();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}

// Write to file for amplifiers ...

public void writeToFileAmplifier(Library<Amplifier> lib){
    if(lib == null){
        System.out.print("Library is null");
        System.exit(0);
    }
    try {
        FileWriter amp = new FileWriter(getRdf());
        for(Amplifier i : lib.getLibrary()){
            amp.write(i.getName() + ",category," + "amplifier" + "\n");
            amp.write(i.getName() + ",cost," + i.getCost()+ "\n");
            amp.write(i.getName() + ",performance," + i.getPerformance()+ "\n");
            amp.write(i.getName() + ",reliability," + i.getReliability()+ "\n");
            amp.write(i.getName() + ",power handling," + i.getPowerHandling()+ "\n");

            for(int x = 0; x < i.getInput().length; x++){
                 amp.write(i.getName() + ",input" + "," + i.getInput()[x] + "\n");
            }

            for(int x = 0; x < i.getOutput().length; x++){
                amp.write(i.getName() + ",output" + "," + i.getOutput()[x] + "\n");
            }
        }
        amp.close();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}

// Write to file for speakers ...

public void writeToFileSpeaker(Library<Speaker> lib){
    if(lib == null){
        System.out.print("Library is null");
        System.exit(0);
    }
    try {
      FileWriter speaker = new FileWriter(getRdf());
      for(Speaker i : lib.getLibrary()){
          speaker.write(i.getName() + ",category," + "speaker" + "\n");
          speaker.write(i.getName() + ",cost," + i.getCost()+ "\n");
          speaker.write(i.getName() + ",performance," + i.getPerformance()+ "\n");
          speaker.write(i.getName() + ",reliability," + i.getReliability()+ "\n");

          for(int x = 0; x < i.getPowerHandling().length; x++){
              speaker.write(i.getName() + ",power handling"+ "," +
                            i.getPowerHandling()[x] + "\n");
          }

          for(int x = 0; x < i.getInput().length; x++){
              speaker.write(i.getName() + ",input" + "," + i.getInput()[x] + "\n");
          }

          for(int x = 0; x < i.getOutput().length; x++){
              speaker.write(i.getName() + ",output" + "," + i.getOutput()[x] + "\n");
          }
      }
      speaker.close();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Appendix G
Generate Requirement CSV file

This appendix contains complete details of GenerateRequirementCsv.java, an implementation of the specification for generating a requirement CSV file.

source code

```
GenerateRequirementCsv.java

package javaBackend;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class GenerateRequirementCsv {
   private File rdf;
   private FileWriter cmp;

   // Create CSV file ...

   public GenerateRequirementCsv(String file){
      rdf = new File(file);
   }

   private File getRdf(){
      return rdf;
   }

   // Write to file for requirements ...

   public void writeToFileRequirement(ArrayList<Requirement> list){
   try {
      FileWriter req = new FileWriter(getRdf());
      int dependencyLength = 0;
      for(Requirement i: list){
          req.write(i.getName() + ",category," + "requirement" + "\n");
          req.write(i.getName() + ",id," + i.getName()+ "\n");
          req.write(i.getName() + ",title," + i.getTitle()+ "\n");
          req.write(i.getName() + ",level," + i.getLevel()+ "\n");
          req.write(i.getName() + ",derived by," + i.getDerivedBy()+ "\n");

          if(i.getDependOn() != null){
             dependencyLength= i.getDependOn().length;
             for(int x = 0; x < dependencyLength; x++){
                 req.write(i.getName() + ",depends on"+"," +
                          i.getDependOn()[x] + "\n");
```

```
              }
           }
      }
      req.close();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```