FRAMEWORK FOR REQUIREMENTS-DRIVEN SYSTEM DESIGN AUTOMATION

by

Mihai Fonoage

A Dissertation Submitted to the Faculty of

The College of Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Florida Atlantic University

Boca Raton, FL

December 2010

FRAMEWORK FOR REQUIREMENTS-DRIVEN SYSTEM DESIGN AUTOMATION

by

Mihai Fonoage

This dissertation was prepared under the direction of the candidate's dissertation advisor, Dr. Ionut Cardei, Department of Computer and Electrical Engineering and Computer Science, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

SUPERVISORY COMMITTEE:

Ionut Cardei, Ph.D.
Dissertation Advisor

Mihaela Cardei, Ph.D.

Ravi Shankar, Ph.D.

Xin Wang, Ph.D.

Borko Furth, Ph.D.
Chair, Department of Computer and Electrical Engineering and Computer Science

Karl K. Stevens, Ph.D.
Dean, College of Engineering and Computer Science

Barry T. Rosson, Ph.D.
Dean, Graduate College

November 4, 2010
Date

ACKNOWLEDGEMENTS

I thank each of the committee members for the valuable comments and feedback. Particularly, I owe special gratitude to my dissertation advisor and mentor, Dr. Ionut Cardei, for being a constant source of support and encouragement, and for providing me with technical advice any time I needed one. His guidance and help have made my Ph.D. program an enjoyable and productive experience. I thank Dr. Mihaela Cardei, Dr. Ravi Shankar, and Dr. Xin Wang for their support as my committee members.

I thank my colleagues Ms. Yinying Yang, and Mr. Quan Yuan for their friendship during my Ph.D, making it an easier and excitement experience.

I am grateful for the department financial support during the whole period of my doctorate studies. I have received continuous help from many faculty members and staff: Dr. Borko Furht, Dr. Imad Mahgoub, Dr. Eduardo Fernandez, Dr. Sam Hsu, Dr. Martin Solomon, Dr. Lofton Bullard, Dr. Oge Marques-Filho, Dr. Ankur Agarwal, Ms. Tami Sorgente, Ms. Joy Woodworth, Ms. Jean Mangiaracina, Ms. Helene Tomaszewski, and Ms. Rosemary Stewart. The classes and labs experience has greatly helped me enrich my technical and teaching experience.

My doctorate work has been sustained in part through the generous support from Mr. Jaime Borras, former Motorola Corporate Senior Fellow and Senior Vice President of Technology, and his colleagues from the Advanced Technology Group. The findings and opinions of this work pertain solely to the author and are not necessarily those of the sponsors or collaborators.

Finally, I am deeply thankful to my parents, Dorina and Vasile Fonoage, my sister Anamaria Knight, and especially my wife Mirela Fonoage. Without their sacrifices and their continuous support, this work would not have been possible.

ABSTRACT

Author:	Mihai Fonoage
Title:	Framework for Requirements-Driven System Design Automation
Institution:	Florida Atlantic University
Dissertation Advisor:	Dr. Ionut Cardei
Degree:	Doctor of Philosophy
Year:	2010

In this thesis, a framework for improving model-driven system design productivity with Requirements-Driven Design Automation (RDDA) is presented. The key to the proposed approach is to close the semantic gap between requirements, components and architecture by using compatible semantic models for describing product requirements and component capabilities, including constraints. An ontology-based representation language is designed that spans requirements for the application domain, the software design domain and the component domain. Design automation is supported for architecture development by machine-based mapping of desired product/subsystem features and capabilities to library components and by synthesis and maintenance of Systems Modeling Language (SysML) design structure diagrams. The RDDA

framework uses standards-based semantic web technologies and can be integrated with exiting modeling tools.

Requirements specification is a major component of the system development cycle. Mistakes and omissions in requirements documents lead to ambiguous or wrong interpretation by engineers, causing errors that trickle down in design and implementation with consequences on the overall development cost. We describe a methodology for requirements specification that aims to alleviate the above issues and that produces models for functional requirements that can be automatically validated for completeness and consistency. The RDDA framework uses an ontology-based language for semantic description of functional product requirements, SysML structure diagrams, component constraints, and Quality of Service. The front-end method for requirements specification is the SysML editor in Rhapsody. A requirements model in Web Ontology Language (OWL) is converted from SysML to Extensible Markup Language Metadata Interchange (XMI) representation. The specification is validated for completeness and consistency with a ruled-based system implemented in Prolog. With our methodology, omissions and several types of consistency errors present in the requirements specification are detected early on, before the design stage.

Component selection and design automation have the potential to play a major role in reducing the system development time and cost caused by the rapid change in technology advances and the large solution search space. In our work, we start from a structured representation of requirements and

components using SysML, and based on specific set of rules written in Prolog, we partially automate the process of architecture design.

FRAMEWORK FOR REQUIREMENTS-DRIVEN SYSTEM DESIGN AUTOMATION

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

Most of the current system development methodologies follow the flow below:

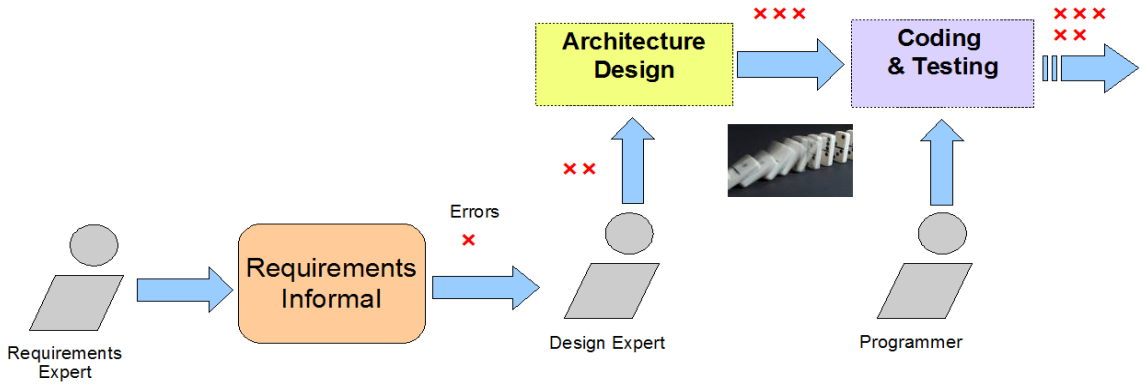

Figure 1.1 Current Development Method

As part of the system development cycle, development iteration begins with requirements specification, where marketing specialists and product managers describe functional requirements, technical specifications, features and use cases in natural language, in a semi-formal format, such as Marketing Requirements Document, Unified Modeling Language (UML) [1], and Systems Modeling Language (SysML) [2], or using requirements management tools such

as DOORS [3] or RequisitePro [4]. Following the requirements specification, system architects and engineers create a hardware/software architecture design that must fulfill all the requirements and satisfy any specified constraints. This design stage includes mapping the features, QoS constraints, and behaviors to a component-based (and hierarchical) architecture. This product decomposition process involves QoS translation (from product to sub-product to component level), matching requirements/constraints to components, and component configuration. The implementation and deployment stages follow thereafter.

The transition from requirements to an architecture design is largely done manually with the help of UML modeling tools. In most cases designers have available a considerable volume of pre-existing components and frameworks, from earlier projects or from third parties. At an abstract level, building a component architecture from requirements is a search in a design space. With libraries holding hundreds of components, this search could take considerable time and manpower, especially when requirements are updated frequently or what-if exploration and trade-off analyses are performed.

Not only do requirements errors or changes permeate through the development stages, but also there is a domino effect affecting the entire development process.

The aim of this thesis is to consider the system development problem from the viewpoint of software decomposition.

1.2 An Application Example

We consider how our requirements-driven automation approach can be applied to the Location-Based Services domain. The services available are as follows:

- Location Service: provides information related to the a location desired by user. Such services can come in the form of turn-by-turn navigation to a specific address or locating people on a map.
- Proximity Service: provides information in the form of alerts or notifications when the user comes in proximity with a specific point of interest. Some examples could be alerts when the user has reached the proximity of a specific restaurant, hotel, or movie theater.
- Place Finder Service: provides information on specific points of interest requested by the user. Examples would be finding someone or something, like a doctor office or an Italian restaurant; proximity-based notification like target advertising, buddy list, automatic airport checking; proximity-based actuation such as payments made based on proximity (EZ pass, toll watch).

The GPS that calculates the position is provided by different companies such as Texas Instruments, SiRF, and Broadcom. Each such component comes with its own set of properties. We are interested in those that match the initial requirements of the system. Such properties include:

- Location Accuracy: represents the precision of the location the GPS provides to the user.

- Query Response Time: indicates the time needed for the GPS to respond to a location request.
- Altitude Information: GPS provide information regarding the elevation of the user from the sea level.
- Speed Information: GPS provides information related to the current speed of the user.
- A-GPS Support: Assisted GPS mode is supported to enhance the startup performance of the GPS.
- Cold Start – GPS Receiver cold starts after a power interruption of more than about one minute or after a cold start command sent to it via the serial port. When cold starting the GPS Receiver has no information about the current time, the orbits of the satellites or its own current position.
- Warm Start – Previous data is available to the GPS, but it needs additional or updated data.
- Power Consumption: denotes how much power the GPS consumes. This is mostly related to the GPS Receiver.

Our work example is simpler than what it is required for practical systems, in order to keep the presentation simple. In practice, there could be more location services available. In addition, GPS properties are more complex, numerous, and varied.

1.3 Research Questions

The overall research question this thesis tries to answer is:

How can we dramatically increase the productivity of system development?

In order to be able to answer this question, we define a set of research questions that address the problem in detail.

RQ1: How can we model requirements?

- What tools can we used to represent requirements?
- What restrictions on the representation of requirements do we need to impose in order to be able to extract specific information out of them?
- How can we translate requirements into a more convenient format for processing.

RQ2: How do we validate the requirements?

- What is needed to validate requirements?
- How should valid requirements be?

RQ3: How can we specify software and hardware system components?

- What tools can we use to represent components?
- What kind of metadata should we tag the components with?

RQ4: How do we automate architecture synthesis?

RQ5: How can we achieve an optimal architecture synthesis?

RQ6: What is the computational complexity of your methods?

1.4 Proposed Solution

We have approached the problem by designing a framework for Requirements-Driven Design Automation that aims to reduce the cost of system design by partially automating the process of architecture decomposition from requirements to existing library components. The framework can be applied to design of software and systems that use UML or SysML.

The key to the proposed approach is to close the semantic gap between requirements, components, and architecture by using compatible semantic models for describing both product requirements and component capabilities, including constraints. A domain-specific representation language entitled OPP is designed that spans the application domain (mobile applications, in our case), the software design domain (UML/SysML meta-schema), and the component domains. This language is used to represent ontologies, which are textual representations that capture the semantics of concepts common to product requirements and design modeling, and the relationships between them. The ontology (metamodel) for requirements specifications is based on the Semantic Web Ontology Web Language (OWL) [5]. It covers concepts for product requirements (features and structure), and a set of constraint checking rules. These rules permit consistency and completeness validation for requirements models before their use in architecture design. The RDDA ontology is expanded to cover knowledge representation for system architecture (UML and SysML diagrams) and for components (semantic annotations). In addition to specifications for product/subsystem features and capabilities, the RDDA

ontology supports Quality of Service and system resource constraints, such as energy, CPU load, bandwidth, weight, and volume.

Design automation is supported for architecture development by component selection and design structure synthesis. Automated selection of components from libraries is useful when the number of candidate components is large or when there are many constraints that have to be met, including dependencies. Criteria for component selection include interfaces required and provided, implementation platform, capabilities and constraints that have to be satisfied. A selection criterion that cannot be represented in UML/SysML is described as OWL annotation metadata. Components that match the requirements and provide the necessary interfaces are pulled from the library to populate a structural UML diagram.

Synthesis of design structure diagrams takes the process further by producing new diagrams that can be edited by the user. It works bottom-up from existing structural models describing design relationships and derives feasible configurations represented as UML structural diagrams that satisfy the requirements. The RDDA framework currently supports functional requirements expressed as required capabilities and constraints, such as symbolic elements (e.g. “supports GPS localization”) and numeric elements (e.g. “maximum latency is 10 s” or “cost between $10 and $20”.

The ontology representation also supports requirements tracing to design artifacts. This function is generally used by modeling software with requirements

management capabilities to track design and implementation artifacts that are affected by changes to requirements.

The RDDA framework is designed to integrate with UML/SysML modeling tools compatible with OMG's XML Metadata Interchange (XMI) format. Design models are translated to and from OWL specifications using Extensible Stylesheet Language Transformations (XSLT). This approach bypasses the limitations of the MDA's Queries/Views/Transformations (QVT), such as XMI-only conversion. At the core of the RDDA architecture is a reasoning framework built on top of the Prolog's Inference Engine [6].

1.5 Thesis Outline

This thesis presents a framework for requirements driven system design automation, describing the work, the results, and the contributions. An outline of the structure of this thesis is shown as follows:

Chapter 2 provides the background for this thesis. Concepts related to modeling languages, markup and stylesheet languages, semantic web, Prolog and Java are presented so as to better prepare and help the reader understand the concepts presented throughout this thesis.

Chapter 3 introduces the related work, comparing and contrasting previous solutions to similar problems.

Chapter 4 discusses the requirements workflow, starting with the specification and modeling of requirements, and going through the transformation, processing, and validation of those requirements.

Chapter 5 presents a methodology for automated architecture synthesis and component selection based on validated requirements models and semantic component specifications.

Chapter 6 introduces the architecture optimization related to Quality of Services and Resource Utilization parameters. The description of what is an optimization problem, how to build one, and how we solve it is given here.

Chapter 7 evaluates the framework proposed, based on publications, prior work and its limitations.

Chapter 8 summarizes the results and contributions, and outlines the future directions for this work.

Chapter 2

BACKGROUND

2.1 Modeling Languages

Modeling is a way to capture ideas, relationships, and data, broadly speaking, any information needed, using a well-defined notation. A modeling language is a language that can be used to represent data or systems in a way consistent with a well-defined set of rules. In this thesis, we use modeling languages that express information in a graphical manner, using diagrams. Two such languages are described next: the Unified Modeling Language (UML) [1] and the Systems Modeling Language (SysML) [2].

2.1.1 Unified Modeling Language

The Unified Modeling Language is a widely used general purpose modeling language defined by the Object Management Group (OMG) [7] to model the business process, structure, behavior, integration, and architecture of software systems. There are two broad types of diagrams in UML:

- Structural Diagrams, describing the static structure of the system. Some examples of such diagrams are Class Diagram, Component Diagram, Package Diagram, and Deployment Diagram.

- Behavioral Diagrams, describing the dynamic behavior of the system. Some examples of such diagrams are Activity Diagram, Use Case Diagram, and Sequence Diagram.

There are four distinct views of a system and one overview of how everything fits together:

- Design view, capturing the classes, interfaces, and patterns that describe the problem domain and how the software we build will address it.
- Deployment view, capturing how a system is configured, installed, and executed.
- Implementation view, capturing the components, files, and resources utilized by the system.
- Process view, capturing the concurrency, performance, and scalability of the system.

The final view, that brings together the above four mentioned views, is the Use Case view that captures the functionality required by the end user.

Some of the drawbacks of UML come from its orientation towards modeling software-related concepts, which made it very difficult for system engineers to adopt. Furthermore, there is no direct support for requirements and parametric models in UML. These limitations motivated the decision to customize UML for system engineering, which resulted in the development of the Systems Modeling Language described next.

2.1.2 Systems Modeling Language

OMG's Systems Modeling Language (SysML) is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying systems such as hardware and software, information, personnel, procedures and facilities. Because of the lack of support for specific system engineering concepts in UML, SysML provides a semantic foundation for modeling requirements and parametrics. Defined as a subset of UML 2.0 with extensions, the relationship between SysML and UML is depicted in Figure 2.1 below:

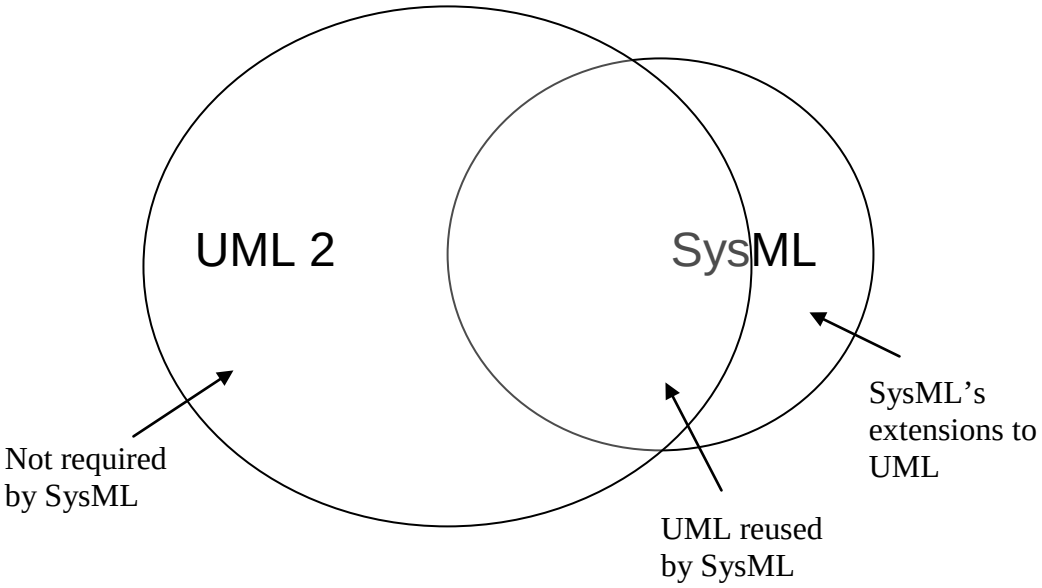

Figure 2.1 Relationship between SysML and UML

SysML uses XML Metadata Interchange (XMI) (explained in subsequent chapters) to exchange modeling data between different modeling tools.

Figure 2.2 gives an overview of the diagrams in SysML:

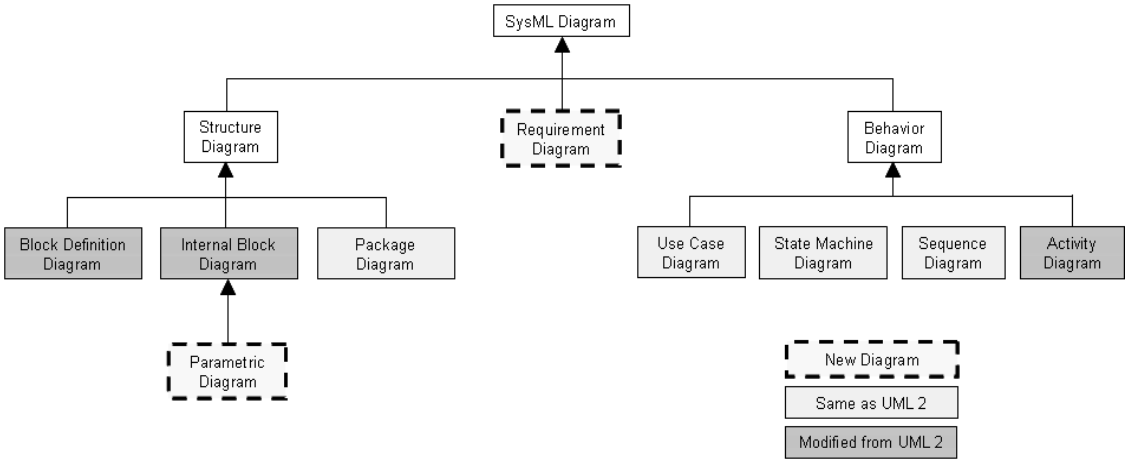

Figure 2.2 SysML Diagram Types

The overlap between SysML and UML 2.0 is clear, as well as what has been changed or added in order to better support system engineers. There are four main views that should be highlighted: Structure, Behavior, Requirements, and Parametric. The system structure is expressed in SysML with Internal and External Block Diagrams, which describe the internal and external composition of a system, subsystem, etc. The behavior of a system is encapsulated in Use Case Diagrams, Sequence Diagrams, State Machine or Statechart Diagrams, and Activity Diagrams. The first three from this group are unchanged from UML 2.0, while the Activity Diagram has been modified as will be described shortly. SysML extends UML 2.0 with the Requirement and Parametric Diagrams, used to express system requirements and system parametric equations. We first describe structure diagrams, followed by behavior and requirements diagrams.

Blocks in SysML are the basic unit of structure. A block incorporates a set of properties used to describe a system, sub-system, and components that are part of the system, such as software, hardware, logical and physical elements. Parts in these systems interact in many different ways, such as by means of software operations, discrete state transitions, flows of input and output, or continuous interactions.

Blocks are based on UML 2 structured classes, which support the ability to hold ports, parts, and connectors. Some elements specific to UML 2 classes have been eliminated, such as association classes; also, blocks differentiate between value properties and part properties (usage of a block in the context of an enclosing block). An advantage of blocks is that, depending on the development phase, they can provide different levels of detail, starting with a basic outline, and going to exhaustive detail for simulation or executions. Multiple compartments of a block can describe various characteristics, such as properties (parts, values, and references), operations, constraints, allocations to the block, like activities, requirements the block satisfies.

2.1.3 Modeling Tools

There are a variety of tools used for UML and SysML modeling. A list of UML tools can be found in [8], while a list of SysML vendors can be found in [9]. Some of the most used tools include Sparx System's Enterprise Architect [10], with an easy to use, intuitive user interface, providing automated round tripping of structural models such as classes and interfaces, and plug-in support for

SysML. No Magic's MagicDraw [11] offers powerful drawing capabilities together with an intuitive user interface and SysML support in form of a plug-in. Telelogic's Rhapsody [12], the tool used in during our research efforts also, is an executable modeling tool focused on real-time and embedded systems, capable of producing complete software code. Rhapsody has a strong focus on system engineering supporting the latest version of SysML's specification. Model simulation and execution is provided to verify the models.

For non-proprietary modeling tools, Papyrus and Topcased are based on Eclipse and both offer support for SysML. The downside of this two tools is the lack of intuitiveness of the user interface, making them hard to use.

2.2 Model Driven Architecture

With the establishments of UML 2.0 and SysML, the extensions introduced gear more towards supporting the notion of MDA, or Model Driven Architecture [13], a framework for software development outlined by OMG. The concept of a model takes the central role in the MDA approach because the entire development process is determined by the activity of modeling the system. The artifacts created during the development process are formal models. Three models have been identified:

- PIM (Platform Independent Model), which is independent of any implementation technology and defined at the highest level of abstraction.

- PSM (Platform Specific Model), which resulted from the transformation of the PIM. Depending on the technology platform, a specific PSM is generated such that to specify the system in terms of the available constructs.
- Code, resulted from the transformation of each PSM.

MDA transformations are always carried out by specific tools, thus highlighting the benefits of MDA. Other main benefits are outlined below:

- *Productivity*, since most of the work is done at the PIM level, developers have less work to do because the platform-specific details do not need to be designed because they were addressed in the transformation definition. Furthermore, the focus can be shifted from code to the PIM design, thus solving the business problem at hand, resulting in a system more closely aligned with the needs of the end user.
- *Portability*, for the reason that the focus is on PIMs, which are by definition platform independent, high portability is achieved.
- *Interoperability*, achieved by bridging the PSMs or Code generated, which is done by the tools that generate not only the PSMs, but also the necessary bridges between them.
- *Maintenance and Documentation*. Because the initial model is the exact representation of the code, the PIM fulfills the function of a high-level system documentation.

The types of models that we are concerned with are structural models. In UML, for example, the class diagram is an example of such a model; in SysML, we have the block definition diagram as an example. The specific structural models utilized in the proposed framework are block definition diagram and internal block diagram that are used to capture the structural aspects of the system that is being modeled, thus presenting different views of the system in one model. The benefits of building a model of a system are that it eliminates ambiguities, it is formally verifiable, and it is machine processable.

Additional information about MDA can be found in [14].

2.3 Extensible Markup Language

XML, or Extensible Markup Language [15], is a markup language where one can create your own tags to represent and describe any structure. By defining your own tags, one can structure the data that is being represented in such a way as to be more readable, more descriptive, thus making it easier to understand and use. The success of XML comes from several factors:

- It separates the information represented by the data, form how that data should be presented.
- Easy transmission of data between different entities.

In the proposed framework, many of the technology employed are based on XML or are used in conjunction with XML, such as XMI, OWL, RDF, XPath, and XSLT, all of which will be detailed in the following sections.

2.4 Extensible Markup Language Metadata Interchange

XMI, of Extensible Markup Language Metadata Interchange [16], is a standard adopted by the Object Management Group for exchanging metadata (information about data) through XML. In the RDDA framework, the SysML model is exported through the Rhapsody tool into XMI, which in turn will be further processed to extract useful information. For every aspect captured by the SysML diagrams, there will be a matching description in the exported XMI file. For example, the description of a SysML diagram is capture in the XML element < UML:Diagram >. To describe a package or stereotype, <UML:Package> and <UML:Stereotype> is used. Reading through such a document bears no challenges exactly because of how descriptive these tags are. Since the name of the elements do not change, only their respective value does, processing such documents in a general way becomes much more easier.

2.5 Extensible Stylesheet Language Family

XSL, or Extensible Stylesheet Language Family [17], is a group of recommendations for presenting and transforming an XML document. XSL consist of three languages:

- XSLT (XSL Transformations) – used for transforming XML documents.
- XPath (XML Path Language) – used to navigate through the XML document.
- XSL-FO (XSL Formatting Objects) – used for specifying formatting semantics.

Only the first two languages are used in this thesis, and are described in more details next.

2.5.1 Extensible Stylesheet Language Transformations

XSLT, or Extensible Stylesheet Language Transformations [18], is a W3C Recommendation that converts an XML document into any other type of document, such as XML, HTML, CSV (Comma Separated Value), or even PDF. In the work presented here, XSLT is used to convert XMI documents into OWL files, and OWL files into XMI documents.

In XSLT, the user starts with a template that is gradually filled with parts of the initial XML document, either by manipulating those parts, or by including them altogether. Michael Kay, the editor of the XSLT 2.0 specification, describes the transformation process of an XML document in [19] as follows:

- Data form the incoming XML is transformed into a structure that reflects the desired output.
- Formatting the new structure based on the desired format (XML, PDF, etc).

Before the entire process is started, the XML document is converted into a tree structure. The actual transformation process is expressed as a set of patterns defined as rules, and which denote what output should be generated when a specific pattern (i.e. <xsl:template match="/"> that will match the start of the source document) in the tree representation of the XML has been encountered. There is no sequence of instructions that one has to write in order to achieve

this, but rather it describes what transformations are required to attain the desired output.

2.5.1.1 XSLT Processor

The role of the XSLT Processor is to take the XML source document, apply an XSLT stylesheet, and produce the result document, process which is captured in Figure 2.3 below:

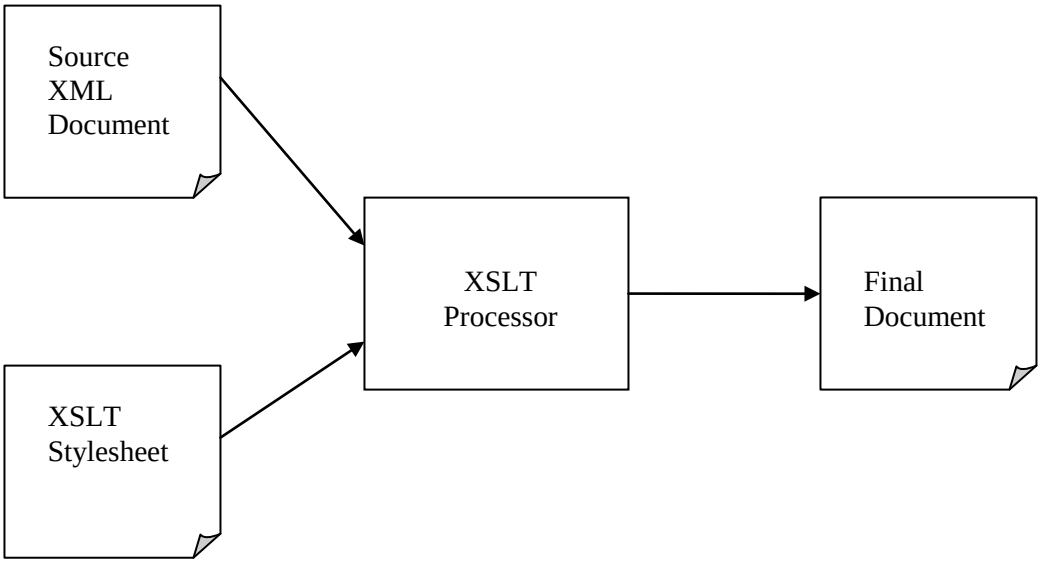

Figure 2.3 XSLT Processing Model

The two inputs in the above figure are a source document, usually in XML format, and a stylesheet containing patterns and translation rules. The XSLT Processor will look for patterns in the source XML document and apply the specified rules on the nodes that fall within that pattern, thus creating bits and pieces of the result representation of the final transformed document. Here are the actual steps taken by the processor:

- Load source document as an optimized DOM tree
- Walk the tree in a depth-first manner
- For each node in the tree, select a template in the stylesheet
- Apply template on the node, creating zero or more nodes in the output document (which is also a tree initially)
- At the end, write the output tree

Because of the tree nature of the input and output, when writing the transformation stylesheet, it is best to think and express the problem in terms of tree manipulation.

2.5.2 Extensible Markup Language Path Language

XPath, or Extensible Markup Language Path Language [20], is an expression language used for accessing and manipulating data in XML documents. It is not used to create or modify nodes, as that is the job of XSLT, but it can however create values, or sequences of values.

There are three categories of operations that XPath can perform [21]:

- Operations on values, such as string comparison and arithmetic functions, very similar to what we have in conventional programming languages
- Operations for node selection, where a path expression is used to select an attribute or a node based on the expression written.

- Operations on sequences, where an expression is applied on every item (usually a node) in a sequence, thus realizing a one-to-one mapping between the input sequence and the output sequence.

XPath is used as a sublanguage within the XSLT stylesheet, and its mostly used for identifying parts of the source document that have to be processed.

2.6 Semantic Web

The current Web represents information using natural language, multimedia, graphics, page layout, and other various ways, all of which are easily processable by humans because of the ability to deduce facts from partial information, and create (mental) associations. Applications have access to this data too, but it becomes very difficult to use partial information, to make sense of data, such as images, and to draw analogies automatically. What is needed then? A way to provide information about a resource (called metadata), that is in a machine processable format such that applications (sometimes called software agents) can reason about the (meta)data. To make such metadata machine processable, resources should be uniquely named (URIs), a common data model for expressing0020metadata should exist (RDF), and a common vocabulary should be defined (Ontologies). This is where the Semantic Web comes into play.

The Semantic Web [22] is an extension to the World Wide Web where data is annotated with metadata such as to better enable computers, through

software programs, to manipulate the data easily and more useful. Once the data is mapped with useful information, queries can be performed, such as “Give me all title of papers written by Mihai Fonoage on the subject of semantic web”. Because of the annotation process, and because the data is independent of its internal representation, such queries can be performed automatically.

The main layers of the Semantic Web are shown in Figure 2.4 below:

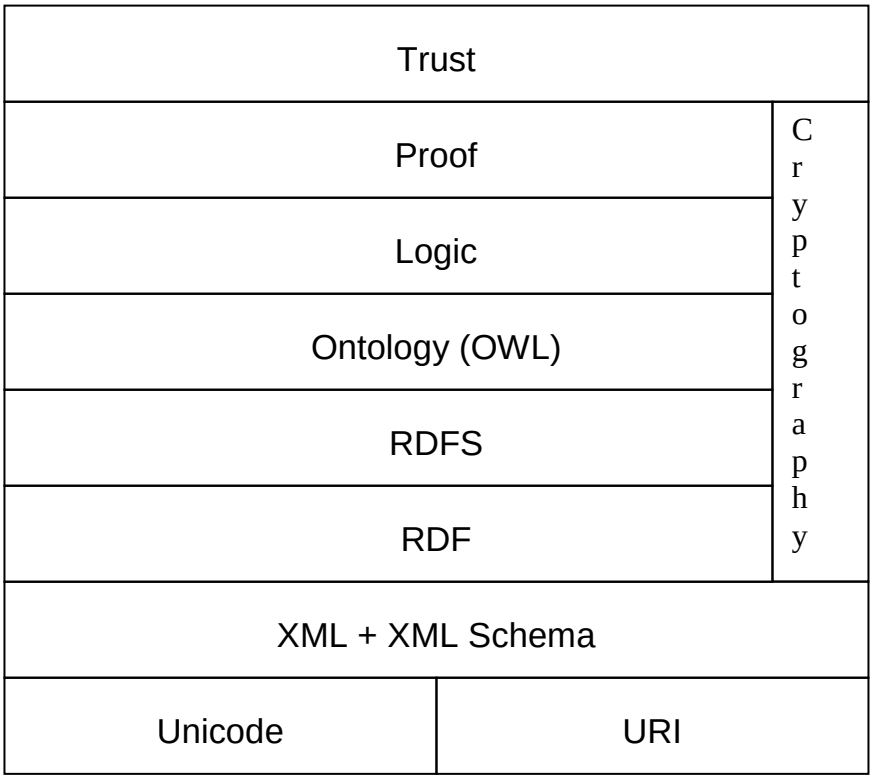

Figure 2.4 The layered Semantic Web specification architecture

The main layers are:

- URI for uniquely describing resources;

- XML for writing structured Web documents with user-defined vocabulary, plus sending documents across the Web; a detailed description has been provided in section 2.3.
- RDF is a data model for writing statements about Web objects (resources);
- RDF Schema provides modeling primitives (such as classes, properties, range restrictions, etc) for organizing Web objects into hierarchies;
- Ontology language to allow representation of more complex relationships between resources;
- Logic layer used to enhance the ontology language by allowing application-specific declarative knowledge;
- Proof layer involves the deductive process + representation of proofs + proof validation
- Trust layer will emerge through the use of *digital signatures* and other kinds of knowledge

The following sections will describe in more detail the semantic web layers mentioned above.

2.6.1 XML Schema

XML Schema [23] is a language for defining the structure of XML documents, and who's syntax is based on XML itself. XML Schema allows the definition of new types by extending or restricting existing ones, and provides a

set of data types that can be used on XML documents. The schema elements are defined next.

Element types have a syntax of <element name="…"/>, and they may have a number of optional attributes such as types or cardinality constraints.

Attribute types have a syntax of <attribute name="…"/> and may have a number of optional attributes such as types, existence, or a default value.

Data types come in many flavors, such as built-in types (numerical, strings, date, etc), and user defined to extend the capability of the built-in types.

2.6.2 Resource Description Framework

RDF, or Resource Description Framework [24], is a language for representing information about resources on the Web, expressing data models that refer to resources and their relationships. The building blocks of RDF are the RDF Triples, a labeled connection between two resources, or more formally, an subject-predicate-object, or (s,p,o), triple called a statement or a triplet, where "s" and "p" are URI-s, i.e. resources on the Web, while "o" is a URI or a literal. Thus, RDF is a general model for such triples. To make the transition to RDF easier, in the requirements diagram inside the RDDA framework, we have the textual representation of the triple "GPS requires Location Service", where "GPS" is the subject, "requires" is the predicate/property, and "Location Service" is the object. Each RDF Triplet forms a directed, labeled graph.

The fundamental concepts of RDF are resources, properties, and statements.

A resource is a “thing” that we want to describe, such as GPS, Phone, Camera, Location Service, etc. Every such resource has a URI, or Universal Resource Identifier, which should uniquely identity the resource.

A property is still a resource, but a special kind describing the relationship between resources, such as “requires”, “provides”, “implements”, “inherits”, etc. Properties are also identified using URIs.

Statements are subject-predicate-object triples consisting of a resource, a property, and a value (other resource or literal, as mentioned before).

It is important to understand that in RDF, making statements about other statements is possible, describing for example belief (or disbelief) in other statements.

2.6.3 Resource Description Framework Schema

RDFS, or Resource Description Framework Schema [25], is an extension to RDF that defines the vocabulary used in RDF data models. In RDFS we can define a vocabulary, together with properties that apply to objects and specify what values they can take, and describe the relationship between those objects [26]. The extra knowledge provided by RDFS is defined in terms of resources and classes, where we know that everything in RDF is a resource, while a class, beside being a resource itself, can also be a collection of possible resources, such as individuals. It is thus possible to specify that an individual belongs to a specific class (“GPS is a Subsystem”), or that an instance of one class is also an instance of another class (“every product is a subsystem”).

The fundamental concepts of RDFS are classes, properties, class hierarchies, and inheritance.

A class is a set of elements, with individuals that belong to a class referred to as instances of that class. This relationship is captured by using rdf:type. A class is defined using rdfs:Class, i.e. <rdfs:Class rdf:ID="RqSubsystem">, where RqSubsystem is a subsystem found in the requirements section of our RDDA framework.

Properties that describe the relationship between resources can be constrained in terms of the values they can have, such that the range or domain of each property is restricted. Such properties are defined using the rdf:Property construct.

Relationships between classes are established by means of hierarchies captured in terms of inheritance ("is a subclass of"), where each class inherits the "abilities" of its superclass is such an inheritance relationship exists. Such relationships are specified using the rdfs:subClassOf property.

2.6.4 Ontologies (OWL)

The expressiveness of RDF and RFDS is limited to predicates (for RDF), or to subclass and property hierarchies, together with domain and range definitions of the properties (in case of RDFS). As such, there was a need to have a language that had a well-defined syntax (so that software programs could process the information), had sufficient expressive power (construct classes, not just name them; restrict property range when used for a specific

class), and allowed for efficient reasoning (be able to reason about some terms by deducing statements that are obvious to humans, but not to programs). This is where ontologies come into play. An ontology, or vocabulary, is used to "*define the concepts and relationships used to describe and represent an area of knowledge*
"[27].

To describe such a vocabulary, one technique employed is the Web Ontology Language, or OWL. It is used to define the terminology in a specific situation, and adds more constraints on properties, and generally, has more facilities of expressing semantics and meanings than RDFS. In terms of describing properties and classes, OWL adds more vocabulary, such as relations between classes (i.e. disjointness), cardinality (i.e. "exactly one"), equality, characteristics of properties (i.e. symmetric), enumerated classes.

In the RDDA framework, ontologies are used to encode domain-specific knowledge relevant for hardware-software co-design for cellphone architectures. They describe requirements, component semantics, system architecture (component relationships), design patterns, and processing rules. OWL is used to define domain-specific vocabulary ("OPP Design Language") that describes SysML concepts, such as classes, inheritance, constraints requirements, and many more. The domain of the mobile phones uses this ontology to further describe the existing components of the domain, their relationships, and characteristics. Below is an example of the definition for a domain-specific metamodel, namely classes, relationships, and properties:

```
<owl:Class rdf:ID="MSwComponent">
        <rdfs:subClassOf>
                <owl:Class rdf:resource="MClassifier"/>
        </rdfs:subClassOf>
</owl:Class>
<owl:ObjectProperty rdf:ID="providesInterface">
        <rdfs:domain rdf:resource="#MGenComponent"/>
        <rdfs:range rdf:resource="#MSwInterface"/>
</owl:ObjectProperty>
```

Another example defines the actual model knowledge, namely instances and property values:

```
<MSwComponent rdf:ID="Camera">
        <providesInterfacerdf:resource="#iCamera"/>
                <providesFeature rdf:resource="# fGIFImageFormat"/>
</MSwComponent>
```

2.7 Prolog

Prolog is a declarative logic programming language that uses formal logic expressed in terms of relations between different types (known as *terms*). A Prolog program consists of a series of *facts* and *rules* that are stored in a database called a *Knowledge Base* (KB). The main goal of a Prolog program is to answer *queries* based on these facts and rules. For example, consider the following facts for the *father* and *male* predicates in the knowledge base:

father(john,megan).

father(john,david).

male(david).

Atoms in lowercase are symbols. The first fact declares that john is megan's father. The rule

son(X,Y) :- father(Y,X), male(X).

declares that X is a son of Y *if* Y is *father* of X *and* X is *male*. The notation for variables uses capitals. The query posed by the user

?- son(david,john).

will return true given the two facts that state that john is father of david, and that david is male. The result is inferred by the Prolog inference engine by reasoning about the information in the knowledge base (the facts and rules), trying to satisfy the query goal(s).

An advantage of Prolog is that it provides backward chaining, meaning that when a (sub)goal fails, Prolog will trace its steps backwards (backtrack) to the previous goal (choice point) and tries to re-satisfy it. The computation is simply undone to the last choice made and a different computation path is taken. With the query son(X,john), the Prolog engine will execute the first sub-goal father(john,X) and unify variable X with megan, then attempt to execute male(megan) that fails because it is not in the KB. At this point, the Prolog engine backtracks and attempts the second sub-goal father(john, X), with variable X being bound to david. This time, the second sub goal of the son predicate will succeed, and the query returns variable X bound to david.

Backtracking is also useful for finding more than one solution to a query.

One of the requirements for our research project was to be able to manipulate constraints. SWI-Prolog, a free standard Prolog platform, provides a library that facilitates constraint programming called CLP(FD). Using constraint programming, we have a declarative formalism that lets one describe conditions a solution must satisfy. As an example of finite domain constraints we have Expr1#=< Expr2, denoting that Expr1 is smaller than or equal to Expr2.

Chapter 3

Related Work

This chapter describes part of the work done in the areas covered by the proposed framework, such as requirements specification and validation, design automation, and design optimization.

The authors in [28] propose a Semantic Web approach for maintaining software, making use of the Resource Description Framework (RDF) for representing information related to software components, the Web Ontology Language (OWL) for describing the relationships between those software components and different metrics, tests, and requirements, and the SPARQL query language for querying the RDF graph to extract information about the state of the software system. Although the solution proposed by the authors provides a way of tracing Requirements to implementation in the source code thus validating them, it does not deal with consistency checking of the Requirements and also of the Models; at the same time, it does not address the conflict resolution at the selection and configuration of UML components. Our work specifically deals with these cases.

An approach based on the use of Ontologies is also proposed in [29] where the authors make different semantic processing in order to analyze the Requirements. Unlike [28], Requirements are checked for inconsistency and incompleteness in a manner similar to ours, by detecting contradicting elements using inference rules. The limitation in [29] is that the authors do not deal with anything beside the Requirements analyzes. In our work, we also show how UML component are selected and configured, and resolve dependencies between products and sub-products. Thus, our work is an important step forward to automating the design flow of a system.

The use of Semantic Descriptions is proposed in [30] for finding, filtering, and integrating Web Services. Because of the dynamic composition of existing services, usages of existing techniques like WSDL and SOAP are insufficient and the authors make use of the Semantic Web to address these challenges, using an inference engine to store information about known services and to find matching services, as well as a composer used as a user interface between the human operator and the inference engine. While this paper focuses on a single case, namely Web Services, our work can be applied to any system. In addition, we are working not only at the component selection and composition, but also at the same time deal with partitioning of the product and checking for consistency of the requirements and the model.

The same problem of Web Service Composition talked about in the previous paper is addressed in [31], where the authors present the current solutions and the existing problems related to the field. The authors present two

industry solutions: the Web Services Definition Language (WSDL) together with the Web Services flow specification language BPEL4WS, and on the other side the Semantic Web solution. Like in [30], the described solutions suffer from the same drawbacks, and a complete and more general solution is needed, thus having our work apply to the universal cases. In [30] and [31], as with our work also, the main idea is the same, mainly to make use of the Semantic Web, but in our work, we broaden the utility of semantics so that we achieve, beside component description and selection, also requirements and model checking and validation.

Similar to the previous two papers mentioned, Ontologies have been applied in [32] to a different area, to Software Patterns. The authors introduce the concept of ONTOPATTERN, an ontology that incorporates knowledge about the description and localization of patterns. Because patterns are included as instances of classes in the ontology, ONTOPATTERN becomes a knowledge base where inferences and searches can be made seemingly thus aiding the reuse of patterns. The authors in [30], [31] and [32] have thus employed Ontologies in specific domains, like Web Services and Patterns, reducing the time for selection and configuration. The difference of our work is that we propose a top-down methodology, instead of the bottom-up described in these papers, where we introduce the concept of ontology-based UML architecture synthesis from requirements and component specification.

In [33] we are presented with a software engineering tool called CASSANDRA, which assists and guides developers through the software

development process. It is implemented in WIN-PROLOG and has different interface agents and application agents so that to adapt to various external applications like CASE tools. The proposed tool goes through the processes of Analysis, Design, Construction, and Project Management. The Analysis Process follows the following activities:

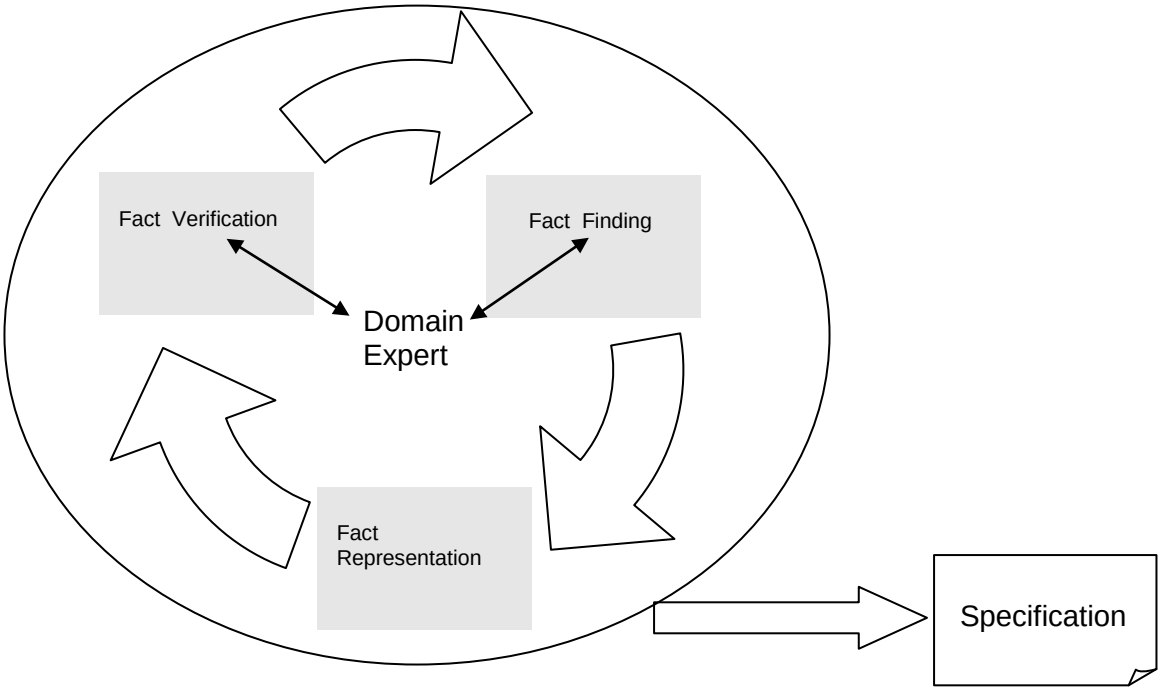

Figure 3.1 CASSANDRA: Analysis Process

The output of the Analysis stage is the Specification, which is also the input of the second stage, the Design Process represented below:

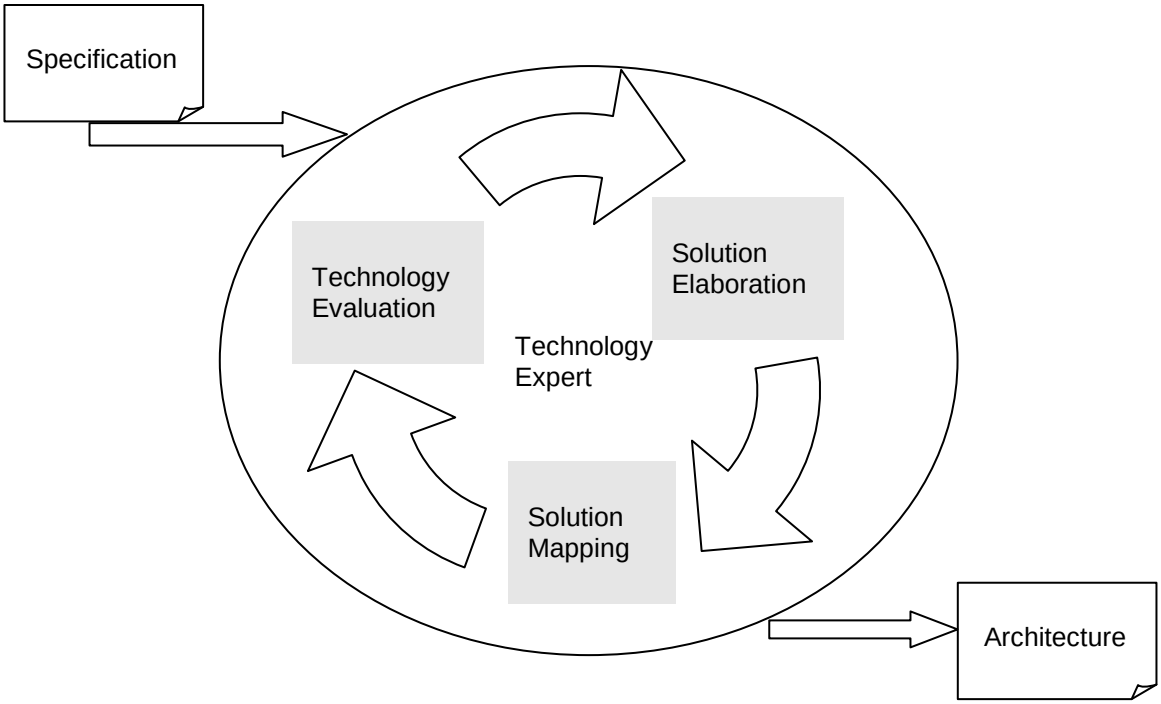

Figure 3.2 CASSANDRA: Design Process

The main deliverable of this process is the Architecture, which together with the specification, are the inputs for the construction process described below:

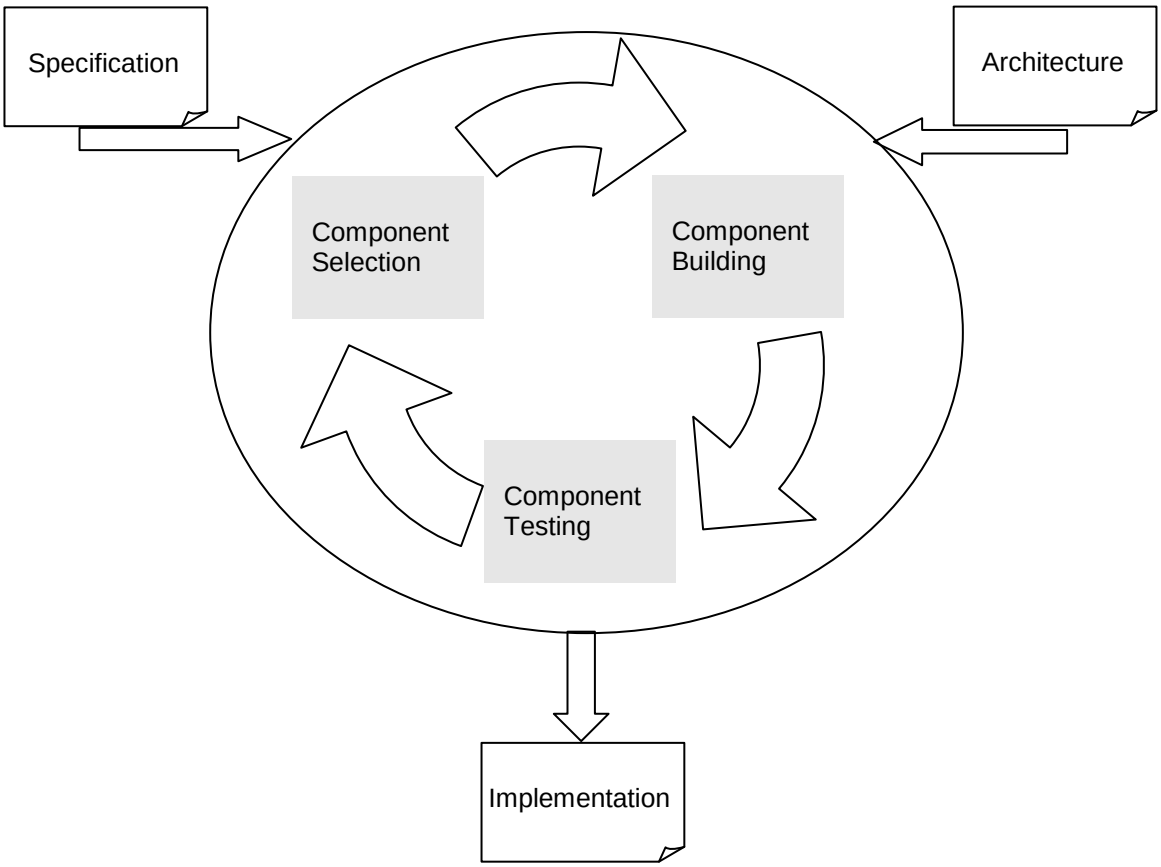

Figure 3.3 CASSANDRA: Construction Process

The deliverable for this stage is the Implementation that consists of the application that is tested and ready to run. Although specification and model checking is provided by means of Prolog and xUML, the project does not deal with finding an optimal set of components based on their QoS constraints.

The authors in [34] propose a framework called *Semantic Streams* in which a user can take advantage of declarative statements to query the sensor network. The principles from the service and semantic domains are combined here to form a semantic services programming model where each service is a

process that deduces semantic data about the world. These services are converted to rules with pre and post conditions and the inference engine uses backward chaining to match every element of the query with the post-condition of a service. In our approach, we follow a similar path for constraint validation and architecture synthesis. As in [30] and [31], composition is obtained by allowing the processes of interpreting information to be wrapped up, forming thus semantically new applications.

In [35] the authors propose a methodology for transforming requirements into a valid, conforming, and dependable formal model or system that is often needed in Sensor Networks. The part of model generation from requirements, together with the validation and verification of that model against the customer's requirements, is automated and achieved through mathematical models. Automatic code generation from requirements is achieved in a way as to avoid a mismatch with the design. The figure below describes the approach:

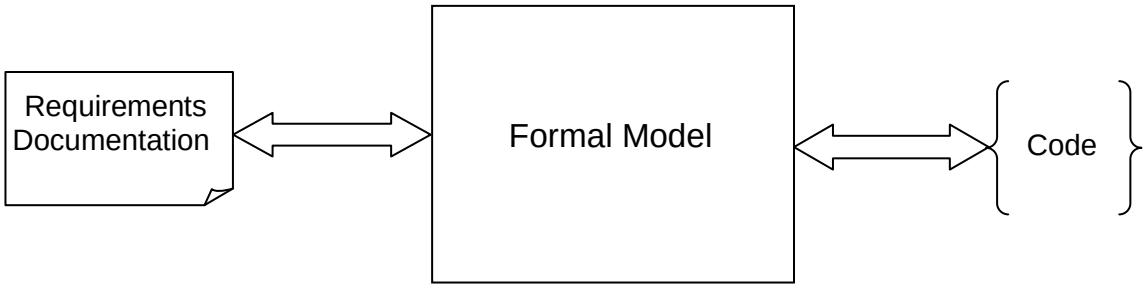

Figure 3.4 Sensor Networks - Requirements Capture

The mathematical approach mentioned above provides this round-trip tractability property to system development. The authors make use of CSP,

Hoare's language of Communicating Sequential Processes, suitable for different types of analysis. Our work although similar in most of the key advantages offered by their system, differs in that it uses Ontologies and Inference Rules for Requirements checking and validation.

In [36] a CASE-tool integration platform called *GeneralStore* is suggested which enables concurrent development of embedded systems at all design phases. The authors make use of UML to store CASE data into MOF object repository, and XMI for interchanging this information with other CASE tools. The main idea behind this approach is the model-based coupling of heterogeneous subsystems provided by the GeneralStore tool. In our work, we use XMI for the same reasons, for exporting the initial model and importing the final model into the CASE-tool. In this paper, like in [28], [30], and [33], requirements consistency checking and validation is not provided.

In [37], the authors present an approach for the representation and management of requirements, representation, and synthesis of system architectures from reusable component specifications, together with the validation and verification of the system architecture based ontologies and reasoning.

The work is concentrated around a software prototype called Paladin that supports component and requirements specification. Paladin is integrated with Ontology-Based Rule Checking, thus resulting in the architecture composed of the following components and functions:

- Paladin, a software prototype for supporting UML diagrams

- Component Assembly based on specific (sub) components, used for building the final element
- System Structure and Behavior that describes the structure and behavior of the system
- Validation of System Architecture based on ontologies and reasoning for validating and verifying the system architecture
- Store Connectivity Information of Objects in RDF
- Import and Export Visual Properties to XML based on the System structure and behavior
- Requirement Template Structure: placeholders that provide input on the values of specific pre-defined requirement attributes
- Merging two Requirement Trees, where different viewpoints need to be joint together
- Requirement Traceability and Controlled Visualization, where requirements link to one another within the same or different levels.
- Collapsing Requirement Tree with Duplicates, thus removing duplicate nodes and revealing the requirements graph structure

To reuse objects and sub-systems, a combination of top-down decomposition mixed with bottom-up synthesis has been employed. The decomposition is achieved by decomposing the higher-level requirements into lower level, more specific and detailed requirements, thus influencing the overall system design. The synthesis takes place by coupling the right components, testing, and verifying the architecture and, in the end, delivering de final product.

To better structure the requirements and to support the bottom-up system development by making the requirements processable, specific templates have been created, such that to adhere to the format of <attribute, relation, value>. Such templates add semantics to each requirement, while allowing the components to be checked against them. Still a downside is the fact that this process of checking component specification against the requirements is not automated.

The internal representation of requirements is based on XML and RDF. By applying XSLT transformations, or by using an XML parser, the authors can generate requirements documents directly from XML. The following are the three files that compose the system requirements document:

- An XML document that stores the visual properties of requirements, namely the way they are drawn in the Paladin tool
- An XML file that stores the internal properties of each requirement
- An RDF file containing the information regarding the connectivity of different requirement objects.

Using Paladin, requirements can be traced to other requirements by means of extraction and visualization, where starting from a particular requirement, the user has the option to view all the complying (lower level) and/or defining (higher level) requirements starting from that particular node. In our approach, by using SysML's Requirements Diagram, we have more flexibility and more options in choosing how a requirement is bounded to another requirement, i.e. by means of traceability, extension, inclusion, dependability, etc.

Furthermore, we can trace requirements into the model, thus specifying which model component satisfies a particular requirement, or which test suite verifies another requirement. In the solution presented by the authors, RDQL is used for a more controlled visualization of the defining and complying requirements.

Another major component of the proposed system is the synthesis of system-level architectures from reusable component specifications. In order for this goal to be achieved, the system architecture has to be defined by components, with a well-defined functionality/semantic exposed through interfaces, and connections, describing the valid interactions between components. To fully describe an Object, we must know what is required and provided by the object, together with its interface that hide the implementation details of the components for an external user. A complete interface specification describes what operations can be performed on the object, the type of data and information that flows through the object, together with the pre- and post-conditions.

In order for a component to be able to be processed, an XML-based schema specification would have to be available. This is a desired outcome not only for this project, but also for our OPP project, where (hardware) components should be made available by manufacturers over the web and can be downloaded and used on the fly.

As mentioned above, RDF is used to specify the relationship between objects and classes in a more general and processable way, providing semantics to the encoded data. Each node and edge defined in an UML

diagram inside Paladin can be represented/stored using an RDF Schema. It is important to notice that RDF requires the presence of a triplet formed by a subject, predicate, and object. For example, if we have node A, a property vcard:N, and an object “A”, the RDF representation would look like this

http://somewhere/A http://www.w3.org/vcard-rdf/3.0#N “A”

Requirement validation is achieved in the following ways: first, check if the requirement is quantifiable and has a logical meaning, problem partially solved by the use of templates in defining requirements; second, check if components are available to satisfy the requirement. In our approach, we verify requirements for completeness and consistency, achieved by means of Prolog rules (more details in the following sections).

To better understand the role of Semantic Web technologies in modeling requirements and defining system architectures, together with the scope of validating their work, the authors have developed a Home Theater System.

The system architecture is first described, showing the components, together with the port and cable specifications. The system requirements are defined next, at three level of abstraction or detail, starting from the agreement between the customer and the designer, and ending with the component requirements. After the textual part has been written, the requirements templates are outlined, partially using the user input. As per the description in their approach, requirements are traced to one another, followed by the merging of different viewpoints of the requirements or system architecture. Duplicate node are

removed from the tree structure, thus exposing the final requirements graph structure.

The component specification, which should be stored somewhere on the vendor's website, together with the ontologies, can be used by reasoning engines for validating the requirements. To exemplify such a methodology, the authors choose an instance of a TV with the specified requirements, and invoke a command on the Paladin toolkit to check the requirement against the component specification file. The result is a dialog presenting different options available to the user, based on how the component satisfies the requirement.
The last step in their example if the validation and verification of the system architecture, based on the use of ontologies and reasoning. The following figure describes the overall schema for their ontology-enabled model checking:

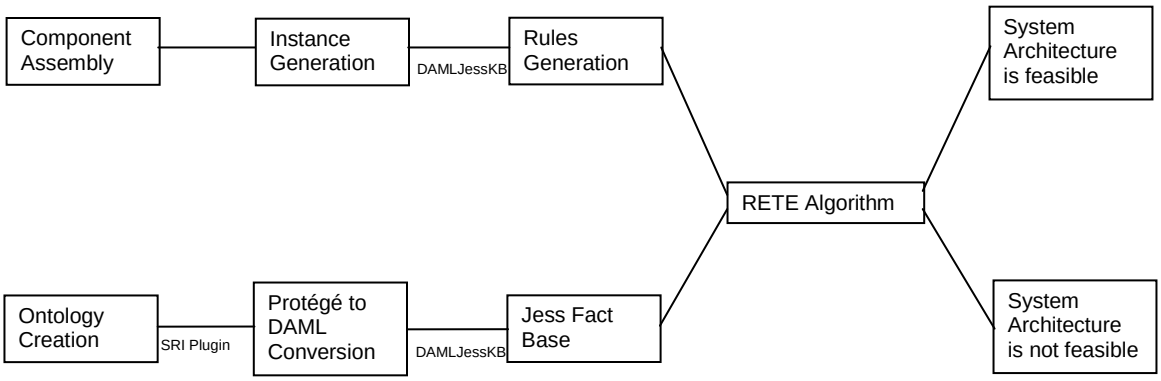

Figure 3.5 Paladin: Model Checking

The above figure shows the Ontology development together with its integration with Paladin to achieve model checking. The component assembly

defined in Paladin is not detailed in the paper, and we are not certain if this is done manually or automatically. Next, the paladin toolkit generates instances (process done manually by the authors) of classes defined in the ontology, together with the connectivity between the ports and jacks in the form of constraints as specified by the user. Based on DAMLJessKB, rules are generated. One point to make here is the fact that this side of the figure has not been implemented at the time of writing the paper. On the right-hand side, classes and relationships are defined using the Protégé. By means of a SRI Plugin, the ontology is conversed into DAML, followed by the creation of the Jess input file that is based on RDF triples provided by way of DAMLJessKB. As a side note, Jess is a rule engine written in Java. Finally, based on the execution of the RETE Algorithm, a pattern-matching algorithm, the feasibility of the system architecture results.

In [38], the authors describe CoSMIC, a tool for synthesizing components by means of formal representation, composition, analysis, and manipulation of system models during the design process. The main components are shown below:

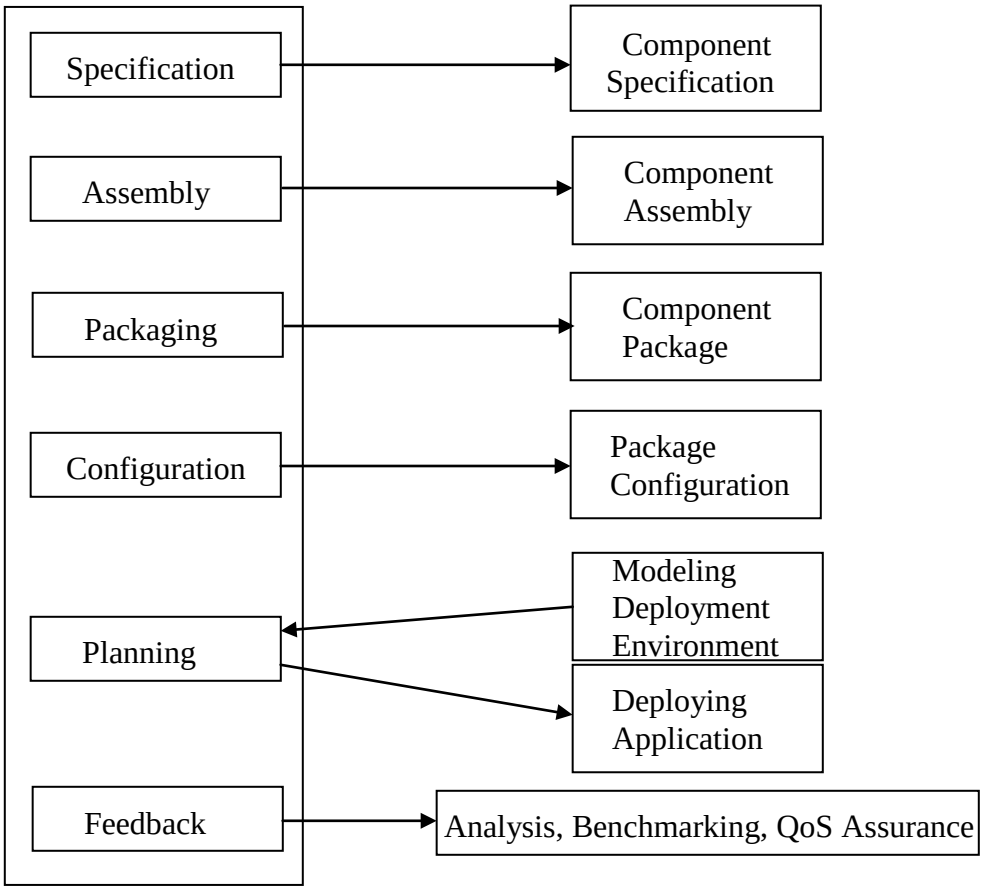

Figure 3.6 CoSMIC Components

For the specification and implementation aspect, the granularity and the implementation options of the components are determined. CoSMIC provides an Interface Definition Modeling Language used to denote component definitions. Different implementations of components can be provided as to adapt for various use cases and environments.

Component assembly and packaging involves the bundling of software modules and metadata representing the components specific to the application. CoSMIC provides a Platform Independent Component Modeling Language,

which models the connections between components to form an assembly, enabling these assemblies to be composed into packages ready to be deployed to specific nodes in the system. To ensure that the packages created are valid, a constraint checker is provided. The constraint language used for the constraints definition on elements in the meta-model is the Object Constraint Language (OCL), a formal, declarative, typed, expression language for describing constraints that apply to UML models.

With the configuration process, packages can be customized with the suitable parameters that fulfill the requirements of the application. CoSMIC provides the following tools to address multi-layer middleware configuration:

- Options Configuration Modeling Language: provides constraints and dependencies for the options used to customize tools and libraries needed by developers.
- Event QoS Aspect Language: provides means of configuring event services; event models can be built and from them, middleware configuration files can be synthesized.
- Federated Event Services Modeling Languages: addresses the federated event channel configuration aspect.

The Planning entity makes decision concerning where the packages should be deployed. The Model Integrated Deployment and Configuration Environment for Compassable Software Systems tool is used to model the deployment plans for the system components.

Another important component of CoSMIC is the Deployment tool that starts the installation of binaries and brings the application to a ready state. The entity that enables the deployment and launching of an application is the Deployment and Configuration Engine runtime infrastructure, inspired by OMG's Deployment and Configuration Specification.

For analyzing and benchmarking, CoSMIC provides the Benchmark Generation Modeling Language that models QoS requirements and produces benchmarking test suites. In this manner, the configuration and deployment of applications based on components can be validated.

To obtain run-time reconfiguration and resource management, CoSMIC employs the Assurance module that uses the J2EE Modeling Language customized to the needs of EJB applications.

In [39], the authors propose a hierarchical interface automaton language for designing software components and applications from the embedded area. Automated analysis techniques are developed stating if components are compatible and compassable. The proposed approach captures the complex and dynamic behavior of component interfaces, allowing a description of more complex behavior. The motivation behind this research is the fact that manual writing and wiring of a large number of components in a language such as nesC (represents the concepts and execution model of TinyOS, an OS for sensor networks) is error prone. The authors try to increase the abstraction level of sensor network programming by also using GRATIS, a visual Domain Specific Language for the nesC component model, where components can be defined

and wired together in a visual environment. Furthermore, GRATIS supports most of the nesC concepts such as Interface (where we define a set of functions, commands, and events), Module (where we implement functions) and Configuration (where we assembly modules). This visual tool is implemented on the Generic Modeling Environment, such that its metamodel defines model elements corresponding to nesC concepts.

As mentioned above, the work is also based on the Interface Automaton, used to represent protocols. The authors add two extensions, namely the hierarchical interface automaton, to improve scalability, besides designing components and applications as mentioned at the beginning, and the non-preemptable states, which do not allow interrupts, and to better suite the concurrency model of TinyOS.

For compatibility checking, Prolog has been utilized to automatically generate logical predicates based on interface specifications. These predicates, together with the formal rules, are combined in the Prolog interpreter, which checks for compatibility between components. Because redundancy still exists, the authors use a constraint language in order to eliminate them. The language used is OCL (Object Constraint Language), which was utilized in [29] also. In this paper, the purpose of OCL was to define semantics and constraints that cannot be captured using a visual notation. Because of the declarative nature of the language, formal rules can be implemented and executed more easily. The result of incorporating OCL version of the composition rule into the meta-model of the hierarchical interface automata domain, composability constraint

violations in every TinyOS configuration models are detected and a proper message is displayed.

For the model verification part, the approach taken by the authors queries the existing modules and generates code based on the interface models. What the user gets is a wrapper around the component one wants to test, something like a black box, generating each signal accepted by the component under test, while at the same time being prepared to receive events coming from the module.

The work done in [40] proposes a UML profile to model through structure and class diagrams non-functional elements in Service Oriented Architecture. The authors make the case that it is important to separate non-functional characteristics of services from functional ones because applications can use those services in different non-functional circumstances. Non-functional requirements are those requirements that describe the quality of certain operations of the system (quality of service, or QoS), while functional requirements describe the behavior of a system. View non-functional requirements as how a system is supposed to be, whereas functional requirements as what a system is supposed to do.

The authors have defined a UML profile by means of stereotypes and tagged-values. Stereotypes are applied to model elements in order to better describe the meanings of model elements based on the domain where they are used. Tagged-values are the data fields specific to a stereotype, consisting of a

name-value pares. The below figure shows an example model of the UML profile proposed by the authors:

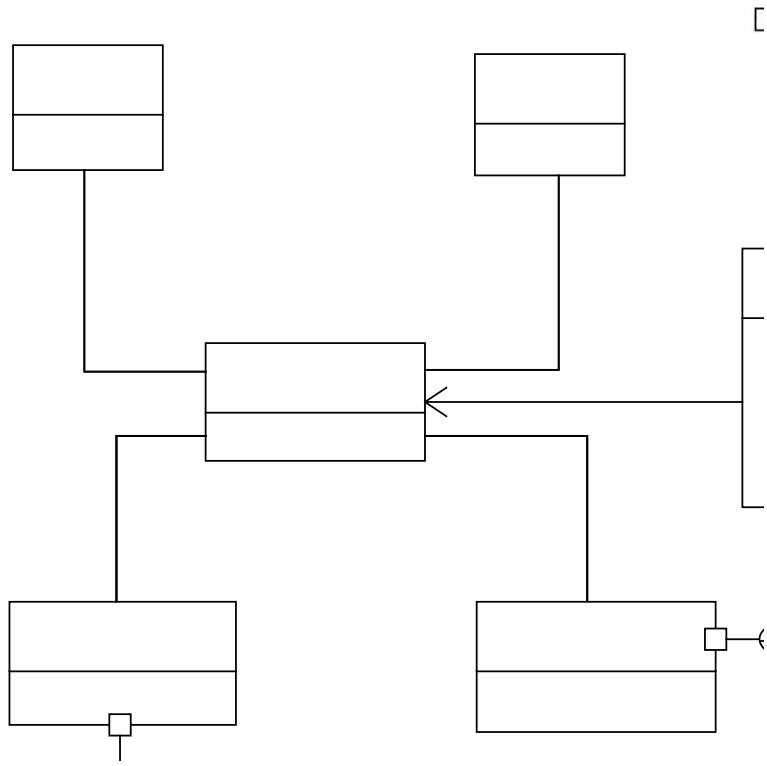

Figure 3.7 Order Processing Application Model

In the example above, there are two services, Buyer and Supplier, of type Service, which exchange messages of type Message that can be either an OrderMsg or an InvoiceMsg. Each request-reply message is represented by an Order of type MessageExchange. Each message is transmitted through the OrderConn of stereotype Connector. Each connector can have multiple filters inside used to customize message processing semantics in a connector. In the

example above, the Login class of type Logger logs OrderMsg and InvoiceMsg. The tagged-values, which are part of this new profile defined by the authors, specify additional semantics for the message processing. In future sections, I will describe what these tagged-values are.

The stereotypes in the proposed UML profile are as follows:

- Service: denotes a network services
- MessageExchange: denotes a request-reply pair of messages, and also denotes which services send and receive those messages
- Connector: denotes a connection between services, describes the meaning of message processing, and also which messages to transmit
- Filter: used for customizing the semantics of message processing and transmission

The stereotypes are defined by extending the UML metamodel.

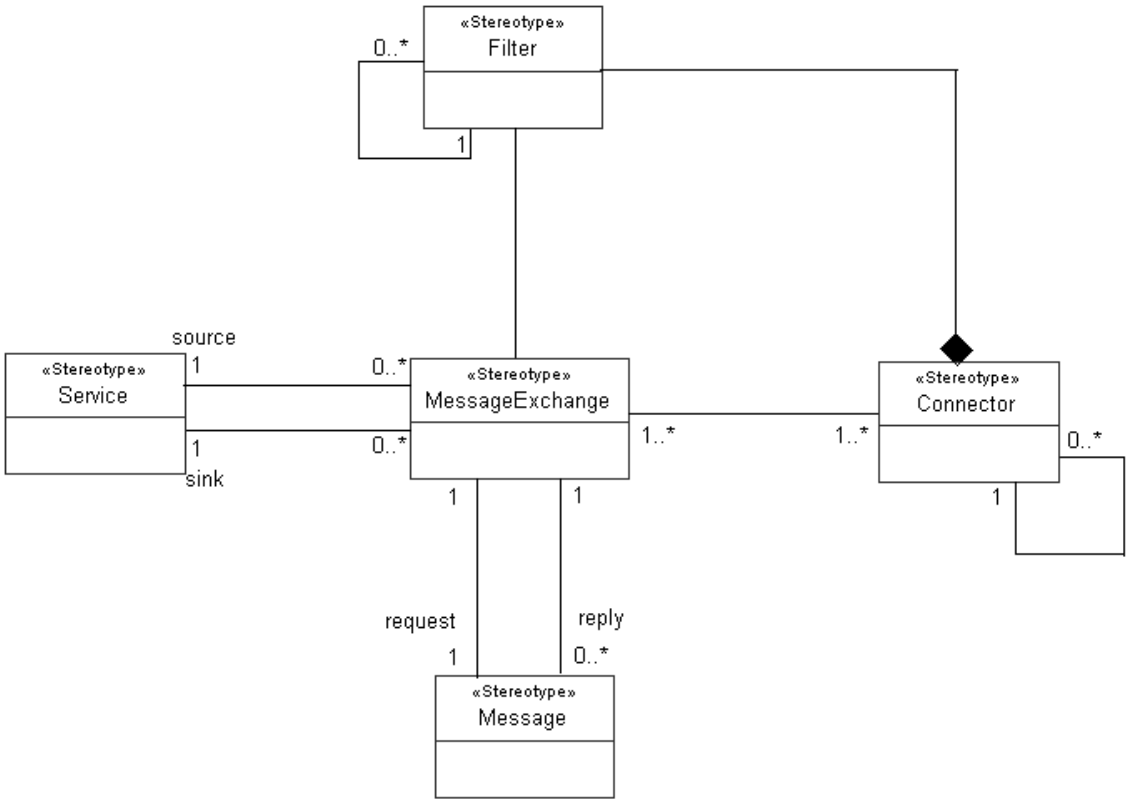

Figure 3.8 Stereotype Diagram

The Filter, Service, and MessageExchange stereotypes extend the Class metaclass in Kernel. A Service, being either a source or a sink, can have zero to many MessageExchange instances, namely can have zero to many request-replies pairs of messages. Each of these messages can have one request and zero to many replies, both being of the stereotype Message. Connector has a one to many relationships with MessageExchange, and it can contain zero to many Filters that specify the semantics of message processing and transmission. Connector extends the Class metaclass in the InternalStructures package of the UML metamodel.

Connector has five tagged-values:

- timeout, which is mandatory, of type Time, and describes the timeout of message transmission
- synchrony, which is mandatory, of type Synchrony(Enum), and describes the synchrony of message transmissions between services (can be Sync, Async, Oneway).
- inOrder, which is mandatory, of type Boolean, and describes the automatic ordering of messages between services
- deliveryAssurance, which is optional, of type DeliveryAssurance(Enum), and describes the assurance level of message delivery (can be AtMostOnce, AtLeastOnce, ExactlyOnce).
- queueParameters, which is optional, of type QueueParameters, and describes the queuing semantics of message transmission (can have size, flushWhenFull, discardPolicy and orderingPolicy based on FIFO, LIFO, PriorityBased, and DeadlineFirst, etc).

The UML Profile defines seven filters:

- Multicast, receives a message and transmits it to multiple destinations simultaneously.
- Manycast, forwards a request message to a specific group of destinations.
- Anycast, forwards a (request) message only to one destination in a group of replicated services.
- Logger, records the transmission of messages.

- Router, routes an incoming message to one or many destinations.
- Validator, validates a message based on a message schema defined in its tagged-value schema.
- MessageFilter, filter out messages based on some particular criteria.

Service has three optional tagged-values:

- priority, which indicates the priority of each message that a service issues.
- timeout, which indicates the timeout period of each message that a service issues.
- redundancy, which indicates the number of runtime instances of a service.

There are three types of Services: MessageConverter that converts an incoming message with a given rule, MessageSplitter that splits incoming messages into fragments with a certain rule, and MessageAggregator that combines multiple incoming messages into a single message.

Message has one mandatory tagged-value and two optional ones, given in the following order:

- schemaURI, identifies the schema of a message, and is of type String.
- priority, specifies the priority of a message, and is of type int.
- timeout, specifies the timeout of a message, and is of type Time.

The application development with the proposed UML Profile involves a MDD tool named Ark, takes as input a UML model designed using the specified

UML Profile (model specified in the XMI format), and transforms it into (Java) code.

The work in [41] proposes a technique in which a specification is derived from a requirement. The author's approach is based on the problem frame from artificial intelligence; in the paper, a problem frame defines the nature of a problem by depicting connections and properties of the parts of the environment that the problem is related to. During this requirement progression process, a trail of the domain assumptions (called *breadcrumbs*) are obtained, which serve as justification of the progression and which, together with the new requirements that result, can give us the path to the reasoning that lead to the specification. The analyst has to come up with the breadcrumbs, which is not an easy and straightforward task, since it requires domain expertise. In addition, their solution works best if the requirements are expressed in a formal language.

In [42], the authors provide means of analyzing requirements by checking for inconsistency and incompleteness; the quality of the specification and change prediction are also enabled through the requirements analysis methodology proposed. Using domain ontology, , which comprises of a thesaurus consisting of domain specific concepts and relationships, and inference rules together with an interpretation function, the authors achieve a semantic processing of the requirements.

Raven [43] is a requirements authoring and validation environment that takes as input text-based requirements and automatically generates three different diagram views of those requirements. These diagrams can be exported

to UML, targeting popular modeling tools such as IBM Rational Software Modeler and other. Through the diagrams, the tool highlights logical or structural potential problems, such as incomplete decision points, flow breaks etc. Test cases can be created and exported from requirements, and requirements specification documents can be automatically generated by the tool.

The goal of SoftWiki [44] is the acquisition and management of requirements using semantic web techniques. The architecture of SoftWiki is shown below:

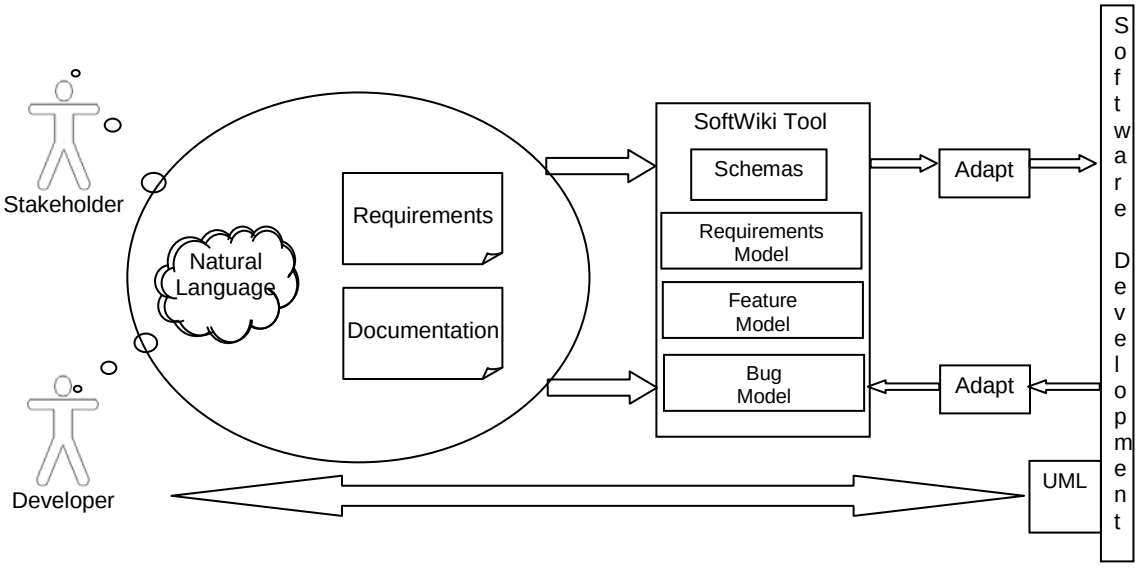

Figure 3.9 SoftWiki Architecture

The main objective is to support the semantic structure of requirements, achieved fitting into semantic schemes and ontologies, and providing semantic patterns and taxonomies. The requirement elicitation and structuring aspect is supported by the moderation of requirements evocation, analyzing of textual

requirements, and providing feedback and review. What SoftWiki has achieved is to combine concepts related to community content management, such as Wikis and web logs, with method of the semantic structure and knowledge representation.

[45] illustrates how natural language requirements are mapped through MDA principals, such as transformation of the Platform Independent Model to the Platform Specific Model, using a formal system of rules expressed in a Two-level Grammar (TLG). Using this approach, requirements can be evolved from domain knowledge to the actual realization of components. A natural language requirement document is first converted into XML, which is next parsed using natural language processing in order to build a knowledge base. Based on domain specific knowledge and by removing contextual dependencies, the knowledge base is converted into TLG. We can think of the TLG as being a bridge between the (informal) knowledge base and the formal specification language representation. The final step is the translation of the TLG code into VDM++, which is an extension of Vienna Development Method that supports the modeling of object-oriented and concurrent systems. The VDM++ representation can be converted into UML or into object-oriented languages such as Java or C++.

Similar to [44], the authors in [46] put forward means of taking advantage of Wikis to gain semantic knowledge of different information. In [46], from existing Wiki content template instances, semantic information is extracted and converted into RDF. The extraction algorithm operates in several stages; first,

Wikipedia pages that contain templates are selected; next, only those templates are extracted that have a high probability of containing structured information. Each template obtained is parsed and RDF triples are generated; URI references or literal values are generated by processing object values. The last step is determining for the processed Wikipedia page its class membership.

The wok in [47] proposes natural language processing by extracting terms from the text written by the domain expert, by means of extracting subjects and objects, and using the predicates to classify them. Next step is clustering terms according to grammatical contexts they are used in. The main clusters are built by subjects or objects of the same predicate. If an overlap occurs, clusters are marked as being related and are joined. Last stage is finding associations between those extracted terms.

The author in [48] discusses some techniques that improve the development process by looking more carefully in how we should interpret and manage requirements. Eliciting requirements is concerned with interpreting the requirements correctly, which implies that during the development life cycle, requirements are identified, defined, and clarified. Planning the requirements management activities is important, and making sure that the project team follows it is not an easy task. Interacting with the customer and active stakeholder participation role are also mentioned in the paper, and these are common agile methodologies.

Twelve requirements basics for project success are discussed in [49]. Some of them mentioned there are training project participants about the

requirements processes, be proactive with your customer, utilize an incremental project approach, properly manage requirements changes, address requirement-related risks, etc. This article helps project teams improve the requirements practices of your project.

Maybe the most comprehensive work that has been done towards reasoning based on rules and semantics, is the REWERSE network [50]. REWERSE is used for software development of Web systems and applications, namely to develop reasoning languages for the Web. The network is divided into various working groups. Since our work involves composing system architectures, the group that is related to us is the Composition and Typing group [51]. Some of the technologies developed by the group are the Reuseware Composition Framework [52] and the Xcerpt typing system XcerptT [53].

The goal of Reuseware is to offer composition techniques and technologies to formal languages that do not have such mechanisms, and it is aimed towards languages involved in the Semantic Web, such as the OWL language and the UML modeling language used in our own framework. The beauty of Reuseware is that it is language independent in the sense that any language can be tailored to support composition and reuse. The architecture of the framework is shown below:

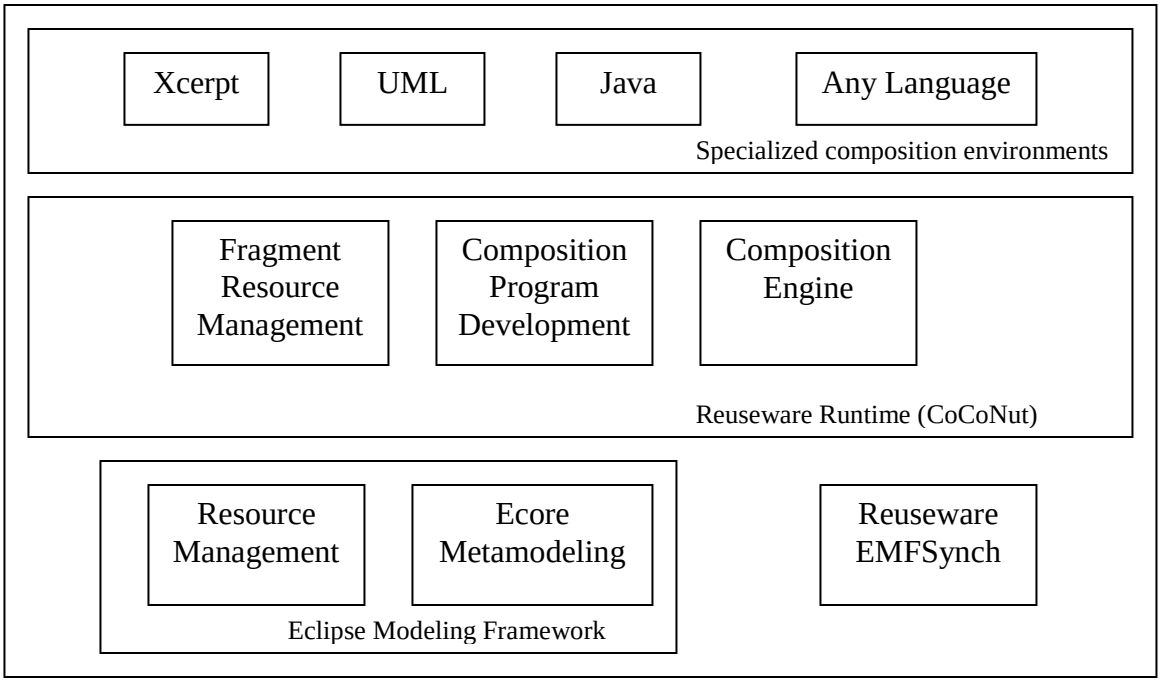

Figure 3.10 Reuseware Composition Framework Architecture

Using the Reuseware SDK, different composition systems can be defined and ran on the CoCoNut Reuseware Runtime framework. The Composition Engine executes composition programs and can generate views on a composed system or be applied as a pre-processor in a tool chain. CoCoNut extends Eclipse Modeling Framework's resource management with a fragment resource management system that can find fragments in the repositories and which is aware of composition interfaces. EMFSynch is used to synchronize components and their occurrences in different composed artifacts.

XcerptT is a type system for the XML query language Xcerpt, where as Xcerpt is an untyped query and transformation language for XML. A Type is a set of data terms or an XML document. XcerptT is written in the Haskell

functional programming language; the type specifications are given as Type Definitions or DTDs. Type definitions have a similar use as schema languages for XML, but they work at a higher level of abstraction so that to better handle the different types defined by different schema languages.

XcerptT makes possible for checking type correctness and type inference. Type checking answers the question is a program written correct given type of database and type of program results specified by the user. For Type inference, based on the type of database, the type of program results can be inferred. Hence we can find out for example if the result type is empty or if it is as expected.

In [54], the authors created a system for software component classification and retrieval. The first part, the encoding, a set of 15 component characteristics are binary encoded to a length of 60 bits. In the second phase, a genetic algorithm discovers several different classifiers, thus attempting to find groups of components with similar characteristics. Determining if a component belongs to a classifier is done based on a threshold, and it's expressed in percentage, denoting how many of the characteristics of the classifier have to be matched with the characteristics of the component. When searching for a component, the matching of user requirements is done with regards to the classifiers, rather than the components. In the third and last step, a component (or rather a set of components will be returned) has to be retrieved based on the specific values for the component's characteristics, and based on the chosen matching threshold value.

It is important to note that the system proposed returns all components that correspond to a classifier. There is no optimization in terms of specific characteristics, i.e. from the end set of components, choose the one that has the lowest cost and the lowest processor utilization (something that has been addressed by the RDAA Framework).

The authors in [55] demonstrate that the complexity of component selection is NP-complete, where the components must be used without any modifications. If this assumption is relaxed, and components can be altered, then the complexity can vary from polynomial to NP, to exponential.

A similar approach to our own work has been taken in [56], where the problem of component selection has been studied, together with two approaches for solving this problem, namely Simulating Annealing and Greedy. Each component is characterized by a set of values such as cost of acquisition, user desirability, development time. A weight is associated for each of those components, combining the desirability and expected revenue values into w_i. The cost of acquisition and the development time are combined into a single cost value c_i. Finally, the value of the item is x_i, where i is an index of the components. The problem of selecting a subset of components that maximizes the total sum of weights, while minimizing the total cost, is recognized to be a multiobjective optimization problem.

Solving such an optimization problem can be done by aggregating the multiple objectives into one scalar objective (an approach that our own component selection algorithm has taken). Another method is to bound one

objective while optimizing the other, and this is the approach taken by the authors.

The simulating annealing algorithm shows the best performance in the simulation results, followed by the greedy algorithm and the manual expert ranking (where a group of experts choose the final components).

In [57], the authors propose an XML-based specification stencil for commercial-of-the-shelf (COTS) components, that would integrate both functional and non-functional aspects of a component, together with the applicability (environment use), standards they conform to, and related components that offer similar services. Component manufacturers adhering to such a specification would greatly simplify the process of reasoning and searching in a component-based design.

The scope of the work in [58] is to reduce the complexity of the component selection process. The authors analyze the Component Selection process by examining approximation solutions that give a practicability to the component selection process. Two variations are proposed; the first one in which the components exhibit uniform emergence; the second one in which pair-wise composition exhibits emergence. On each of the two variations, a Greedy algorithm is applied for the selection process.

The authors in [59] asses a comprehensive evaluation value for each component in a component library, value based on uniforming and standardizing the initial value of the component so as to diminish the impact of large-scale and

small-scale data, together with different scales and units. The resulting value will be between 0 and 1. The equation is shown below:

$$x^{*}_{ij} = (x_{ij} - m_j) / (M_j - m_j), \qquad m_j = min\{x_{ij}\}, M_j = max\{x_{ij}\}.$$

A similar normalization function has been employed in our optimization problem. To this scaled value, a weight is multiplied, weight based on the ration of importance between components. In our own work, the weight is given by the user in form of specifying which QoS parameters have the greatest importance to the user. After this process has been completed, each component is arranged in subsets in an order from great to small, thus achieving the goal of component evaluation.

A component selection technique based on requirements dependencies on component attributes has been proposed in [60]. The component selection based on approximation has two phases; in the first stage, functional, implementation, interface, and performance requirements are considered, in order to reduce the search space; in the second stage, a Greedy algorithm for selecting components is applied. In the Greedy stage, two approaches for extracting indirect dependencies between system performance and components characteristics have been proposed. The first approach is based on non-linear regression analysis, where the aim is to obtain approximations of how well a combinations of components fits the performance requirements. In the second approach, decision trees and conditional probabilities have been employed, the aim being that of ordering the components based on their probability of satisfying different performance requirements.

Another approach that takes into consideration dependencies between requirements is proposed in [61]. The authors make use of semantic web technologies to represent the components and to solve the component selection problem. The main steps of their solution are shown below:

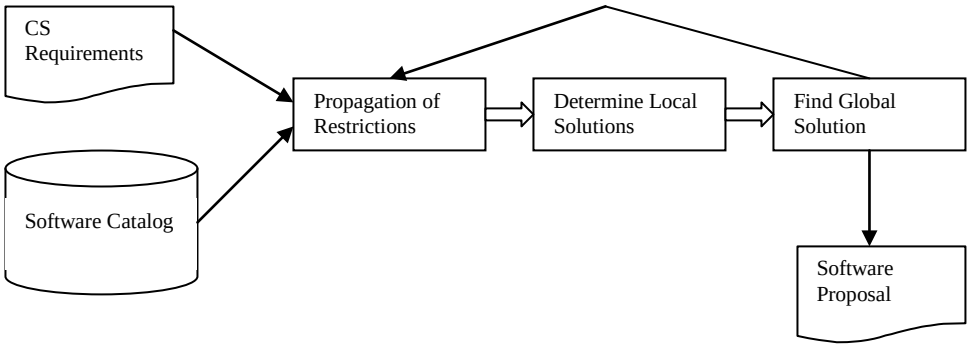

Figure 3.11 Semantic Technologies - Component Selection Steps

In the first step, the propagation of restrictions, the algorithm is based on composition constraints, and thus can only be applied on required functionalities that are in a dependency relationship. In the second step, the determination of local solutions, the software catalog is queried with a query generated from the required functionality. Since the catalog is realized as an RDF repository with OWL descriptions, SPARQL queries are generated. The third and last step involves determining the global solution, and it is achieved by recursively constructing a global solution through incrementally adding one candidate form each set of local solutions.

We have proposed a similar semantic approach: an ontology has been designed to describe components (and requirements), ontology which is loaded

into a Prolog knowledge base, and on which an algorithm was designed (in Prolog) to find the optimal set of components that satisfy the initial requirements.

In [62], the authors propose a Common Criteria based component selection process, thus adhering to the standard for security specification and evaluation of functional and non-functional properties of components. The proposed process includes the following steps: high-level design of the system specifying its architecture and development platforms; creation of a Security Target (ST) document specifying the security requirements for the components; initial component search; evaluation of the components found in the previous step; select the best matching component based on the criteria specified in the ST document; integrate that component into the system.

Chapter 4

METHODOLOGY AND ARCHITECTURE

4.1 Introduction

In the system development cycle, an iteration of a model-driven development process begins with requirements specification, where functional requirements and technical specifications are described by product managers and marketing in natural language, in a semiformal format, such as Marketing Requirements Documents, Unified Modeling Language, or System Modeling Language. After the requirements specification stage, engineers and system architects create a software/hardware architecture design that must fulfill all the initial requirements and satisfy any existing constraints. This design phase includes mapping product features, Quality of Service (QoS) constraints and behaviors to a component-based architecture. This product decomposition process involves QoS translation (from product to sub-product to component level), matching requirements/constraints to components, and component configuration. The implementation and deployment stages follow thereafter.

The transition from requirements to an architecture design is largely done manually with the help of modeling tools, such as Model Driven Architecture

(MDA) UML editors. In most cases designers have available a considerable volume of pre-existing components and frameworks, from earlier projects or from third parties. The problem of finding a configuration from libraries holding hundreds of components, and matching all requirements and constraints without any conflicts, could take considerable time and manpower, especially when requirements change frequently or what-if exploration and tradeoff analyses are performed.

This thesis presents a framework for Requirements-Driven Design Automation (RDDA) [64] that aims to reduce the cost of system design by partially automating the process of architecture decomposition from requirements to existing library components. The framework can be applied to design of software and systems that use UML or SysML.

The key to the proposed approach is to close the semantic gap between requirements, components and architecture by using compatible semantic models for describing both product requirements and component capabilities, including constraints. A domain-specific representation language is designed that spans the application domain (mobile applications, in our case), the software design domain (UML/SysML meta-schema) and the component domains. This language is used to represent ontologies, which are textual representations that capture the semantics of concepts common to product requirements and design modeling, and the relationships between them. The ontology (metamodel) for requirements specifications is based on the Semantic Web Ontology Web Language (OWL). It covers concepts for

product requirements (features and structure), and a set of constraint checking rules. These rules permit consistency and completeness validation for requirements models. before their use in architecture design. The RDDA ontology is expanded to cover knowledge representation for system architecture (UML and SysML diagrams) and for components (semantic annotations). In addition to specifications for product/subsystem features and capabilities, the RDDA ontology supports Quality of Service and system resource constraints, such as energy, CPU load, bandwidth, weight, and volume.

Design automation is supported for architecture development by component selection and design structure synthesis. Automated selection of components from libraries is useful when the number of candidate components is large or when there are many constraints that have to be met, including dependencies. Criteria for component selection include interfaces required and provided, implementation platform, capabilities and constraints that have to be satisfied. Selection criteria that cannot be represented in UML/SysML is described as OWL annotation metadata. Components that match the requirements and provide the necessary interfaces are pulled from the library to populate a structural UML diagram.

Synthesis of design structure diagrams takes the process further by producing new diagrams that can be edited by the user. It works bottom-up from existing structural models describing design relationships and derives feasible configurations represented as UML structural diagrams that satisfy

the requirements. The RDDA framework currently supports functional requirements expressed as required capabilities and constraints, such as symbolic elements (e.g. “supports GPS localization”) and numeric elements (e.g. “maximum latency is 10 s” or “cost between $10 and $20”). The ontology representation also supports requirements tracing to design artifacts. This function is generally used by modeling software with requirements management capabilities to track design and implementation artifacts that are affected by changes to requirements.

The RDDA framework is designed to integrate with UML/SysML modeling tools compatible with OMG’s XML Metadata Interchange (XMI) format. Design models are translated to and from OWL specifications using Extensible Stylesheet Language Transformations (XSLT). This approach bypasses the limitations of the MDA’s Queries/Views/-Transformations (QVT), such as XMI-only conversion. At the core of the RDDA architecture is a reasoning framework built on top of Prolog.

4.2 RDDA Methodology

The need for a different methodology approach is best seen when describing the shortcomings of the current development methodology, shown below:

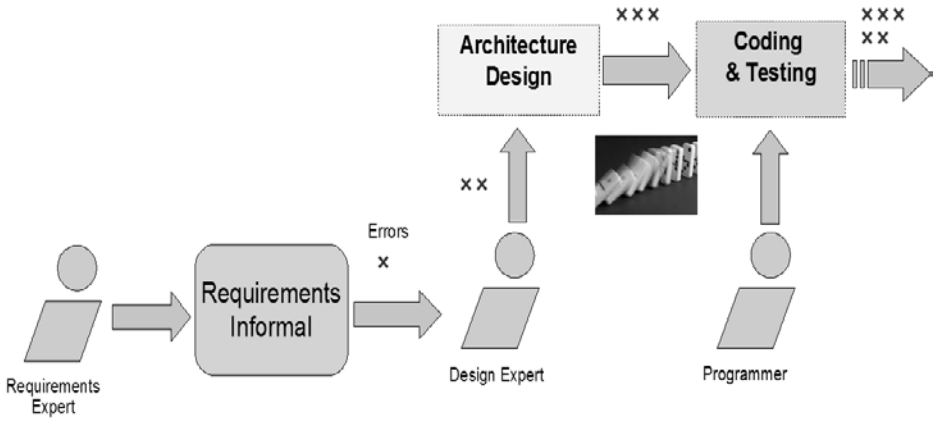

Figure 4.1 Current Development Methodology

The problem with the above approach is that requirements errors/changes permeate through all development stages, resulting in a domino effect. Requirements changes that stem from specification repairs or specification updates for a new product or new version, have to be manually processed, which involves a high-risk, and high-cost. What-if analysis requires manual updates to the model, while subsystem, component selection, and configuration is time-consuming due to the large design space. The vision for this framework is to mitigate all of these negative aspects:

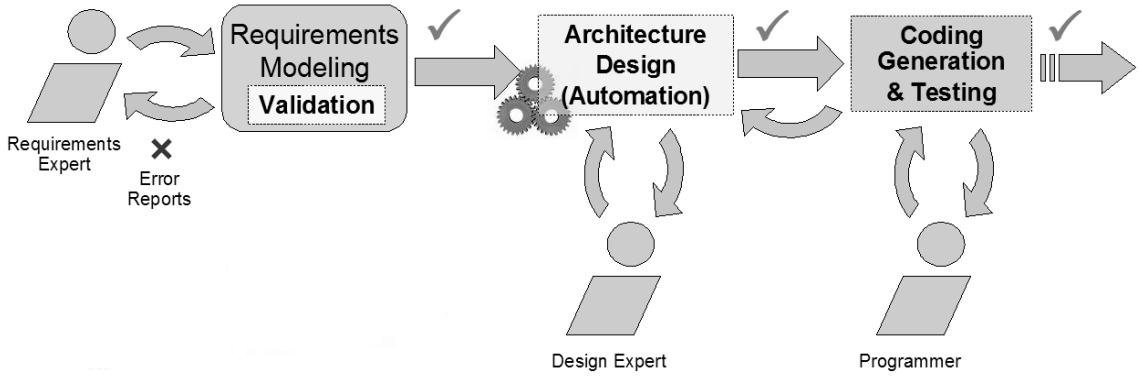

Figure 4.2 RDDA Vision

By building a requirements model, the framework eliminates ambiguities, makes that model formally verifiable, and most importantly, machine processable.

At the core of the requirements-driven design automation framework is the *OPP Design Language* (ODL), an OWL-based language that defines an ontology (taxonomy and properties) for describing semantic models for product requirements, component capabilities/constraints, and design artifacts (UML). In this context, ontologies encode domain specific concepts and the relationships between them in a machine-processable format. An OWL format brings several advantages for knowledge representation in our project, from which the most relevant are that: a) it has a standardized XML based encoding with wide tool support, b) the OWL DL dialect is supported by many reasoning engines and is decidable, c) OWL supports importing of other documents from the web or from a file system, and d) the XML encoding of OWL ontologies facilitates model transformation with XSLT. In our project we target the development of mobile systems and applications. As the full scope of these domains is exceedingly large for the scope of this research, we develop prototype ontologies with limited scope that address a particular case study (an application for location-based services) and we provide a methodology for refining and extending ontologies by qualified personnel. The RDDA methodology is illustrated in Figure 4.3. The input to the Requirements-Driven Design Automation methodology consists of:

(a) requirement models specified in the formal ODL language capturing functional aspects, such as features, constraints and QoS.

(b) component and SysML classifier semantic annotation encoded in ODL describing capabilities and constraints, and

(c) SysML models encoded in XMI from which additional knowledge and relationships between components and classifiers are automatically extracted (e.g. dependencies, interfaces, associations).

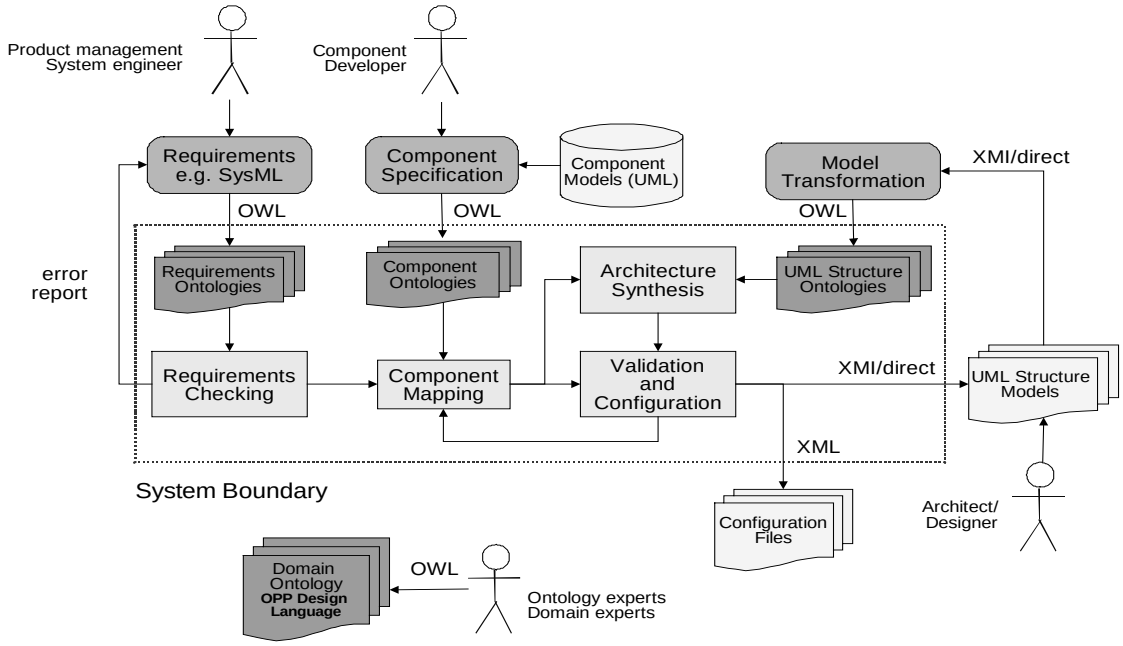

Figure 4.3 RDDA Methodology

The RDDA methodology produces:

(a) Partial UML structural diagrams

(b) component configuration files (XML)

(c) reports on requirements consistency.

The main elements of the Requirements-Driven Design Automation methodology are outlined next, followed by a more detailed description in the upcoming sections.

The OPP Design Language Definition (ODL) is defined initially by application domain experts and ontology experts. The ODL language consists of a series of OWL ontologies that covers the domains of application requirements, component capabilities and constraints, and system design (UML). Details follow in a later section.

Requirements specification is achieved through a requirements model that must be expressed in ODL. The ODL requirements taxonomy defines terms for describing features, capabilities and various constraints. Currently, a requirement specification document must be hand-written or developed with an OWL modeling tool, such as Protégé. As this is tedious and requires OWL expertise, we investigate alternative methods that are more user-friendly, such as using SysML/UML2.0 modeling tools, such as Rational Rhapsody from IBM, and exporting to an XMI format. Requirements specifications in UML/SysML can be translated from XMI to an ODL ontology using XSLT model transformation process, described later. Another approach is to use modelers that process natural language requirements documents. Such a tool should produce semantic descriptions for features, constraints and QoS.

Component specification is necessary in a mature stage of the product line development process, where a stable population of software/hardware components is maintained. SysML components are stored in their native SysML

file format or in XMI. For legacy software/hardware components, wrapper SysML models can be built. Components are tagged with metadata in ODL that describe their semantics in terms of capabilities and constraints that cannot be captured in SysML. For new components that are either developed in-house or acquired from

third-parties throughout the design process, such metadata descriptions must be edited.

The design modeling step involves the system architect and software designers. Users analyze requirements and create software design models that satisfy them. The RDDA methodology assists this modeling step by helping with selecting components from libraries that satisfy the requirements, by synthesizing new UML structure diagrams and by creating feasible component configurations.

The functional components of the RDDA framework are outlined next.

First we have requirements checking. This component validates consistency of the requirements specified in ODL. It checks for any contradictions and verifies whether elements are missing from an existing specification. Validated requirement ontologies and component specifications expressed in ODL statements are compiled to facts in a knowledge base (KB).

Mapping components to the initial requirements is done in several steps. Component specifications in ODL are loaded in a knowledge base (KB) and passed to a reasoning engine that attempts to answer queries for matching

components with requirements. The result from this step are sets of feasible components and configurations.

Architecture synthesis involves automatic analysis of design models in form of system design ontologies compiled from UML diagrams. Based on requirements – features, QoS and constraints – an automated rule-based reasoning mechanism generates new instances and properties that “fill in” the new and updated models such that they satisfy the requirements. When this is not entirely possible, the system will indicate the user the requirements that cannot be satisfied and the model components that are involved.

The validation and configuration step starts by checking the newly assembled design structures and the selected components for consistency. Conflicting designs and component selections are reported to the user, marked and resubmitted to the Component Mapping process. Once the architecture designs are *feasible*, component parameters are derived and saved to runtime configuration files, if required.

4.3 RDDA Architecture

The high-level architecture of the RDDA framework is shown in Figure 4.4 below:

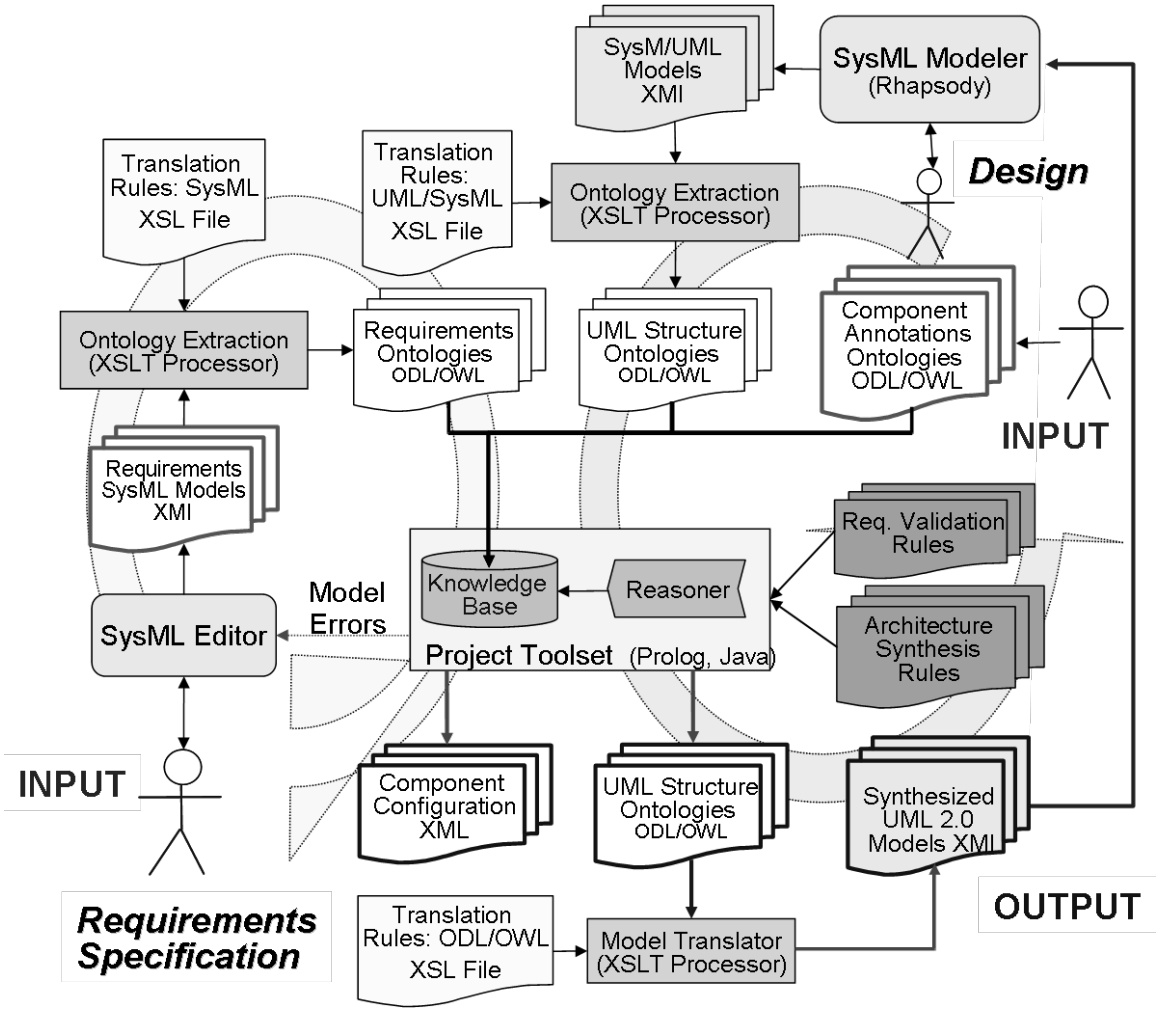

Figure 4.4 RDDA Architecture

The RDDA framework from Figure 4.4 provides roundtrip engineering through two workflows, the *requirements workflow* and the *design workflow*.

4.3.1 Requirements workflow

The *requirements workflow* begins with requirements specialists describing product requirements using a modeling tool, in our case a SysML editor. SysML supports several diagram types for describing requirements,

including a Requirements Diagram, where textual requirements statements and their inter-relationships are represented visually. External and Block Definition Diagrams are used to describe the high-level product hierarchical structure. Behavior diagrams inherited from UML 2 can be used to specify subsystem behavior. For this project, we designed a method for users to add semantic descriptions inside SysML for functional requirements in SysML models specifying system resource and QoS constraints, as well as functional capabilities and features. The requirements specification method is further detailed in section 5.1.

From the SysML editor, the diagram models are exported to XMI and then transformed by an XSLT translator to OWL ODL ontology files. The ODL requirements are then loaded into a Prolog knowledge base for processing. We use the freely available SWI-Prolog environment. The knowledge base is populated with a set of rules for validating the requirements models, searching for completeness and consistency errors. Errors that are identified are highlighted to the user who can go back and repair the requirements specification models inside SysML, closing the workflow cycle. The requirements validation method is described in detail in section 5.3.

4.3.2 Design workflow

The *design workflow* implements a methodology for automated architecture synthesis and component selection based on validated requirements models and semantic component specifications.

The system or software design engineer creates and maintains SysML and UML models using a modeling tool, such as Rhapsody, or an Eclipse-based UML plug-in. The design models include structural diagrams that are supported by the RDDA framework, such as block diagrams, class diagrams, package and component diagrams. The user describes structural models for classifiers (i.e. classes, blocks, interfaces, components, ports). Users also specify semantic annotations for these classifiers, in the same ODL format used for requirements, describing features/capabilities, QoS and resource constraints. These semantic annotations, together with the classifiers, and the intrinsic relationships embedded in the structural model diagrams, form the combined semantic specification of the system architecture. A main function of the RDDA framework is to synthesize structural models. For this, the user builds placeholder UML or SysML components with the desired ODL attributes matching the requirements models. The framework synthesizes an internal composite structure for the placeholder models that satisfies the requirements.

The structural UML/SysML model diagrams are exported to XMI format and then transformed with an XSLT processor to OWL ODL ontology files. These ODL files are loaded to the Prolog knowledge base. The next phase involves system structure model verification by the Prolog reasoner using a set of rules applied to the facts and relationships just loaded to the knowledge base. These rules find consistency errors related to the structural models and related to the semantic annotations (QoS and constraints). Error reports are indicated to the user, who can fix the models and their annotations and trigger a reload to

the knowledge base. Error reporting can be integrated with the modeling tool, and this will be part of our future work. Error feedback is one path closing the *design workflow* cycle back to the user. After structure model verification, the knowledge base contains valid structure models and requirements models. A set of rules perform structure model synthesis, generating composite class and component diagrams for the placeholder components such that the requirements are satisfied without any conflicts. Structure models are converted from Prolog clauses to OWL ODL statements that are further converted by an XSLT translator to XMI code. The XMI code for the generated structure models is merged with the original XMI models exported from the SysML/UML modeling tool and loaded back into the SysML/UML modeling tool for the user to work on. Thus, the second path from the *design workflow* is closed.

4.3.3 Model translation

The model translation phase works conceptually the same for both requirements and component transformations. The overall view of the translation process is captured below:

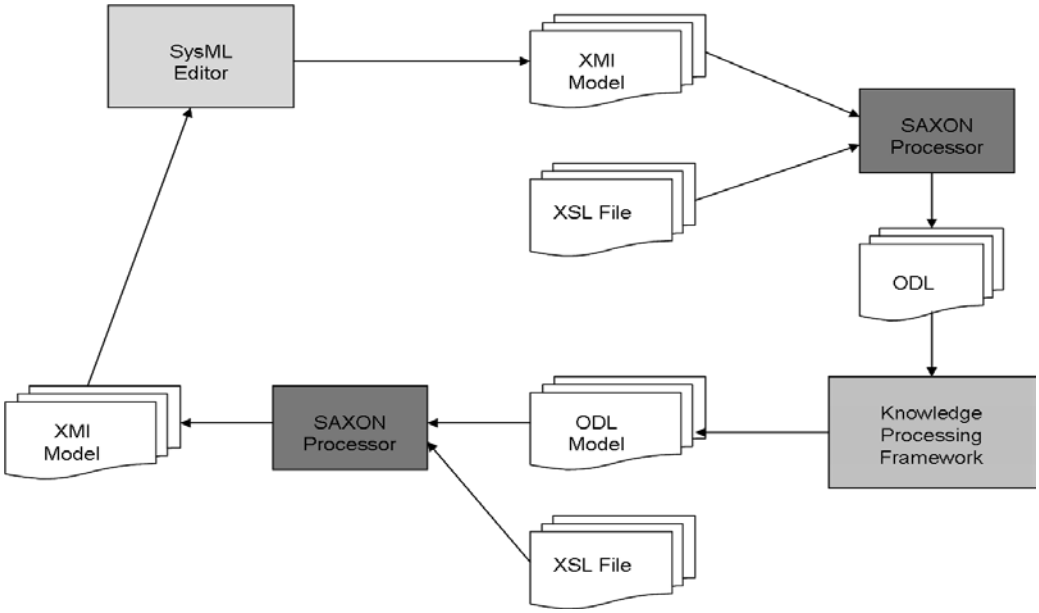

Figure 4.5 Model translation

UML and SysML models are exported to XMI portable specification format from a modeling tool, such as Rhapsody. The XMI format is based on XML and is converted to ODL to be compatible with the RDDA knowledge processing framework. The output from RDDA is converted back to XMI to be loaded into the UML/SysML modeling tool for further modeling use. For conversion we use the Extensible Stylesheet Language (XSL), an XML language for transforming and formatting XML-based documents standardized by the World Wide Web Consortium. Relevant parts of XSL that we need are XSL Transformations (XSLT) for converting XML documents and the XML Path Language (XPath), for navigating in XML documents.

There are several benefits for using XSLT for model transformation to/from ODL. XSLT is compatible with the OWL XML format and is data driven,

not code driven. We only have to specify the transformation rules with matching patterns and the generated format, as opposed to actively traversing the XML document tree with DOM or SAX code. XSLT has a well defined XML data model that requires all compliant XSLT engines to parse XML in exactly the same way.

Chapter 5

REQUIREMENTS SPECIFICATION AND VALIDATION

This section begins with a description of the ODL ontology for requirements model, and the model specification in SysML, continues with the translation rules that take the model form SysML and transforms it into OWL, and ends with the methodology for requirements verification [65].

5.1 Requirements Ontology

The OWL ontology that describes the ODL vocabulary defines a taxonomy (OWL classes) and relationships between instances (OWL properties). A part of the ODL OWL class hierarchy used for requirements is shown in Figure 5.1:

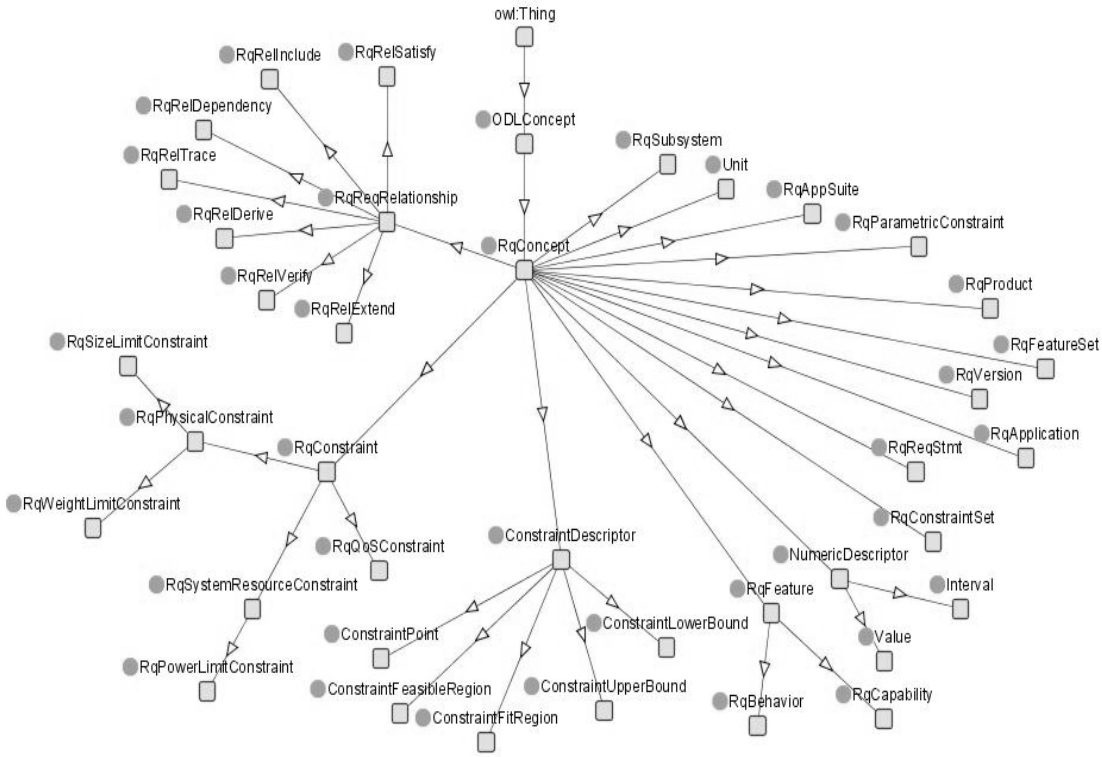

Figure 5.1 ODL Requirements Ontology

The requirements ontology describes concept and properties for

1) product decomposition into applications and subsystems
2) hierarchy of features (functional and behavioral)
3) constraints (system resource, physical, QoS)
4) requirement statement management (versioning, tracking, dependencies).

The ODL metamodel ontologies are developed with Protégé. The ODL OWL metamodel is extensible and can pull in third-party ontologies through the OWL import feature.

The main concepts relating to requirements specification are listed below in Table 5.1:

Table 5.1 Concepts from the requirements ontology

Concept	*Purpose*
RqConcept	Top-level class for all requirements ontology concepts.
RqProduct	The top-level system that is the subject of these requirements (system, application or a component).
RqApplication	Application that runs on the product.
RqSubsystem	A subsystem of the product design hierarchy.
RqFeature	A feature that is required/provided by a product part.
RqCapability	Product capability supported by a product part. E.g. location service. Subclass of RqFeature.
RqConstraint	A generic constraint that applies to a feature.
RqQoSConstraint	QoS constraint that applies to a feature. E.g. localization delay. Subclass of RqConstraint.
RqPhysical Constraint	A physical constraint. E.g. volume, cost, weight. Subclass of RqConstraint.
RqSystemResource Constraint	A system resource constraint. E.g. memory, power. Subclass of RqConstraint.
Constraint Descriptor	Describes a numeric constraint on a feature. Subtypes include upper/lower bounds, feasible

	regions, points, and numeric inclusions.
RqReqStmt	Represents the natural language text for one requirement statement.
RqVersion	Represents a requirements model version number.
RqReqRelationship	Various relationships between requirements statements used for model management. E.g. dependency, inclusion, derivation, tracing.

The requirements ontology supports several types of numeric constraints that apply to features, capabilities, and resources. We describe one constraint with an example from the requirements model of an LBS application:

```
<!-- Subsystem Description -->
<odl:RqReqStmt rdf:ID="Default.Subsystem=Phone has a Camera">
        <odl:rqTracks>
                <odl:RqProduct rdf:about="#Phone">
                <odl:hasSubsystem>
                                <odl:RqSubsystem rdf:about="#Camera"/>
                </odl:hasSubsystem>
                </odl:RqProduct>
         </odl:rqTracks>
         <odl:reqStmtText rdf:datatype="&xsd;string">
                 Subsystem=Phone has a Camera Phone
        </odl:reqStmtText>
</odl:RqReqStmt>
```

The above requirement starts with assigning a unique ID to the requirement statement, followed by specifying the subject of the requirement,

namely the Phone product. It then indicates that the product has a Camera subsystem. The constraints description ends with a textual representation of the requirement, described in natural language, but compliant to a specific grammar. The constraint on the text is that it should follow the pattern of “Type = Subject Predicate Object”, where Type can be System, Subsystem, Application, Feature, Capability, Dependence, or a type of constraint such as ConstraintUpperBound, ConstraintLowerBound, ConstraintFeasible, ConstraintFit, or ConstraintPoint.. The Subject would be Phone, the Predicate would be “has a”, and finally the Object being “Camera Phone”. When describing a feature for the system, the predicate has to specify if that specific requirement feature is provided or required. Such a semi-structured representation is useful when the processing of requirements takes place, which will be detailed in the following sections.

5.2 Requirements Modeling

For creating a requirement model, SysML is used within the Rhapsody modeling tool. The language provides requirements diagrams specifically for such use case.

The first requirements diagram is shown in Figure 5.2 below:

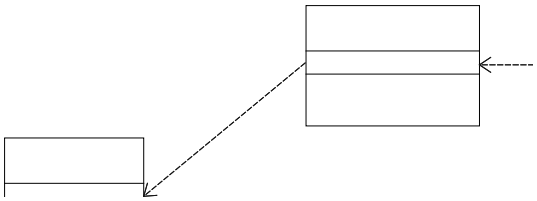

Figure 5.2 LBS Requirements Diagram – Required features and (sub)systems – part 1

Requirement 1.1 is at the highest level, where we specify that the system/product we are trying to build is a Phone. Form this, two additional requirements are derived, requesting that the Phone provide the GPS, and Camera subsystems. The GPS subsystem, or component, requires the location service feature to be present, while the Camera requires the availability of the GIF image format. In addition, requirement 1.3 is extended by two other requirements: 1.10 that states that the feature fLocationService depends on the constraint QoSCost, and 1.4, which states that GPS provides a cost of 200USD. Similar, the Camera component provides a cost of 100USD. These two cost

requirements are going to be checked against requirement 1.9 that states that the phone requires a cost of 300USD.

The second requirements diagram is shown in Figure 5.3 below:

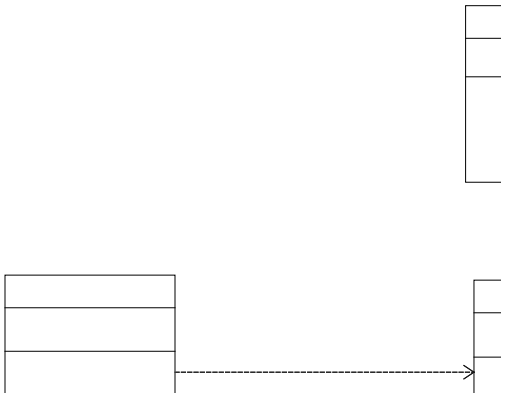

Figure 5.3 LBS Requirements Diagram – Required features and (sub)systems – part 2

The diagram centers around the GPS component. The requirements state that the GPS

- requires an internal memory between 0KB and 128KB
- requires a power consumption between 0microA and 500microA
- and requires a location error between 0m and 5m.

Furthermore, several dependencies are declared, such as:

- the location service depends on QoSInternalMemory
- the location service depends on QoSPowerConsumption
- and the location service depends on QoSLocationError

These six requirements tie with the constraint that a GPS requires the location service feature.

Since the requirements from figure 5.2 and 5.3 are the ones that mostly mention what each component needs, another set of requirements mention what the application offers:

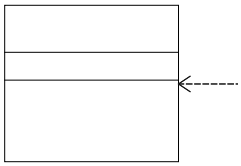

Figure 5.4 LBS Requirements Diagram – Provided features and (sub)system

The set of requirements above start with the Phone product providing an LBS Application that will (try to) satisfy the requirements of the product itself. Since the LBS Application depends on the GPS, the provided features and constraints will be matched against the required ones later in the requirements validation part. The above diagram states that the LBS Application provides:

- Cost of 200USD
- Location Error between 0m and 4m
- Internal Memory between 0KB and 100KB
- and Power Consumption between 0microA and 450 microA

5.3 Requirements Translation

Representing requirements using SysML Requirement Diagrams is the first stage in the process of requirements specification and validation. The nest phase involves translating these requirements into OWL format, that will be then loaded into the Prolog KB. In order for the translation process to take place, the requirements model is exported using the Rhapsody modeling tool into XMI. This process is done directly within the tool, by choosing the “Export” menu item. Below is part of the resulted XMI file that describes requirement 1.3 (“Feature=GPS requires Location Service”):

<UML:Comment xmi.id="_57" xmi.uuid="GUID 5c450cfb-c5e1-44d4-b2a4-f84920bd4f28" name="Feature=GPS requires Location Service" visibility="public" body="Feature=GPS requires Location Service" stereotype="_974" presentation="_56 _121" supplierDependency="_676 _677 _680 _682 _683 _684 _685

```
_687" clientDependency="_840 _713"
namespace="_824">
        <UML:ModelElement.taggedValue>
            <UML:TaggedValue xmi.id="_1008" tag="documentation"
                value="Feature=GPS requires Location
Service" modelElement="_57"/>
        </UML:ModelElement.taggedValue>
    </UML:Comment>
```

The equivalent of a RqReqStmt is the <UML:Comment> tag. The actual textual representation of the requirement can be found in several places, such as the value for the attributes name, body, and value.

Once the requirement model is exported into XMI, translation rules are written into a XSL stylesheet, by using XSLT and XPath to traverse the XMI document and apply specific transformations that will result with an OWL file. The template sought is anything that matches UML:Comment:

```
<xsl:apply-templates
select="//UML:Model/UML:Namespace.ownedElement/UML:Package/UML:Namespace.
ownedElement/UML:Comment"/>
```

Once such a node has been found, the processing starts. For example, to find the main Product that is being described, the following code is applied:

```
<!-- SYSTEM -->
<xsl:if test="starts-with($req, 'Product')">
        <xsl:text>&#xa;</xsl:text>
        <xsl:text>  </xsl:text>
        <xsl:comment>
```

```
                <xsl:text> Product Description </xsl:text>
            </xsl:comment>
            <xsl:text>&#xa;</xsl:text>
            <xsl:text>   </xsl:text>
            <odl:RqReqStmt rdf:ID="{../../@name}.{@name}">
                <odl:rqTracks>
                    <odl:RqProduct rdf:about="#{substring-after($req, '=')}"/>
                </odl:rqTracks>

                <odl:reqStmtText rdf:datatype="~xsd;string">
                    <xsl:text>&#xa;</xsl:text>
                    <xsl:text>        </xsl:text>
                    <xsl:value-of select="string-join((@name,$req), ' ')"/>
                    <xsl:text>&#xa;</xsl:text>
                    <xsl:text>     </xsl:text>
                </odl:reqStmtText>
            </odl:RqReqStmt>
        </xsl:if>
```

The above code checks to see if the requirement starts with the characters ‘Product’ (remember that each requirement in a requirements diagram is preceded by “Product=…”, or “Subsystem=…”, or “Feature=”…”). This first set of characters denote what type of requirement we have, what specific constraint it describes. The code does several things: it creates a unique ID based on the package name and the actual requirement text; next, it specifies the name of the actual product, followed by the actual textual representation of the requirement. What results is shown below:

```
<!-- Product Description -->
<odl:RqReqStmt rdf:ID="Default.Product=Phone">
    <odl:rqTracks>
        <odl:RqProduct rdf:about="#Phone"/>
    </odl:rqTracks>
    <odl:reqStmtText rdf:datatype="&xsd;string">
        Product=Phone
    </odl:reqStmtText>
</odl:RqReqStmt>
```

For a requirement of type constraint feasible region, where a specific interval of possible values has to be met, we have the following XMI representation:

```
<UML:Comment xmi.id="_129" xmi.uuid="GUID 1114a9cb-7b69-4212-
                9d1a-c09fd8891d06" name="ConstraintFeasibleRegion=GPS
                    requires Power Consumption between 0microA and
500microA. "
                visibility="public" body="ConstraintFeasibleRegion=GPS requires
                    Power Consumption between 0microA and 500microA. "
                stereotype="_974" presentation="_128" clientDependency="_843
                    _683" namespace="_824">
    <UML:ModelElement.taggedValue>
        <UML:TaggedValue xmi.id="_1015" tag="documentation"
                                value="ConstraintFeasibleRegion=GPS
requires                                Power Consumption between
0microA and                                500microA. "
                            modelElement="_129"/>
    </UML:ModelElement.taggedValue>
```

```
</UML:Comment>
```

The above requirement states that the GPS component requires a Power Consumption between 0 micro ampere and 500 micro ampere. As before, the name, body and value attributes contain the textual representation of the requirement. To translate this requirement into owl, the following transformations are applied. First, a test is performed to check that the requirement starts with “ConstraintFeasible”:

```
<xsl:if test="starts-with($req, 'ConstraintFeasible')">
```

Next, four variables are created that will hold the actual constraint type (“ConstraintFeasibleRegion”), the textual requirement (“GPS requires Power Consumption between 0microA and 500microA.”), the subject of the triple that forms the requirement (“GPS”), and the remaining property-object values (“Power Consumption between 0microA and 500microA.”):

```
<xsl:variable name="constraint_type" select="substring-before($req, '=')"/>
<xsl:variable name="specific_req" select="substring-after($req, '=')"/>
<xsl:variable name="subject">
    <xsl:choose>
        <xsl:when test="substring-before($specific_req, ' provides')="">
            <xsl:value-of select="replace(substring-
                before($specific_req, ' requires'), ' ',
'')"/>
        </xsl:when>
    <xsl:otherwise>
        <xsl:value-of select="replace(substring-before($specific_req, '
            provides'), ' ', '')"/>
```

```
            </xsl:otherwise>
            </xsl:choose>
        </xsl:variable>
        <xsl:variable name="temp">
            <xsl:choose>
                <xsl:when test="substring-after($specific_req, ' provides')="">
                    <xsl:value-of select="substring-after($specific_req, '
                        requires')"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="substring-after($specific_req, '
                        provides')"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:variable>
```

After the actual requirement has been extracted, the constraint (“Power Consumption”), min (0microA) and max (500microA) values are also stored inside variables for further processing:

```
        <xsl:variable name="constraint"
            select="replace(substring-before($temp, ' between'), ' ', '')"/>
        <xsl:variable name="values" select="substring-after($specific_req, 'between ')"/>
        <xsl:variable name="min_value" select="substring-before($values, ' and')"/>
        <xsl:variable name="max_value" select="substring-after($values, 'and ')"/>
```

How the output will be displayed is determined next:

```
        <odl:RqReqStmt rdf:ID="{../../@name}.{@name}">
            <odl:rqTracks>
                <xsl:element name="odl:{$constraint_type}" rdf:ID="{@xmi.id}">
```

<xsl:if test="substring-after($specific_req, '
provides')=''">
<odl:reqOrProvided
rdf:resource="~odl;isRequired"/>
</xsl:if>
<xsl:if test="substring-after($specific_req,
'requires')=''">
<odl:reqOrProvided
rdf:resource="~odl;isProvided"/>
</xsl:if>
<odl:hasSubject rdf:resource="#{$subject}"/>
<odl:hasConstraintObject rdf:resource="#QoS{$constraint}"/>
<odl:hasNumericDescriptor>
<odl:Interval>
<odl:minValue>
<odl:Value>
<odl:numericValue
rdf:datatype="~xsd;double">
<xsl:text>
</xsl:text>
<xsl:value-of select="translate($min_value,
translate($min_value, '0123456789',
''), '')"/>
<xsl:text>
</xsl:text>
</odl:numericValue>
<odl:unit

```
                                        rdf:resource="~odl;u_{translate($min_value,
                                                '0123456789. ', '')}" />
                                    </odl:Value>
                                </odl:minValue>
                                <odl:maxValue>
                                    <odl:Value>
                                        <odl:numericValue rdf:datatype="~xsd;double">
                                            <xsl:text>&#xa;</xsl:text>
                                            <xsl:value-of
                                                select="translate($max_value,
                                                        translate($max_value,  '0123456789',
'), ')"/>
                                            <xsl:text>&#xa;</xsl:text>
                                        </odl:numericValue>
                                        <odl:unit
                                            rdf:resource="~odl;u_{translate($max_value,
                                                '0123456789. ', '')}" />
                                    </odl:Value>
                                </odl:maxValue>
                            </odl:Interval>
                        </odl:hasNumericDescriptor>
                    </xsl:element>
                </odl:rqTracks>

                <odl:reqStmtText rdf:datatype="~xsd;string">
                    <xsl:text>&#xa;</xsl:text>
                    <xsl:text>        </xsl:text>
                    <xsl:value-of select="string-join((@name,$specific_req), ' ')"/>
```

```
                    <xsl:text>&#xa;</xsl:text>
                    <xsl:text>      </xsl:text>
                </odl:reqStmtText>
            </odl:RqReqStmt>
        </xsl:if>
```

The output OWL file that results after the above transformation rules have been applied to the initial XMI file is shown below:

```
        <!-- Feasible Region Constraint Description -->
        <odl:RqReqStmt rdf:ID="Default.ConstraintFeasibleRegion=GPS requires Power
                Consumption between 0microA and 500microA. ">
            <odl:rqTracks>
                <odl:ConstraintFeasibleRegion>
                    <odl:reqOrProvided rdf:resource="&odl;isRequired"/>
                    <odl:hasSubject rdf:resource="#GPS"/>
                    <odl:hasConstraintObject
                            rdf:resource="#QoSPowerConsumption"/>
                    <odl:hasNumericDescriptor>
                      <odl:Interval>
                        <odl:minValue>
                          <odl:Value>
                            <odl:numericValue rdf:datatype="&xsd;double">
                              0
                            </odl:numericValue>
                            <odl:unit rdf:resource="&odl;u_microA"/>
                          </odl:Value>
                        </odl:minValue>
                        <odl:maxValue>
```

<odl:Value>
<odl:numericValue rdf:datatype="&xsd;double">
500
</odl:numericValue>
<odl:unit rdf:resource="&odl;u_microA"/>
</odl:Value>
</odl:maxValue>
</odl:Interval>
</odl:hasNumericDescriptor>
</odl:ConstraintFeasibleRegion>
</odl:rqTracks>
<odl:reqStmtText rdf:datatype="&xsd;string">
ConstraintFeasibleRegion=GPS requires Power Consumption between
0microA and 500microA. GPS requires Power Consumption
between 0microA and 500microA.
</odl:reqStmtText>
</odl:RqReqStmt>

5.4 Requirements Validation

This section describes the process of requirements validation by checking them for completeness and consistency. In order to achieve this, we have built a formal model based on first order predicate logic, where given a set of formulas with certain properties, another specific formula can be derived as a conclusion. As such, given a set of properties derived from the requirements model, certain conclusions about those requirements can be inferred, conclusions that would validate (or invalidate) those requirements.

The validation process starts after the requirements are translated into OWL, and loaded into Prolog's knowledge base. Let us consider a requirements model MR = <P,A,S, F,C,NC,R>, where P is the set of products described, A is the set of applications (A $\subset$ S), S is the set of subsystems, F is the set of features, C is the set of constraints, NC is the set of constraint numeric descriptors, and R is the set of relationships on these sets describing the model.

The requirements model relationships are modeled as a set of predicates listed in Table 5.2. The following notation is used for variables: s $\in$ S subsystems, f $\in$ F features, c $\in$ C constraints, and nd $\in$ N_C numeric constraint descriptors.

Table 5.2 Requirements model predicates

Predicate	Notation
s_1 has subsystem s_2	$u_s(s_1, s_2)$
s_1 has subsystem (transitive) s_2	$u^*_s(s_1, s_2)$
s_1 depends on subsystem s_2 for features	$d_s(s_1, s_2)$
s_1 depends on subsystem (transitive) s_2 for features	$d^*_s(s_1, s_2)$
s_1 depends on subsystem (for features or structurally) s_2	$d^\wedge_s(s_1, s_2) \Leftrightarrow u^*_s(s_1, s_2) \vee d^*_s(s_1, s_2)$
s requires feature *f*	$f_r(s, f)$
s provides feature *f*	$f_p(s, f)$
s depends on feature *f*	$d_{sf}(s, c) \Leftrightarrow f_r(s, f) \vee f_p(s, f)$
f depends on constraint *c*	$d_{fc}(f, c)$

constraint descriptor *cd*	*cd(s, c, nd)*

The verification rules look for conflicts in specification for QoS constraints, such as maximum query delay, and system resource constraints, such as power, CPU load, weight, etc.

5.4.1 Completeness Verification

This is the first step in verifying the correctness of a requirements model. An initial completeness check at the syntax level is done on the OWL source by the Vowlidator OWL checker [63]. The tool examines OWL files for potential errors, and reports them together with the exact location of the errors in those files. For example, if in the OWL file there are references to resources (i.e. “#GPS”) that were actually not yet declared with an ID (i.e. “rdf:Class rdf:ID=GPS”), the Vowlidator will complain and raise an error. Using such a tool is a perfect first step in making sure that requirements are in a state free of syntax errors and other errors such as undefined resource or mismatched namespaces.

In the next step, a set of Prolog rules are applied to the requirements model loaded into the knowledge base, searching for individuals (subsystems, features, constraints) that are referred but not defined. For example, the rule in logical sentence that checks if a declaration for a subsystem is missing is shown below:

$$\forall s \in P \cup S,\ u_S(s, sub) \wedge sub \notin S \Rightarrow missingSubsystem(sub)$$

The equivalent rule expressed in Prolog is:

```
checkComp_Subsys :- (hasIndividual(odl:'RqSubsystem', S) ;
        hasIndividual(odl:'RqProduct', S)),
    odl_hasSubsystem(S, Sub), \+ hasIndividual(odl:'RqSubsystem', Sub),
        printlist(['ERROR:  missing subsystem ', Sub, ' required by ', S, '\n']).
```

The ';' operator stands for logical OR in Prolog and the '\+' operator is logical not. Similar, the rule that checks for a missing dependent is described below:

$\forall s \in P \cup S, d_s(s, o) \wedge o \notin S \Rightarrow missingDependent(o)$

Using Prolog, the above rule translates into the following:

```
checkComp_dependsOn :- odl_dependsOn(S, O), \+ hasIndividual(_C, O),
    printlist(['ERROR:  undefined dependent ', O, ' required by ', S, '\n']).
```

Analogous, there is a second set of rules search for cases where a subsystem s1 requires a feature that is not provided by any subsystems s2 on which s1 is dependent on (transitive and inclusion), first expressed in first order logic:

$\forall s_1 \in P \cup S, f_r(s, f) \wedge \neg(\exists s_2 \in S, \hat{d}_s(s_1, s_2) \wedge f_p(s_2, f)) \Rightarrow$ *missingRequiredFeature(s, f).*

The prolog equivalent:

```
checkComp_featuresMissing :- hasIndividual(odl:'RqApplication', A),
    odl_requiresFeature(A, F), checkComp_featureMissing(A, F), fail.  % continue
        with subsystems

checkComp_featuresMissing :- hasIndividual(odl:'RqApplication', A),
    dependsOnTrans(A, Sub),
```

```
        odl_requiresFeature(Sub, F),
        checkComp_featureMissing(Sub, F).
```

5.4.2 Consistency Verification

Consistency verification involves two types of checks:

- Model structure sanity
- Constraint validation

For the model structure sanity checks, rules were written that verify the requirements model internal structure. One of these checks is for dependency loops (a subsystem depending on itself for features or structurally):

```
checkConsist_dependsCycle :-  checkConsist_dependsCycle(odl:'RqApplication'),
        checkConsist_dependsCycle(odl:'RqSubsystem').
checkConsist_dependsCycle(C) :-  hasIndividual(C, A), checkConsist_dependsCycle(A,
        A, [A]).
checkConsist_dependsCycle(X, Y, L) :- checkConsist_dependsFindCycle(X, Y, L).
checkConsist_dependsFindCycle(X, Y, [_|L]) :- member(Y, L),
        printlist(['ERROR: found dependency cycle in ', X, '\n']),
        reverse([Y|L], LR), print(LR), print('\n').
checkConsist_dependsFindCycle(X, Y, L) :- odl_dependsOn(Y, Z),
        checkConsist_dependsFindCycle(X, Z, [Z|L]).
```

The above rule is equivalent to the following first order logic:

$\exists s \in S, d^{\wedge}_{s}(s, s)$

The second check is for subsystem unique ownership, meaning that a subsystem cannot belong to more than one parent subsystem/product/application:

```
checkConsist_subsysOwn :- hasIndividual(odl:'RqSubsystem', Sub),
        odl_hasSubsystem(Parent1, Sub), odl_hasSubsystem(Parent2, Sub),
        Parent1 @< Parent2,
        printlist(['ERROR: subsystem ', Sub, ' has 2 parents: ', Parent1, ' and ', Parent2,
                '\n']),
        fail.
```

The above rule translates into the following first order logic:

$\forall s_2 \in S, \exists! s_1 \in S \cup P, u_s(s_1, s_2)$

For constraint validation, these rules verify the consistency of numeric constraint descriptors. A constraint descriptor associates a subject subsystem (e.g. GPS subsystem), a constraint object (e.g. localization error), with a numeric descriptor – [min,max] interval or a point value – for a performance metric or system resource indicator. Two constraint descriptors are checked for consistency conflicts if the following rule applies:

Constraint Matching Rule:

Two constraint descriptors are checked for consistency if they refer to the same constraint object, the first subsystem depends on the second (feature-wise or structurally), and the corresponding feature is required by the first subsystem and provided by the second subsystem.

The logical expression for the above rule is as follows:

$cd(s_1, c, nd_1) \wedge cd(s_2, c, nd_2) \wedge \hat{d}_s(s_1, s_2) \wedge d_f(f, c) \wedge f_r(s_1, f) \wedge f_p(s_2, f)$

$\Rightarrow checkConsistency(nd_1, nd_2)$

The exact method for constraint descriptor checking depends on the respective subclasses, as follows:

- ConstraintUpperBound - specifies maximum limit
- ConstraintLowerBound - specifies minimum limit
- ConstraintFeasibleRegion - specifies interval with acceptable values. Provider must cover at least partly the interval.
- ConstraintFitRegion - specifies interval with mandatory values. Provider subsystem must cover the entire interval.
- ConstraintPoint - specifies a single value that must be matched by provider subsystem.

The checking rules for *valid QoS constraints* are listed in Table 5.3. UB stands for upper bound constraint descriptor, LB for lower bound.

Table 5.3 Consistency checking rules for QoS constraints

cd2 / *cd1*	*UB*	*LB*	*Feasible or Fit*	*Point*
UB	$M_1 \geq M_2$	C	$M_1 \geq M_2$	$M_1 \geq v_2$
LB	C	$m_1 \leq m_2$	$m_1 \leq m_2$	$m_1 \leq v_2$
Feasible	$m_1 \leq M_2$	$M_1 \geq m_2$	$m_1 \leq m_2 \leq M_2 \leq M_1$	$m_1 \leq v_2 \leq M_1$
Fit	$M_1 \leq M_2$	$m_1 \geq m_2$	$m_2 \leq m_1 \leq M_1 \leq M_2$	C
Point	C	C	C	$v_1 = v_2$

The above rules indicate valid cases, or conflicts (‘C’). The corresponding numeric descriptor are intervals $[m_i, M_i]$ or point values v_i for i = 1, 2.

As an example, below is the Prolog rule for checking all UB-UB constraint descriptors that involve features required by a subsystem S. The description for each important section of the rule accompanies the code:

```
checkConsist_constraintUpperBound :- hasIndividual(odl:'RqProduct', A),
        checkConsist_constraintUpperBound(A).
checkConsist_constraintUpperBound :- hasIndividual(odl:'RqProduct', A),
        hasIndividual(odl:'RqSubsystem', S), dependsOnTrans(A, S),
        checkConsist_constraintUpperBound(S).
```

The above section will start the checking process by identifying either a product A, or a subsystem S that depends on the product A.

```
checkConsist_constraintUpperBound(S) :-
        odl_requiresFeature(S, F), odl_dependsOn(F, Constraint),
        hasIndividual(odl:'RqConstraint', Constraint), odl_hasSubject(CDReq, S),
        odl_hasConstraintObject(CDReq, Constraint),
                hasIndividual(odl:'ConstraintUpperBound', CDReq),
        odl_hasNumericDescriptor(CDReq, NDReq),
        odl_maxValue(NDReq, ValueReq), odl_numericValue(ValueReq, MaxReq),
```

The above code will start looking for a feature F required by the system S, that depends on a Constraint, who's constraint object CDReq is of type ConstraintUpperBound, and its numeric descriptor NDReq has a maximum value ValueReq denoted by the numeric value MaxReq.

```
        odl_hasConstraintObject(CDProv, Constraint), CDProv \== CDReq,
        hasIndividual(odl:'ConstraintUpperBound', CDProv),
        odl_hasSubject(CDProv, SDep), dependsOnTrans(SDep, S),
        provideRequire(SDep,F), odl_hasNumericDescriptor(CDProv, NDProv),
        odl_maxValue(NDProv, ValueProv), odl_numericValue(ValueProv, MaxProv),
```

The same constraint required by feature F has to have a constraint object CDProv of type ConstraintUpperBound that is part of a system SDep who is dependent on the initial system S, and who provides the required feature F with the maximum value of MaxProv. If such a system exists, then the rule will check to see if what is provided is greater than what is required (thus finding a consistency error):

```
MaxProv > MaxReq,
printlist(['ERROR: upper bound constraint not met for ', Constraint, ' on feature ',
                F, ' required by ', S, ', provided by ', SDep, ' (CD=', CDReq,
')\n']),
        assert(upperBoundErr(Constraint, F, S, SDep, CDReq)), fail.
```

The beauty of Prolog lies in its backward-chaining inference capabilities, making it a powerful tool for rule-based query. The rule will keep backtracking until the first conflict is found from all applicable UpperBound constraints descriptors and all required features, or all search possibilities are exhausted. Similar rules have been written for all other QoS constraint consistency rules from Table 5.3.

Chapter 6

ARCHITECTURE SYNTHESIS

Architecture selection during system design is a time consuming activity from the large design space that requires manual checking of existing components, mapping system requirements such as features and constraints to available subsystems or components, and choosing the optimal subset that satisfy the specification. In addition, what-if analysis requires manual updates to the models, thus adding high risk and high cost to it. A way to improve the productivity of architecture design is by implementing an automated component-based methodology for system design that searches for a feasible component-based design within a potentially very large design space. The methodology for design synthesis implemented in the RDDA framework closes the system development cycle in support of an iterative and incremental model-based process. The proposed approach is generic enough to support specification and design for other types of systems; the main changes that would be addressed are in the domain-specific descriptions of requirements, such as features, system constraints, and QoS [66].

6.1 Component Specification

The fist step in the component selection process is to define the initial component diagram using a Block Definition Diagram in SysML:

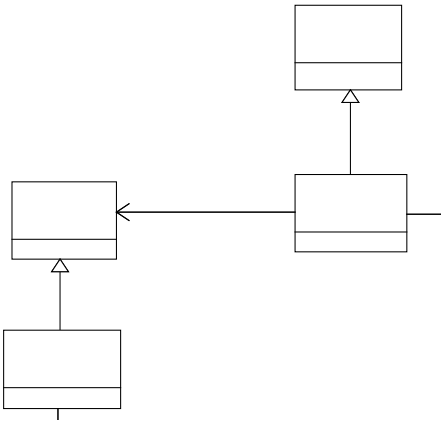

Figure 6.1 Initial Component Diagram

The above diagram identifies three main components: Phone, Camera, and GPS. The Phone components requires three interfaces: iPhone, iCamera, and iGPS. The iCamera interface is provided by the Camera component, while the iGPS interface is provided by the GPS component. The Camera component requires the iImageEncoder interface that is provided by two camera type components: Canon and Olympus. Both Canon and Olympus provide the fGIFImageFormat feature. The GPS component requires the iLocationService

that is provided by the three gps type components: TI, SiRF, and Amtel. TI provides the fPlaceFinder and fLocationService features, SiRF provides the fLocationService and fAltitudeInformation features, while Amtel provides only the fLocationService.

When we capture just the features that are required and provided, the below diagram results:

Figure 6.2 Features Diagram

The required features for both the Camera and GPS components are GIFImageFormat and LocationService. These features are provided by the Canon and Olympus components, and by the TI, SiRF and Amtel ones.

The constraints for each of the three gps-type components (TI, SiRF, and Amtel) are shown in Figure 6.3 below. Each component has three constraints based on location error, internal memory, and power consumption. All constraints are of type feasible region, which means that both a minimum and a maximum are specified. These constraints have to be checked against the required ones that are specified in the initial requirements diagram.

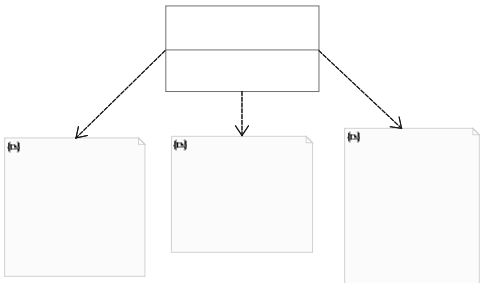

Figure 6.3 GSP QoS Constraints

6.2 Model Transformation and Ontology Extraction

As with the requirements diagrams, the component related model is exported by the Rhapsody SysML editor into XMI. The definition of the Phone component and the relationship with the required iPhone interface is shown below:

```
<UML:Class xmi.id="_6" name="Phone" visibility="public" stereotype="_265"
                    generalization="_30" associationEnd="_876 _882">
</UML:Class>
<UML:Generalization xmi.id="_30" name="iPhone" visibility="public" />
```

The two associations with the iCamera and iGPS interfaces, together with the definition of those interfaces, are represented in the XMI code below:

```
<UML:Interface xmi.id="_8" name="iCamera" visibility="public" associationEnd="_872">
</UML:Interface>
<UML:Association xmi.id="_32" >
        <UML:Association.connection>
                <UML:AssociationEnd xmi.id="_872" name="itsICamera"
                        aggregation="none" association="_32">
                    <UML:AssociationEnd.multiplicity>
                        …
                        <UML:MultiplicityRange xmi.id="_875" lower="1"
                                upper="1"/>
                        …
                    </UML:AssociationEnd.multiplicity>
                </UML:AssociationEnd>
                <UML:AssociationEnd xmi.id="_876” association="_32"/>
        </UML:Association.connection>
</UML:Association>
<UML:Interface xmi.id="_10" name="iGPS" visibility="public" associationEnd="_878">
</UML:Interface>
<UML:Association xmi.id="_34" >
        <UML:Association.connection>
                <UML:AssociationEnd xmi.id="_878" name="itsIGPS"
                        aggregation="none" association="_34">
                    <UML:AssociationEnd.multiplicity>
                        …
                        <UML:MultiplicityRange xmi.id="_881" lower="1"
                                upper="1"/>
                        </UML:Multiplicity.range>
                    </UML:AssociationEnd.multiplicity>
```

```
                    </UML:AssociationEnd>
                    <UML:AssociationEnd xmi.id="_882" aggregation="none"
                                        association="_34" />
                </UML:Association.connection>
            </UML:Association>
```

Next step is to write transformation rules that would convert the XMI document into OWL. For example, to extract the UML:Class related values from XMI, the following rules are applied:

```
<xsl:template match="UML:Package/UML:Namespace.ownedElement/UML:Class">
        <odl:MClass rdf:ID="{@name}">
                <odl:hasName rdf:datatype="~xsd;string">
                        <xsl:value-of select="@name"/>
                </odl:hasName>
                <xsl:if test="key('id_to_name', @stereotype)/@name">
                        <odl:hasStereotype

rdf:resource="~odl;stereotype_{key('id_to_name',
                @stereotype)/@name}"/>
                </xsl:if>
                <odl:inPackage rdf:resource="#{../../@name}"/>
                <xsl:for-each select="UML:Classifier.feature/UML:Operation">
                        <odl:providesFeature>
                                <odl:RqFeature rdf:about="#{@name}"/>
                        </odl:providesFeature>
                </xsl:for-each>
                <odl:hasXmiId rdf:datatype="~xsd;string">
                        <xsl:value-of select="@xmi.id"/>
```

```
                </odl:hasXmiId>
            </odl:MClass>
        </xsl:template>
```

The above code will start by checking for UML:Class in the source tree, followed by creating the output tags based on the information found in the XMI, information such as name of the class, stereotype (if present), package, features provided, and the unique XMI Id. After applying the above transformation rules on the XMI document, the result is as follows:

```
<odl:MClass rdf:ID="Phone">
        <odl:hasName rdf:datatype="&xsd;string">Phone</odl:hasName>
        <odl:hasStereotype rdf:resource="&odl;stereotype_block"/>
        <odl:inPackage rdf:resource="#Default"/>
        <odl:hasXmiId rdf:datatype="&xsd;string">_6</odl:hasXmiId>
</odl:MClass>
```

The OWL representation of a component that provides a certain feature (such as Amtel gps component that provides a place finder a location service feature) is shown below:

```
<odl:MClass rdf:ID="Amtel">
        <odl:hasName rdf:datatype="&xsd;string">Amtel</odl:hasName>
        <odl:hasStereotype rdf:resource="&odl;stereotype_block"/>
        <odl:inPackage rdf:resource="#Default"/>
        <odl:providesFeature>
                <odl:RqFeature rdf:about="#fPlaceFinder"/>
        </odl:providesFeature>
        <odl:providesFeature>
                <odl:RqFeature rdf:about="#fLocationService"/>
        </odl:providesFeature>
        <odl:hasXmiId rdf:datatype="&xsd;string">_28</odl:hasXmiId>
</odl:MClass>
```

The association between the Phone component and the required iCamera interface, that was captured using a Block Definition Diagram, and later exported into XMI, is presented below:

```
<odl:MAssociation rdf:ID="_32">
    <odl:hasRelEnd>
        <odl:MRelationEnd rdf:ID="_872" name="itsICamera">
            <odl:relEndRef rdf:resource="#iCamera"/>
            <odl:relEndMultiplicity rdf:datatype="&xsd;string">
                1
            </odl:relEndMultiplicity>
            <odl:relEndIsNavigable rdf:datatype="&xsd;boolean"> true
            </odl:relEndIsNavigable>
            <odl:relEndIsAggregation rdf:datatype="&xsd;boolean"> false
            </odl:relEndIsAggregation>
        </odl:MRelationEnd>
    </odl:hasRelEnd>
    <odl:hasRelEnd>
        <odl:MRelationEnd rdf:ID="_876" name="">
            <odl:relEndRef rdf:resource="#Phone"/>
            <odl:relEndMultiplicity rdf:datatype="&xsd;string"/>
            <odl:relEndIsNavigable rdf:datatype="&xsd;boolean"> false
            </odl:relEndIsNavigable>
            <odl:relEndIsAggregation rdf:datatype="&xsd;boolean"> false
            </odl:relEndIsAggregation>
        </odl:MRelationEnd>
    </odl:hasRelEnd>
</odl:MAssociation>
```

6.3 Component Mapping

The requirements models, the design models, and the semantic component annotations are all loaded in the KB. All these models constitute a *design graph* of concepts (components, features, constraints) connected by edges representing the relationships between concepts. Nodes can be annotated with numeric constraints (values or intervals). The architecture synthesis problem is a search for a feasible subgraph in the design graph that satisfies all constraints. Such a solution is a point in the search space. The objective for the search is formulated as a subset of nodes and edges (i.e. a subgraph) that must be *covered,* meaning the solution subgraph must include all nodes/edges from the objective set, plus their transitive closure. The transitive relation is one of dependency between ODL concepts, and is defined based on the semantics of the involved concepts. For instance, if component c_1 belongs to the partial solution and c_1 requires feature *f*, then the solution must include a component c_2 that provides feature *f*. The inclusion of numeric constraints in the design space precludes the use of classic graph-theoretic algorithms. The proposed solution relies instead on constraint logic programming (CLP) – logic programming in Prolog with extensions for constraint satisfaction in the body of rules.

Loading the OWL files with the requirements and design models into the Prolog KB populates it with triplets in the form of (S, P, O), or (subject, property, object). Such a representation is inconvenient to work with; for this reason, the triplets are converted to a more condensed representation, namely P(S,O)

Prolog facts. After this transformation, the facts describing the SysML artifacts from Figure 6.1 will be loaded to the Prolog knowledge base. The predicates involved are described next:

- odl_comp(component): defines one component
- odl_compReq(c,i): component c requires (uses) interface i
- odl_compProv(c,i): component c provides interface i
- odl_compReqsFeature(c,f): component c requires feature f
- odl_compProvFeature(c,f): component c provides feature f
- odl_compReqsQoS(c,q,v1,v2): component c requires QoS numeric constraint between v1 and v2 (dependence)
- odl_compProvQoS(c,q,v1,v2): component c provides QoS numeric constraint between v1 and v2.

The features and constraints that were detailed in section 5 are preserved as facts that will be processed by our component selection mechanism. Once all the facts are in the knowledge base, the next step is selecting those components that match the required interfaces, features, and constraints. The problem is one of finding a sub-graph that satisfies all the requirements. The steps taken by our algorithm that builds the component architecture are mentioned below:

1. Find initial components that match the required interface(s).
2. Check if the required features and constraints have been met by the components from the previous step.

3. If there are features or constraints not provided by the components in the solution, check if there are other components that provide those missing features.
4. Return a list S of edges ci(CR, I, CP), where CR is the component that requires interface I that is provided by the component CP. Furthermore, return a list of features cpf(CR, F, CP) that have not been met by components in S, but which are satisfied by components that are not part of S; CR is the component that requires feature F which is provided by component CP that is not part of S. In addition, return a list of features that are not provided by any existing component. Similar lists are returned for constraints.

For step 1, the rules written ensure that the initial interfaces are provided, and in addition, that all interfaces required by every component are added to the solution. These Prolog rules are presented below:

```
%Find the initial components that satisfy all required interfaces.
%X is a list of those components.
%L is a list of required interfaces.
findInitComp(X,L) :- findall(c(dummy,I), member(I,L),P), findInitComp(X,[],P).
findInitComp(S,S,[]).
findInitComp(X,A,[c(CR,I)|T]) :- odl_compProv(CP,I), member(ci(CR,I,CP),A),
                                 findInitComp(X,A,T).
findInitComp(X,A,[c(CR,I)|T]) :- odl_compProv(CP,I), not(member(ci(CR,I,CP),A)),
                                 append([ci(CR,I,CP)],A,S), findall(c(CP,Ir),
```

```
                                        odl_compReq(CP,Ir),R),
add2end(R,T,F),                                  findInitComp(X,S,F).
```

If during the initial search, the algorithm finds a component that provides a particular required interface, but does not satisfy the required features or constraints, the algorithm will backtrack and search for another possible candidate. Each of the four steps contain one or more predicates.

For step 2, the below code checks if the required features and constraints have been met:

```
%Checks to see if the required features/constraints have been met by the component
        %that is part of the solution and for which there is an edge ci(...).
%LC is list of edges ci(CR, I, CP) where all interfaces I required by components CR have
%been provided by components CP;
%R is list of components CP that provide interfaces required;
%MF is list of missing features;
%MC is list of missing constraints.
checkFeat_QoS(LC,R,MF,MC) :- checkFeat_QoS(LC,R,_,_,MF,MC), !.
checkFeat_QoS(LC,L,A,AC,MF,MC) :- findall(crf(C,F), (odl_requiresFeature(C,F),
                                        member(C,L)), LF),
featuresNotMet(LF,L,LC,A),
append(A,AA,MF), findall([C_C,CC,MinV,MaxV],
        odl_compReqsQoS(C_C,CC,MinV,MaxV),LCC),
            constraintsNotMet(LCC,L,LC,AC),
                append(AC,CA,MC).
```

The Prolog findall(+*Template*, :*Goal*, -*Bag*) predicate creates a list of the successful instantiations of *Template* on backtracking over *Goal*, and unifies the result with *Bag*. If there are no solutions in terms of *Goal*, *Bag* is an empty list.

The append(?*List1*,?*List2*,?*List3*) predicate will concatenate *List1* and *List2* into *List3*. The other predicates in the above snippet of code are described next. In featuresNotMet(LF,L,LC,A), LF is a list of features than need to be provided by the components in LC. A is a list with features not provided by the solution components. In constraintsNotMet(LCC,L,LC,AC), LCC is a list of constraints, L a list of components, and AC the final list of constraints not met.

For step 3, we deal with the case of having features and/or constraints not satisfied by the initial components. As such, we search for other components that might satisfy them, and that are currently not part of the initial solution. If we find such components, we add them to the solution list of components. First we check which features/constraints have not been met by the initial components, as shown below:

```
%LF is a list of features that are not provided by the components that are part of the
%solution.
%LC is a list of functors cf(F,CP) where  F is a feature, and CP is the component that
%provides that feature
%(CP is not part of the initial solution, and will be added to the architecture later).
%NoC is a list of features that are not provided by any existing component.
whoProvMissFeat(LF,LC,NoC) :- whoProvMissFeat(LF,[],LC,[],NoC).
whoProvMissFeat([],AC,AC,NoA,NoA) :- !.
whoProvMissFeat([crf(CR,F)|T],AC,LC,NoA,NoC) :- odl_providesFeature(C,F),
                                append([cpf(CR,F,C)],AC,ACN),
                                        whoProvMissFeat(T,ACN,LC,NoA,NoC).
```

```
whoProvMissFeat([crf(CR,F)|T],AC,LC,NoA,NoC) :- not(odl_providesFeature(C,F)),
                append([crf(CR,F)],NoA,NoAN),
whoProvMissFeat(T,AC,LC,NoAN,NoC).
```

A similar predicate exists that checks for which components provide the missing constraints that were not satisfied by the components that resulted in step 1:

```
%Checks the list of components Comp to see if any cumulative constraints are satisfied
%or not. Those who are not, are returned into CumC.
checkCumulativeConstraints(Comp,CumC) :- findall(cr(C,Cons,ConVal),
                (member(C,Comp), odl_compReqsQoS(C,Cons,ConVal)),R),
                        checkCumulativeConstraints(Comp,Comp,R,[],CumC),
!.
checkCumulativeConstraints(_,_,[],S,S) :- !.
checkCumulativeConstraints([],Comp,[cr(C,Cons,ConVal)|L],A,CumC) :-
                append([cr(C,Cons,ConVal)],A,A_A),

checkCumulativeConstraints(Comp,Comp,L,A_A,CumC).
checkCumulativeConstraints([C|T],Comp,[cr(C,Cons,ConVal)|L],A,CumC) :-
                delete(Comp,C,R),                findall(Val,
                        (odl_compProvQoS(C_P,Cons,Val),member(C_P,R)),
Val_L),                 sum(Val_L, #=<, ConVal),

checkCumulativeConstraints([C|T],[C|T],L,A,CumC).
checkCumulativeConstraints([C|T],Comp,[cr(C,Cons,ConVal)|L],A,CumC) :-
                append([cr(C,Cons,ConVal)],A,A_A),
                checkCumulativeConstraints([C|T],[C|T],L,A_A,CumC).
checkCumulativeConstraints([H|T],Comp,[cr(C,Cons,ConVal)|L],A,CumC) :- H \== C,
```

checkCumulativeConstraints(T,Comp,[cr(C,Cons,ConVal)|L],A,CumC).

To better understand the cumulative constraint predicate above, let us look at an example of a of QoS constraint is (gps, qosLocationError, 0, 5), where the gps component requires a location error between 0 and 5. The rule checks to see if there is a component that provides the qosLocationError constraints and for which the min and max values are contained in the required interval. Our algorithm handles additive constraints (e.g. maximum cost allowed) through the following steps: searching for those cumulative constraints and finding all components from the solution that provide those constraints; adding all values for a particular constraint and checking against the required value.

Considering the way this algorithm is currently constructed, it finds all possible solutions, sorted by the solution for which all features and constraints have been satisfied. Hence, if there is such a solution, it is returned first in the list, so the user can take advantage of it. The reason we chose to present solutions to users in this way is because of the assumption that when a new product (or new version of a product) is constructed, there will likely be features/constraints not satisfied by the components used in previous products (or versions). Obtaining a solution that is complete (satisfying all required interfaces/features/ constraints) is unlikely.

6.4 Results

If we look at the initial component diagram in Figure 6.1, there are two choices for the Camera (Canon and Olympus), and three possibilities for a GPS

component (TI, SiRF, and Amtel). Each component comes with its own features and constraints that are checked for conformance to the algorithm described in the previous sections. If we run our algorithm using the query statement buildArch(X,[iPhone]) (meaning that we look for a set of components starting with the one required interface, namely a phone interface), the first solution displayed will be: “[ci(camera, iImageEncoder, canon), ci(gps, iLocationService, ti), ci(phone, iCamera, camera), ci(phone, iGPS, gps), ci(dummy, iPhone, phone)], [], [], [], []”. Another solution, but which does not provide all features and constraints, is “[ci(camera, iImageEncoder, olympus), ci(gps, iLocationService, sirf), ci(phone, iCamera, camera), ci(phone, iGPS, gps), ci(dummy, iPhone, phone)], [cpf(gps, fProximityService, ti), cpf(camera, fGIFImageFormat, canon)], [], [cc(qosLocationError, ti)], []”.

It can be seen that feature fProximityService required by the gps component is not provided by the sirf component that is part of the solution, but is actually provided by the ti component which in this particular case, is not part of the solution. In addition, qosLocationError constraint is not met by the sirf component, which has a maximum location error value of 7, while at most 5 is required. The algorithm will find the component that satisfies this constraint (in our case the ti component) and will include it in the result architecture diagram.

The framework chooses the 'best' solution after the Prolog algorithm generates a list of solutions, with complete ones (if any) presented first in this list. A solution consists of Prolog functors that describe selected components, interfaces and relationships between them. This model is then converted to an

XMI file and merged as a SysML structural diagram with the user's SysML design project. The resulting XMI project file is reloaded by the SysML modeling tool and the designer now can further refine it, or turn it to code generation. Figure 6.4 shows the generated solution diagram for the example used in this chapter consisting of a cell phone with a GPS location-based system:

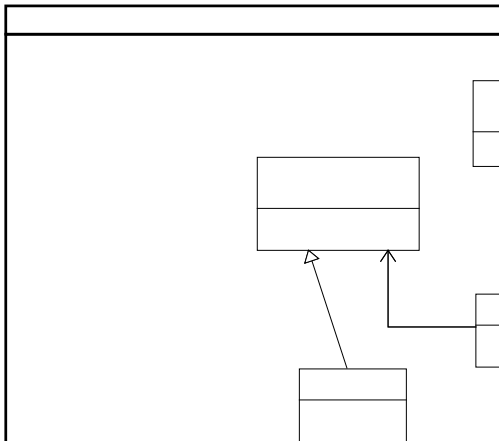

Figure 6.4 Final Component Diagram

Chapter 7

ARCHITECTURE OPTIMIZATION FOR QOS AND RESOURCE UTILIZATION

Since the component selection algorithm devised in Prolog and explained in section 6.4 is finding the best solution in terms of choosing the one that satisfies all features and constraints, but not in terms of timing and QoS/constraints, as it will still search all possible solutions, the RDDA framework also comes with an algorithm written in Prolog that will build the final solution based on optimizing several existing constraints. There are several reasons why such an optimization is required:

- To avoid searching all possible solutions.
- To improve the running time of the component selection and architecture synthesis.
- To find the optimal solution in terms of maximizing or minimizing one or more linear expressions (called objective functions).

The main contribution comes from designing a set of rules for building the optimization problem starting with a block definition diagram that describes the existing components of the system under design. To actual solve the optimization

problem once it was formulated, the simplex algorithm available as a Prolog library is used.

The input to our problem is a list of solutions, where each solution contains a list of connected components that satisfy the initial requirements (in terms of required/provided features and constraints), or, if some features/constraints have not been met, it provides a list of components that would satisfy them, if such components exists. The solutions are not ordered in an optimal way; if there are more than one solution that satisfies all requirements, the order in which they are saved in the list of solutions is non deterministic.

The output should be a list of solutions, where the first solution contains the components that optimally satisfy the QoS constraints.

The following three sub sections will introduce the different existing optimization methodologies, will describe how to build an optimization programming problem starting with the SysML model, and finally will describe the component selection mechanism run through the optimization routine.

7.1 Optimization Methodologies

Optimization deals with choosing the best value from a set of available alternatives, by minimizing or maximizing a function by choosing values from a set of possible values. In subsequent sections we will explore some of the subfields of optimization.

7.1.1 Combinatorial Optimization

Combinatorial optimization is concerned with problems where the set of feasible solutions is discrete or can be reduced to a discrete one. The goal is to find the optimal solution. Given a set I of feasible solutions, and a function $f : I \rightarrow R$, we need to find an element $x \in I$, with $f(x) = \max \{f(y) \mid y \in I\}$.

7.1.2 Linear Programming

A linear program (LP) is a mathematical formulation of a problem. We define a set of decision variables that describe in full the decisions we wish to make. We then use these variables to define an objective function which we wish to minimize or maximize, and a set of constraints which restrict the solution space.

In a linear program, the variables must be continuous and the objective function and constraints must be linear expressions (hence the term linear programming). An expression is linear if it can be expressed in the form $c_1x_1 + c_2x_2 + ... + c_nx_n$ for some constants $c_1, c_2, ... ,c_n$. In defining the variables, we need to ask ourselves what it is that we wish the model to determine.

The object function is a mathematical expression for what we are trying to achieve. It is written in terms of minimizing or maximizing a linear expression that covers the decision variables. A set of constraints will restrict the search space of our variables, with each constraint being a linear expression of our variables, with any appropriate coefficients, followed by the type of restriction and the value of

the right hand side. A great introduction to linear programming can be found at [67].

7.1.3 Integer Programming

Integer programming studies linear programs in which the variables are constrained to take integer values. In many cases, integer programming is considered to be a special case of linear programming where the variables take only integer values, as mentioned before. If not all variables are integers, then the problem is a *mixed integer programming problem* [68].

Some methods of solving integer programs is branch and bound. The initial problem is divided into smaller problems based on restricting the range of the variables such that a variable with a lower bound of *lb* and upper bound *ub* will be divided into two problems with ranges [lb, p] and [p+1, ub]. Through the relaxation of the linear programming problem, lower bounds are obtained such that the objective function and all constraints are unchanged, but the integrality restrictions are relaxed to derive a linear program. If the optimal solution to a relaxed problem is integral, that it is an optimal solution to the sub-problem, and the value can be used to terminate searches of sub-problems whose lower bound is higher, thus pruning the search space.

Another approach is branch and cut. The lower bound is obtained in a similar fashion to branch and bound, namely by the linear-programming relaxation of the integer program. The optimal solution to this linear program is at a corner of a feasible region, which is defined as the region containing the set of

variables that satisfy the existing constraints. If the optimal solution to the linear programming is not integral, this algorithm searches for a constraint that is not violated by any optimal integer solution, but is violated by the current solution. This constraint is called a cutting plane. When this constraint is added to the LP, the old optimal solution is no longer valid, and so the new optimal will be different, potentially providing a better lower bound. Cutting planes are iteratively until either an integral solution is found or until no more cutting planes are found. If no more cutting planes are found, a branch and bound algorithm is applied.

7.1.4 Nonlinear Programming

Nonlinear optimization studies nonlinear programs where the variables are real, and the objective function or constraints are nonlinear [69]:

$$\textit{Minimize (or Maximize)} \quad f(x)$$

$$\textit{subject to } g_i(x) \leq 0 \text{ (or } g_i(x) \geq 0) \text{ for } i = 1, \ldots, m$$

$$h_i(x) = 0 \text{ for } i = 1, \ldots, p$$

where f, g_i, h_i are functions defined on R^n, X is a subset of R^n, and x is a vector of n components $x_1, \ldots, x_n$. The problem has to be solved such that x_i satisfies the constraints (inequality and equality constraints) while the objective function f is minimized (or maximized). A *feasible solution* to the problem is a vector $x \in X$ satisfying all existing constraints. A *feasible region* is formed by the collection of all such solutions. The nonlinear programming problem then is to find a feasible point y such that $f(x) \geq f(y)$, for each feasible point x. Such a point y is called an *optimal solution*.

7.1.5 Multi-Objective Optimization

In a multi-objective optimization problem, we are trying to simultaneously optimize three (conflicting) objectives subject to certain constraints. In a well formed multi-objective problem, there will be more than one solution that simultaneously satisfies each objective to its fullest. In each case, we are searching for a solution for which each objective has been optimized to the extent that if we try to optimize it any further, then the other objectives will suffer as a result. Hence, for the multi-objective problem, it is highly improbable to have a single solution which satisfies every objective simultaneously; therefore, the solution is defined in terms of Pareto optimality in the following sense: a feasible solution to a multi-objective programming problem is *Pareto optimal* if there exists no other feasible solution that will yield an improvement in one objective without causing a degradation in at least one other objective.

7.2 Building an optimization problem

The first step in creating an optimization problem is to extract the necessary information from the SysML block definition and requirements diagrams in order to generate the constraints and the objective function. The constraints are both structural constraints (such as system requiring n components), and QoS constraints (such as power consumption for the GPS component should be below 500 microA). The objective would be to minimize one or more constraints, such as power consumption, location error, etc.

Once the diagrams are translated into constraints, these are then loaded into Prolog's knowledge base as facts and solved by the simplex prolog algorithm provided in the simplex library that comes with SWI-Prolog. Using this library, all numeric quantities are converted to rationals, and rational arithmetic is used throughout to solve the linear programs.

Since everything starts with the SysML model of the system under design, the RDDA framework will extract information form the existing diagrams, and create the set of constraints for the linear programming problem. To the best of our knowledge, such an approach has not been performed yet.

There are four types of relationships between components that the framework deals with: generalization, dependency, composition, and aggregation. Whenever such associations are found, a specific transformation rule is applied that will generate the desired constraint. These relationships are important because they show a logical connection between the different components of the system.

7.2.1 Constraints from Generalization Relationship

Below is an example of a generalization relationship between several model elements in a SysML diagram:

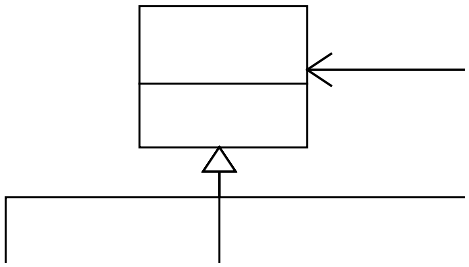

Figure 7.1 Generalization Relationship

The above diagram presents a component C which requires n components that implement the I interface, such as X_1, X_2, …, X_n. To form a constraint that would encapsulate C, I and all n X_i components, the following rule is applied:

$$nC – (X_1 + X_2 + … + X_n) = 0$$

7.2.2 Constraints from Dependency Association

Below is an example of a dependency association between several model elements in a SysML diagram:

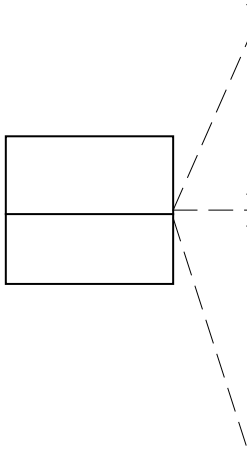

Figure 7.2 Dependency Association

The above diagram presents a component C that depends (functionally or structurally) on n components X_1, X_2, …, X_n. To form a constraint that would encapsulate C, and all n Xi components, the following rule is applied:

$$n(1 - C) + (X_1 + X_2 + \ldots + X_n) \geq n \Leftrightarrow (X_1 + X_2 + \ldots + X_n) - nC \geq 0$$

7.2.3 Constraints from Composition Relationship

Below is an example of a dependency association between several model elements in a SysML diagram:

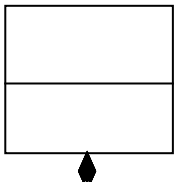

Figure 7.3 Composition Relationship

The above diagram presents a component C that has a whole-part relationship with n components X_1, X_2, …, X_n. To form a constraint that would encapsulate C, and all n Xi components, the following rule is applied:

$nC – (X_1 + X_2 + \ … + X_n) = 0$

7.2.4 Constraints from Aggregation Relationship

Below is an example of a dependency association between several model elements in a SysML diagram:

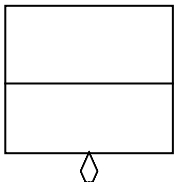

Figure 7.4 Aggregation Relationship

The above diagram presents n components X_1, X_2, …, X_n that are part of the C component. To form a constraint that would encapsulate C, and all n Xi components, the following rule is applied:

$(X_1 + X_2 + \ \ldots + X_n) - nC \geq 0$

7.3 Optimized Component Selection

The problem we want to solve falls into the category of *multi-objective pure (mixed) binary integer/linear programming problems* (or *multi-objective pure (mixed) 0-1 programming problems*), where we are trying to simultaneously optimizing three objectives subject to certain constraints, and our set of variables can only take the values 0 or 1. The optimization problem is solved using the Simplex Algorithm that comes preinstalled with the SWI Prolog engine used. The overview of the process for finding an optimal set of components that satisfy the initial constraints is shown below:

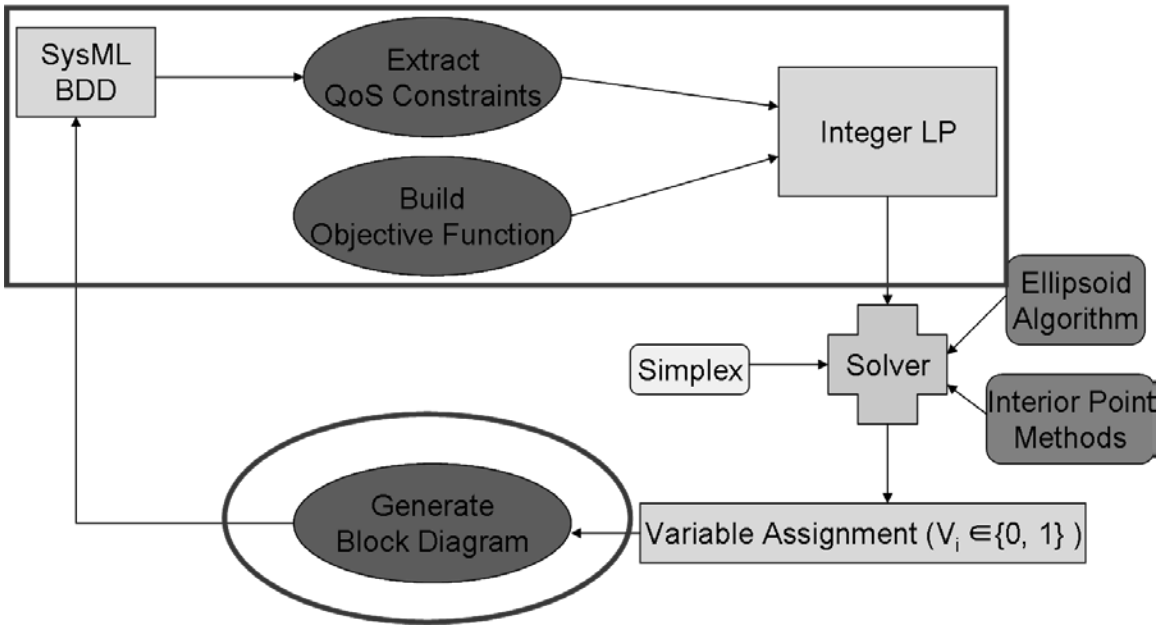

Figure 7.5 Multi-objective Linear/Integer Programming

First, XSLT transformation rules are applied on the XMI model, and Prolog rules are then ran on the resulted OWL file, with the results loaded into the knowledge base. Once all constraints are in the KB, the objective function is built, thus creating the Integer Linear Programming problem that a Solver, such as Simplex, will resolve, assigning values 0 and 1 for components represented as variables.

Let's first look at how the optimization problem is expressed. There are four parts that need to exist in order to form an optimization problem:

1. Variables. In this case, we need to know which component should be used as the GPS subsystem. There are three variables:

   vTI = number of instances for the TI component (can be 0 or 1).

vSiRF = number of instances for the SiRF component (can be 0 or 1).

vAmtel = number of instances for the Amtel component (can be 0 or 1).

2. Objective Function. We need to define our objective function in terms of the variables that we defined for this problem. We need to minimize both location error and power consumption, while maximizing the internal RAM memory. The objective function takes into account weights for constraints that are specified in a configuration file, assigning a higher weight to a constraint that is given by the user. The result will be:

   *minimize([2.04167*vTI, 1.62083*vAmtel, 1.70833*vSiRF'], S1,S)*.

3. Constraints. For our case, we have four constraints in terms of location error not exceeding 5, internal memory not exceeding 128, power consumption not exceeding 500, and finally that the number of GPS components should be one:

   4vTI + 3vSiRF + 2vAmtel =< 5

   100vTI + 120vSiRF + 128vAmtel =< 128

   480vTI + 400vSiRF + 425vAmtel =< 500

   1vTI + 1vSiRF + 1vAmtel = 1

4. Sign Restrictions. We can think of the sign restrictions as additional constraints on our variables. In this section, we specify that all variables positive:

vTI >= 0

vSiRF >= 0

vAmtel >= 0

Based on the OWL model loaded into Prolog's KB, the following LP constraints and objective function are generated:

:- use_module(library(simplex)).

post_constraints -->

constraint([0'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF',*

0'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel',*

0'http://www.csi.fau.edu/opp/lbs-ex.owl#TI']>=0),*

constraint([0'http://www.csi.fau.edu/opp/lbs-ex.owl#TI',*

0'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel',*

0'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF']>=0),*

constraint([0'http://www.csi.fau.edu/opp/lbs-ex.owl#TI',*

0'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel',*

0'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF']>=0),*

constraint([120'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF',*

128'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel',*

100'http://www.csi.fau.edu/opp/lbs-ex.owl#TI']=<128),*

constraint([480'http://www.csi.fau.edu/opp/lbs-ex.owl#TI',*

450'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel',*

400'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF']=<500),*

constraint([4'http://www.csi.fau.edu/opp/lbs-ex.owl#TI',*

2'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel',*

3'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF']=<5),*

constraint([100'http://www.csi.fau.edu/opp/lbs-ex.owl#Camera',*

200'http://www.csi.fau.edu/opp/lbs-ex.owl#GPS']=<300),*

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iPhone']=1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#I', -

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#X1', -*

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#X2', -*

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#X3']=0),*

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#LocationService', -

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF', -*

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#TI', -*

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel']=0),*

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iLocationService', -

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF', -*

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#TI', -*

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel']=0),*

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#GIFImageFormat', -

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Canon', -*

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Olympus']=0),*

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iImageEncoder', -

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Canon', -*

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Olympus']=0),*

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iGPS', -

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#GPS']=0),*

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iCamera', -

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Camera']=0),*

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iPhone', -

(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Phone']=0),*

constraint([-(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#GPS',*

'http://www.csi.fau.edu/opp/lbs-ex.owl#LocationService']>=0),

constraint([-(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Camera',*

'http://www.csi.fau.edu/opp/lbs-ex.owl#GIFImageFormat']>=0),

constraint([-(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#GPS',*

'http://www.csi.fau.edu/opp/lbs-ex.owl#iLocationService']>=0),

constraint([-(1)'http://www.csi.fau.edu/opp/lbs-ex.owl#Camera',*

'http://www.csi.fau.edu/opp/lbs-ex.owl#iImageEncoder']>=0),

constraint([-(2)'http://www.csi.fau.edu/opp/lbs-ex.owl#Phone',*

'http://www.csi.fau.edu/opp/lbs-ex.owl#iCamera',

'http://www.csi.fau.edu/opp/lbs-ex.owl#iGPS']>=0),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#I')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#X1')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#X2')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#X3')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#LocationService')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#TI')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#iLocationService')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#GIFImageFormat')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#Canon')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#Olympus')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#iImageEncoder')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#iGPS')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#GPS')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#iCamera')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#Camera')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#iPhone')),

constraint(integral('http://www.csi.fau.edu/opp/lbs-ex.owl#Phone')),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#I']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#X1']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#X2']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#X3']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#LocationService']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#TI']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iLocationService']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#GIFImageFormat']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#Canon']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#Olympus']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iImageEncoder']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iGPS']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#GPS']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iCamera']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#Camera']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#iPhone']=<1),

constraint(['http://www.csi.fau.edu/opp/lbs-ex.owl#Phone']=<1).

model(S) :- gen_state(S0), post_constraints(S0,S1),

minimize([2.04167'http://www.csi.fau.edu/opp/lbs-ex.owl#TI',*

1.62083'http://www.csi.fau.edu/opp/lbs-ex.owl#Amtel',*

1.70833'http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF'], S1,S).*

Prolog comes with a linear programming library called simplex, based on the Simplex Algorithm of solving linear programming problems. Generally, each inequalities that define a constraint, form a feasible region (a polytope – two-dimensional polygon). The simplex method moves from corner to corner until it can be proven that it has found the optimal solution. Each corner that it visits is an improvement over the previous one. Once it can't find a better corner, it knows that it has the optimal solution. The final result is a set of components that optimally satisfy the initial constraints:

http://www.csi.fau.edu/opp/lbs-ex.owl#Phone - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#iPhone - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#Camera - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#iCamera - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#GPS - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#iGPS - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#iImageEncoder - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#Olympus - 0

http://www.csi.fau.edu/opp/lbs-ex.owl#Canon - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#GIFImageFormat - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#iLocationService - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#AmteI - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#TI - 0

http://www.csi.fau.edu/opp/lbs-ex.owl#SiRF - 0

http://www.csi.fau.edu/opp/lbs-ex.owl#LocationService - 1

http://www.csi.fau.edu/opp/lbs-ex.owl#X3 - 0

http://www.csi.fau.edu/opp/lbs-ex.owl#X2 - 0

http://www.csi.fau.edu/opp/lbs-ex.owl#X1 - 0

http://www.csi.fau.edu/opp/lbs-ex.owl#I - 0

A value of 0 indicates that the component is not part of the solution, while a value of 1 indicates that the component is part of the solution

These components are processed by a Java program and the initial OWL file is stripped down to only contain the solution components. The final OWL file is then translated into XMI through the same process of applying XSLT transformation rules. Inside Rhapsody, the file is then imported and the result diagram is shown in Figure 6.4.

Chapter 8

EVALUATION

The section will contain an evaluation of the RDDA framework in terms of answering the initial research questions, what were the contributions brought, how the framework compares to other related work, and will end with a list of publications and some of the limitations of the framework.

8.1 Answers to the Research Questions

The main research question asked was:

How can we dramatically increase the productivity of system development?

By using a combination of modeling tools and technologies such as SysML, XSL, OWL, Prolog, and Java, the proposed RDDA framework closes the semantic gap between requirements, components and architecture by using compatible semantic models for describing both product requirements and component capabilities, including constraints.

A domain-specific representation language is designed that spans the application domain (mobile applications, in our case), the software design domain (UML/SysML meta-schema) and the component domains. This language

is used to represent Ontologies, which are textual representations that capture the semantics of concepts common to product requirements and design modeling, and the relationships between them. The ontology (meta-model) for requirements specifications is based on the Semantic Web Ontology Web Language (OWL). It covers concepts for product requirements (features and structure), and a set of constraint checking rules. These rules permit automatic consistency validation for requirements models before their use in architecture design. The RDDA ontology is expanded to cover knowledge representation for system architecture (UML and SysML diagrams) and for components (semantic annotations). In addition to specifications for product/subsystem features and capabilities, the RDDA ontology supports Quality of Service (QoS) and system resource constraints, such as energy, CPU load, bandwidth, weight, and volume.

Design automation is supported for architecture development by: *a)* machine-supported *mapping* of desired product/subsystem features and capabilities from formal requirements models to library components (*i.e.* component selection), and *b) synthesis* and maintenance of design structure diagrams (UML/SysML) using prototype diagrams. Component selection involves QoS translation from an end-to-end application-level representation to low-level component-level QoS parameters, and involves constrained optimization in a multidimensional parameter space. The architecture synthesis method works bottom-up from existing structural models and derives feasible configurations (UML/SysML diagrams) that satisfy the requirements. The ontology representation also supports requirements tracing to/from design

artifacts. The tracing function enables model reconfiguration in case of requirements model updates with reduced user support. Our system assists the requirements analyst and the architect/designer in an integrated MDE tool, and is not supposed to replace them.

RQ1: How can we model requirements?

Requirements can be modeled using SysML's Requirements Diagram, a diagram specifically designed to capture requirements hierarchies and requirements derivation, while the satisfy and verify relationships allow a modeler to relate a requirement to a model element that satisfies or verifies the requirements. In addition, the requirement diagram provides a link between the requirements management tools and system models.

- *What tools can we used to represent requirements?*

The requirement diagrams have been designed using IBM's Rational Rhapsody tool

- *What restrictions on the representation of requirements do we need to impose in order to be able to extract specific information out of them?*

A semi-structured textual representation of the requirements has to be imposed. This structure should be in the form of Subject Predicate Object.

- *How can we translate requirements into a more convenient format for processing?*

The modeling tool will export the model into XMI, where by using XSLT, a resulting OWL file is created.

RQ2: How do we validate the requirements?

Requirements are validated using Prolog rules that will check those requirements for completeness and consistency.

- *What is needed to validate requirements?*

  An ODL ontology for the requirements model that supports several types of numeric constraints that apply to features, capabilities, and resources. Rules have to be written in Prolog that will validate the requirements.

- *How should valid requirements be?*

  Valid requirements should not have any inconsistencies such as individuals that are referred but not defined, subsystems that require a feature that is not provided by any subsystem on which the initial system is dependent, dependency loops, and many more.

RQ3: How can we specify software and hardware system components?

SysML offers a range of diagrams that can be used to specify both software and hardware components, such as Block Definition Diagram, Internal Block Diagram, Parametric Diagram, etc.

- *What tools can we use to represent components?*

  IBM's Rational Rhapsody is the tool used to model software and hardware components (same tool we used for modeling requirements).

- *What kind of metadata should we tag the components with?*

  Components are tagged with information that specifies the type of component (block, interface, etc), what features it offers, and what constraints it meets.

RQ4: How do we automate architecture synthesis?

Starting with the SysML model that describes the existing components and how they relate to one another in terms of interfaces required and provided, a Prolog knowledge base is populated with facts. A set of rules are applied on the knowledge base, resulting a diagram with a subset of components that satisfy the interfaces, required features, and constraints. The list of components is presented to the user as a new SysML block diagram. Our approach takes into consideration the case when there are specific features or constraints that cannot be satisfied by the components that are part of the solution, thus making this methodology general and realistic.

RQ5: How can we achieve an optimal architecture synthesis?

By building an optimization problem (in terms of constraints and objective functions) from SysML diagrams using XSLT transformations, and applying a simplex algorithm in Prolog for solving that problem, an optimal set of solutions will be outputted and used for creating the final component diagram.

RQ6: What is the computational complexity of your methods?

The first component selection algorithm devised in Prolog and explained in section 6.4 is only optimized in terms of choosing the solution that satisfies all features and constraints, but will still search all possible solutions. Because of this, the running time of the algorithm is exponential 2^n, where n is the number of existing components. All possible combinations are searched for in order to find a match.

The second algorithm that uses linear programming and the simplex algorithm to solve the linear programming problem has a polynomial-time average-case complexity [70]. The worst-case complexity is still exponential [71].

The novelty of the optimization part of the RDDA framework comes not form solving the linear programming problem, which has been extensively researched in the past, but form actually formulating the optimization problem starting with diagrams written in SysML. Simplex has been chosen for the mere reason that it was made available as a library in the SWI-Prolog distribution.

8.2 Contributions

Main contributions of the proposed RDDA framework are summarized below:

- Building a Requirements Model using Requirements Diagrams available in the Systems Modeling Language (SysML).
- Proposing a semi-structural format for describing the textual part of requirements.
- Validating requirements in terms of completeness and consistency by using a set of transformations written in XSLT (SysML $\rightarrow$ XMI $\rightarrow$ OWL) and a set of rules written in Prolog.
- Building a component model in SysML using Block Definition Diagrams (BBD) and annotating the components with useful data (features, constraints)

- Writing a set of transformation in XSLT that will convert the SysML model into OWL for further processing
- Synthesizing the system architecture based on existing components using rules defined in Prolog.
- Implementing QoS Optimization by means of Integer/Linear Programming, where based on the type of SysML relationships (composition, aggregation, etc) that exists between different components, a set of constraints are derived based on rules written in Prolog.

8.3 Comparison with Other Methods

An initial comparison with other methodologies has been done in the related work section of the thesis. There are three main benefits that are not present in other existing works:

- Round-trip solution. The RDDA framework provides a round-trip solution where the user of the framework starts with the initial set of requirements and components inside the Modeling Tool (such as Rhapsody), and ends up with the final architecture imported back into Rhapsody.
- Support for Numeric Constraints. Numeric constraints described by numeric descriptors such as ConstraintUpperBound, ConstraintLowerBound, ConstraintFeasibleRegion, are supported in the RDDA framework. This allows for requirements that deal with

power consumption, cost, bandwidth, weight, internal memory, and many more, to be represented in the requirements and component model, and utilized when validating the requirements and building the final architecture.

- Representing the SysML models as a set of constraints (inequalities). By using the RDDA framework, constructing an optimization problem is possible because of the rules that were created, which will translate the structural diagrams (such as the Block Definition Diagram) into constraints.

8.4 Publications

This thesis is partly based on papers presented at international conferences and journals. This section summarizes the results presented in those papers.

Paper A – Ionut Cardei, Mihai Fonoage, Ravi Shankar. "*Framework for Requirements-Driven System Design Automation*". In the 1st IEEE Systems Conference, Hawaii, USA, April 2007.

This paper contained a high-level introduction of the Requirements-Driven Design Automation (RDDA) framework for improving the system design productivity through automation. The paper described the main idea behind the proposed approach, namely to describe product requirements and component semantics with a common language that supports automated reasoning on knowledge from application requirements, components and design models. The

common specification is used to annotate library components with semantic descriptions of their capabilities and constraints. A rule-based reasoning engine performs two functions after validating the requirements specification: matching of requirements with components, and UML diagram synthesis.

Paper B – Ionut Cardei, Mihai Fonoage, Ravi Shankar. "*Model Based Requirements Specification and Validation for Component Architectures*". In the 2nd IEEE International Systems Conference, Montreal, QC, Canada, April 2008.

The above paper describes a methodology for specification of functional product requirements using a language built on the semantic web's OWL. We presented a framework for model verification for completeness and consistency. Verification of requirements models is done using a Prolog environment. Completeness verification rules are defined to search for incomplete or missing model elements. Consistency checking rules search for conflicting requirements model statements and for numeric constraint conflicts on QoS and system resources. Operation of these rules are exemplified with elements from a location-based system application specification. System requirements specification is performed with a SysML modeling tool, enhanced with the capability to express requirements models in the suitable representation.

Paper C – Mihai Fonoage, Ionut Cardei, and Ravi Shankar, "*Mechanisms for Requirements Driven Component Selection and Design Automation*", in the 3rd IEEE International Systems Conference, Vancouver, Canada, March 2009.

This paper presents a methodology for architecture synthesis driven by requirements and constraints. Starting with the SysML model that describes the

existing components and how they relate to one another in terms of interfaces required and provided, we populate a Prolog knowledge base with facts. A set of rules are applied on the knowledge base, resulting a diagram with a subset of components that satisfy the interfaces, required features, and constraints. The list of components is presented to the user as a new SysML block diagram. Our approach takes into consideration the case when there are specific features or constraints that cannot be satisfied by the components that are part of the solution, thus making our methodology general and realistic.

Paper D – Mihai Fonoage, Ionut Cardei. “Mechanisms for Requirements Driven Component Selection and Design Automation”. Accepted for the IEEE Systems Journal Special Edition.

The above journal paper is an extended version of Paper C, where a section on Prolog was added in which it was briefly described the inner workings of the language, where more details for the component selection algorithms was also added, improving their description, and finally, where more Prolog predicates were provided to better aid in the understanding of the proposed methodology.

8.5 Limitations

This section describes some of the existing limitations of our framework, such as:

- An initial investment in building the component ontologies either directly or through use of modeling.

- The need for a semi-structured textual description of the requirements such that they can be parsed to extract the needed information.
- Textual requirements have to be represented using Requirements Diagrams and making sure that the structure of the text conforms to the form needed in order to be able to extract information.
- Our approach supports only *static* description of capabilities/constraints. It does not address design artifact compatibility based on dynamic behavior. For instance, two components, one needing and one providing the same *LocationQuery* interface may differ in the order in which operations are invoked. The behavior specification for components and classes is given by UML behavioral diagrams (e.g. sequence and state machine diagrams). Currently, the diagram synthesis rules do not check for this behavior consistency when matching requirements with candidate design artifacts. We will look for a solution to this problem as it would improve the quality of the generated design.
- The XMI-based method for model conversion introduces file consistency issues, as modeling tools do not always support the most recent versions of XMI.

8.6 Lessons Learned

The core part of the work presented in this thesis has been published in international conferences and journals. We have received valuable feedback through the reviewing process.

We learned that working with a multitude of technologies, such as SysML, XML/XMI, XSL (XSLT, XPath), Semantic Web (OWL, RDF), Prolog, Java, and trying to integrate them under on single umbrella that is the RDDA framework is not an easy task. Each language comes with its own set of requirements and challenges, and mashing all of them into one single platform required considerable work.

Chapter 9

CONCLUSION

9.1 Summary of Results and Contributions

In this thesis, we present a framework for improving the design productivity through automation. The main idea behind our approach is to describe product requirements and component semantics with a common language that supports automated reasoning on knowledge from application requirements, components and design models. The common specification is used to annotate library components with semantic descriptions of their capabilities and constraints. A rule-based reasoning engine performs two functions after validating the requirements specification: matching of requirements with components, and UML diagram synthesis.

In addition, we have introduced a methodology for specification of functional product requirements using a language built on the semantic web's OWL. We presented a framework for model verification for completeness and consistency. Verification of requirements models is done using a Prolog environment. Completeness verification rules are defined to search for incomplete or missing model elements. Consistency checking rules search for

conflicting requirements model statements and for numeric constraint conflicts on QoS and system resources. Operation of these rules is exemplified with elements from a location-based system application specification. System requirements specification is performed with a SysML modeling tool, enhanced with the capability to express requirements models in the suitable representation.

Furthermore, we describe a methodology for component selection and architecture synthesis driven by requirements. The methodology starts with the SysML model that describes the existing components and how they relate to one another in terms of interfaces required and provided. A semi-formal requirements model, also developed in SysML, describes system functional (symbolic) and Quality of Service (numeric) requirements/constraints.

It is likely that the number of feasible system designs that seem to satisfy functional and QoS requirements could be large because of combinatorial explosion from existing options. The large design search space could be too large to handle for people. Our systems implements an automated search method looking for feasible configurations. The requirements model and the initial design model are loaded to a Prolog knowledge base as facts. A design synthesis algorithm using Constraint Logic Programming (CLP) in Prolog operates on the facts from the knowledge base and generates a set of components and design relationships that form a structural model that satisfies the interfaces, features, and constraints required by the system. The generated diagram is added to the project and is loaded back into the modeling tool. Our

approach takes into consideration the case where there are specific features or constraints that cannot be satisfied by the components that are part of the solution, as it provides users with information identifying the requirements and constraints that cannot be satisfied. The user can then go back and adjust the requirements model or add components to the model, practically maintaining a component library.

Finally, an optimization problem is built and solved starting with SysML models, and using Prolog rules that translate the models (block definition diagrams that contain relationships between existing components, annotated with features and constraints) into constraints that are processed by the Prolog rule engine and used to solve the architecture synthesis problem.

9.2 Future Work

Future work should be related to the following aspects of the framework.

The XMI-based method for model conversion introduces file consistency issues, as modeling tools do not always support the most recent versions of XMI. A better and more integrated solution is to plug the RDDA system directly in the modeling tool framework. For closed source tools this could be difficult. However, Eclipse has an open framework. We consider integrating our toolset with Eclipse as a plug-in for an existing UML 2.0 modeler, such as Omondo [72].

In the requirements specification and validation area, future improvements of the SysML requirements specification method are needed, specifically, look into developing a new SysML profile for modeling requirements

concepts. New rules for consistency verification should be added. A long-term goal is to integrate all tools for requirements specification, verification, and design synthesis into a common platform based on Eclipse.

Consideration for modeling the behavioral part of a system, by using specific SysML diagrams, will be given. A long-term goal is to integrate all tools for requirements modeling and validation, component selection, and design composition, into a common platform based on Eclipse.

BIBLIOGRAPHY

[1] UML – Unified Modeling Language. Online: http://www.uml.org/.

[2] SysML – Systems Modeling Language. Online: http://www.omgsysml.org/.

[3] DOORS. Online: http://www.telelogic.com/Products/doors/doors/index.cfm.

[4] RequisitePro. Online: http://www-01.ibm.com/software/awdtools/reqpro/.

[5] OWL – Web Ontology Language. Online: http://www.w3.org/2004/OWL/.

[6] Prolog. Online: http://www.swi-prolog.org/.

[7] OMG – Object Management Group. Online: http://www.omg.org/.

[8] List of UML tools. Online: http://en.wikipedia.org/wiki/List_of_UML_tools.

[9] List of SysML vendors. Online: http://sysml-directory.omg.org/vendor/list.htm.

[10] Enterprise Architect. Online:
http://www.sparxsystems.com/products/ea/index.html.

[11] Magic Draw. Online: http://www.magicdraw.com/.

[12] Rhapsody. Online:
http://modeling.telelogic.com/products/rhapsody/index.cfm.

[13] Model Driven Architecture. Online: http://www.omg.org/mda/.

[14] Anneke Kleppe, Jos Warmer, Wim Bast. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley, April 21, 2003.

[15] Extensible Markup Language. Online: http://www.w3.org/XML/.

[16] Extensible Markup Language Metadata Interchange. Online: http://www.omg.org/technology/documents/formal/xmi.htm.

[17] Extensible Stylesheet Language Family. Online: http://www.w3.org/Style/XSL/.

[18] Extensible Stylesheet Language Transformations. Online: http://www.w3.org/TR/xslt.

[19] Michael Kay. XSLT 2.0 Programmer's Reference, Third Edition. Wrox Press, 2004.

[20] XML Path Language. Online: http://www.w3.org/TR/xpath/

[21] Michael Kay. XPath 2.0 Programmer's Reference. Wiley Publishing, 2004.

[22] Semantic Web. Online: http://semanticweb.org/.

[23] XML Schema. Online: http://www.w3.org/XML/Schema.html

[24] Resource Description Framework. Online: http://www.w3.org/TR/rdf-primer/

[25] Resource Description Framework Schema. Online: http://www.w3.org/TR/rdf-schema/

[26] Grigoris Antoniou, Frank van Harmelen. A Semantic Web Primer. The MIT Press, 2004.

[27] Ontologies. Online: http://www.w3.org/standards/semanticweb/ontology

[28] David Hyland-Wood, David Carrington, Simon Kaplan. "Enhancing Software Maintenance by using Semantic Web Techniques".

[29] Haruhiko Kaiya, Motoshi Saeki. "Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach".

[30] Evren Sirin, James Hendler, Bijan Parsia. “Semi-automatic Composition of Web Services using Semantic Descriptions”.

[31] Biplav Srivastava, Jana Koehler. “Web Service Composition – Current Solutions and Open Problems”.

[32] Rosario Girardi, Alisson Neres Lindoso. “An Ontology-based Knowledge Base for the Representation and Reuse of Software Patterns”.

[33] Markus Schacher. “CASSANDRA: An Automated Software Engineering Coach”.

[34] Kamin Whitehouse, Feng Zhao, Jie Liu. “Semantic Streams: a Framework for Declarative Queries and Automatic Data Interpretation”.

[35] Michael G. Hinchey, James L. Rash, Christopher A. Rouff, Denis Gracanin. “Achieving Dependability in Sensor Networks through Automated Requirements-based Programming”.

[36] Clemens Reichmann, Daniel Gebauer, Klaus D. Müller-Glaser. “Model Level Coupling of Heterogeneous Embedded Systems”.

[37] Vimal Mayank, Natalya Kositsyna, Mark Austin. “Requirements Engineering and the Semantic Web: Part II. Representation, Management, and Validation of Requirements and System-Level Architectures”, ISR Technical Report.

[38] Aniruddha Gokhale. “Component Synthesis with Model Integrated Computing (CoSMIC)”. Online: http://www.dre.vanderbilt.edu/cosmic/html/overview.shtml

[39] P. Volgyesi, M. Maroti, S. Dora, E. Osses, A. Ledczi, T. Paka. “Embedded Software Composition and Verification, Technical Report”, TR #: ISIS-04-503.

[40] Hiroshi Wada, Junichi Suzuki, Katsuya Oba. “Modeling Non-Functional Aspects in Service Oriented Architecture”. In Proc. of the 2006 IEEE International Conference on Services Computing, September 2006.

[41] Robert Seater, Daniel Jackson, and Rohit Gheyi. “Requirement Progression in Problem Frames: Deriving Specifications from Requirements”. In Requirement Engineering Journal (REJ), volume 12, number 2, pages 77-102, 2007.

[42] Haruhiko Kaiya and Motoshi Saeki. “Ontology based requirements analysis: Lightweight semantic processing approach”. In Fifth International Conference on Quality Software, 2005. (QSIC 2005). Volume, Issue , 19-20 Sept. 2005 Page(s): 223 – 230.

[43] Ravenflow. “Raven - Requirements Authoring and Validation Environment”. Online: http://www.ravenflow.com/products/index.php

[44] Sören Auer and Klaus-Peter Fähnrich. “SoftWiki – Agiles Requirements-Engineering für Softwareprojekte mit einer großen Anzahl verteilter Stakeholder”. Statuskonferenz Forschungsoffensive “Software Engineering 2006”, 26.-28. June 2006, Leipzig.

[45] Barrett R. Bryant, Beum-Seuk Lee, Fei Cao, Rajeev R. Raje, Andrew M. Olson, and Mikhail Auguston. “From Natural Language Requirements to Executable Models of Software Components”. In Proc. of the Monterey

Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, pp. 51-58, 2003.

[46] Sören Auer and Jens Lehmann. “What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content”. In Franconi et al. (eds), Proceedings of European Semantic Web Conference (ESWC'07), LNCS 4519, pp. 503-517, Springer, 2007.

[47] Leonid Kof. “Natural Language Processing for Requirements Engineering: Applicability to Large Requirements Documents”. In Alessandra Russo, Artur Garcez, and Tim Menzies, editors, Automated Software Engineering, Proceedings of the Workshops, Linz, Austria, September 21 2004.

[48] Deb Jacobs. “Interpreting Requirements In a He Said/She Said World”. Crosstalk – The Journal of Defense Software Engineering, Dec 2006 Issue.

[49] Dr. Ralph R. Young. “Twelve Requirements Basics for Project Success”. Crosstalk – The Journal of Defense Software Engineering, Dec 2006 Issue.

[50] REWERSE, Reasoning on the Web with Rules and Semantics. Online: http://rewerse.net/.

[51] I3: Composition and Typing. Online: http://rewerse.net/I3/.

[52] Reuseware Composition Framework. Online: http://st.inf.tu-dresden.de/reuseware/index.php/Main_Page.

[53] XcerptT. Online: http://www.ida.liu.se/~artwi/XcerptT.

[54] Andreas S. Andreou, Dimitrios G. Vogiatzis, George A. Papadopoulos. “Intelligent Classification and Retrieval of Software Components”. In Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International, Sept. 2006.

[55] Robert G. Bartholet, David C. Brogan, Paul F. Reynolds, Jr.. “The Computational Complexity of Component Selection in Simulation Reuse”. In Proceedings of the 37th conference on Winter simulation, Dec. 2005.

[56] Paul Baker, Mark Harman, Kathleen Steinhöfel, Alexandros Skaliotis. “Search Based Approaches to Component Selection and Prioritization for the Next Release Problem”. In Proceedings of the 22nd IEEE International Conference on Software Maintenance, 2006.

[57] Jing Dong, Sheng Yang, Lawrence Chung, Paulo Alencar, Donald Cowan. “A COTS Architectural Component Specification Stencil for Selection and Reasoning’. In ACM SIGSOFT Software Engineering Notes archive, Volume 30 , Issue 4, July 2005.

[58] M.R. Fox, D.C. Brogan, P.F. Reynolds. “Approximating Component Selection”. In 2004 Winter Simulation Conf., vol.1, pp. 429-434, 2004.

[59] Jun Guo, Bin Zhang, Kening Gao, Hongning Zhu, Ying Liu. “A Method of Component Selection within Component Based Software Development Process”. In 2004 International Conference on Software Process, pp. 1-3, 2004

[60] Georgiana Hamza-Lup, Ankur Agarwal, Ravi Shankar, Cyril Iskandar. “Component Selection Strategies Based on System Requirements

Dependencies on Component Attributes". In 2nd Annual IEEE Systems Conf., 2008

[61] Olaf Hartig, Martin Kost, and Johann-Christoph Freytag. "Automatic Component Selection with Semantic Technologies". In 4th International Workshop on Semantic Web Enabled Software Engineering at ISWC, pp. 1-14, 2008.

[62] Wes J. Lloyd. "A Common Criteria Based Approach for COTS Component Selection". In 6TH GPCE Young Researchers Workshop, pp. 1-7, 2004.

[63] Vowlidator OWL. Online:
http://projects.semwebcentral.org/projects/vowlidator/

[64] Ionut Cardei, Mihai Fonoage, Ravi Shankar. "Framework for Requirements-Driven System Design Automation". In the 1st IEEE Systems Conference, Hawaii, USA, April 2007.

[65] Ionut Cardei, Mihai Fonoage, Ravi Shankar. "Model Based Requirements Specification and Validation for Component Architectures". In the 2nd IEEE International Systems Conference, Montreal, QC, Canada, April 2008.

[66] Mihai Fonoage, Ionut Cardei. "Mechanisms for Requirements Driven Component Selection and Design Automation". Accepted for the IEEE Systems Journal Special Edition.

[67] Kelly L. Croxton. "LP Tutorial". Online:
http://fisher.osu.edu/~croxton_4/tutorial/intro.html.

[68] Mixed Integer Programming. Online: http://www.cs.sandia.gov/opt/survey/mip.html

[69] Mokhtar S. Bazaraa, Hanif D. Sherali, C. M. Shetty. “Nonlinear programming: theory and algorithms”. Wiley-Interscience, 3 edition, May 5, 2006.

[70] Spielman, Daniel, Teng, Shang-Hua. “Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time”. In Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing, ACM, pp. 296–305.

[71] Andrei Z. Broder, Martin E. Dyer, Alan M. Frieze, Prabhakar Raghavan, Eli Upfal. “The worst-case running time of the random simplex algorithm is exponential in the height”. Technical Report: CS95-07, 1995.

[72] Omondo. Online: www.omondo.com.