Model Based Requirements Specification and Validation for Component Architectures

Ionut Cardei, Mihai Fonoage, Ravi Shankar
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
Email: {*icardei@cse., mfonoage@, ravi@cse.*}*fau.edu*

Abstract *— 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 and, in turn, cause errors that trickle down in design and implementation with consequences on the overall development cost. In this paper 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. This methodology is part of the Requirements Driven Design Automation framework (RDDA) that we develop for component-based system development. The RDDA framework uses an ontology-based language for semantic description of functional product requirements, UML/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 OWL is converted from SysML 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.*

Keywords *— requirements specification, consistency validation, design automation*

I. Introduction

Requirements specification is a high-risk activity in the system development cycle as errors and omissions can be introduced that later have cascading effects on product design cost and product quality. Current methods for requirements representation rely on natural language descriptions that are not suitable for automated verification. A way to improve the design and development productivity is by implementing a methodology for model-based requirements specification with automated model verification.

In this paper we present the requirements specification and verification methodology developed as part of the Requirements-Driven Design Automation (RDDA) framework. This project aims to reduce the development cost by partially automating the process of architecture design from requirements to existing library components. The framework can be applied to design of software and systems that use UML [14] or the System Modeling Language (SysML) [13]. This research is supported by the Motorola Corporation, Plantation, Florida, under the *One Pass to Production* (OPP) project and is currently ongoing at Florida Atlantic University's Center for Systems Integration. The RDDA architecture was introduced in [8].

The proposed approach closes the semantic gap between requirements, components and architecture by using compatible semantic models for describing product requirements, component capabilities, and constraints, such as Quality of Service (QoS) and resource limitations. We design a domain-specific representation language called the OPP Design Language (ODL) that covers the requirements domain for mobile applications, the software design domain (UML/SysML metaschema), and the component domain. The ontology representation also supports requirements tracing to design artifacts. This function is generally used by modeling tools with requirements management capabilities to track changes to design and implementation artifacts.

Various requirements and design elements can be described in ODL as ontologies, which are representations for the semantics of concepts common to product requirements and design modeling, and the relationships between them. The ontology (meta-model) for ODL is built on the Ontology Web Language (OWL) [16] which is based on XML.

The machine-readable Description Logic representation for requirements and UML/SysML design artifacts makes possible automated model verification and model transformation.

We have implemented an ODL requirements specification method integrated with the Rhapsody [10] UML/SysML modeling tool. The requirements model is exported and then loaded into a Prolog knowledge base, where a set of rules perform completeness and consistency validation before being used for architecture design. The verification rules look for conflicts in specification of QoS constraints (e.g. maximum query delay) and system resource constraints (e.g. power, CPU load, weight).

This paper continues in the next section with a description of the RDDA framework architecture. Section III describes the ODL requirements specification language, the model verification mechanisms, and the representation methodology with SysML. Related work is described in section IV and section V summarizes conclusions and discusses future work.

II. Architecture and Methodology

This section provides an overall description of the RDDA framework. The main components and the workflow are illustrated in Figure 1. The framework implements these high-level functions that together provide system design au-

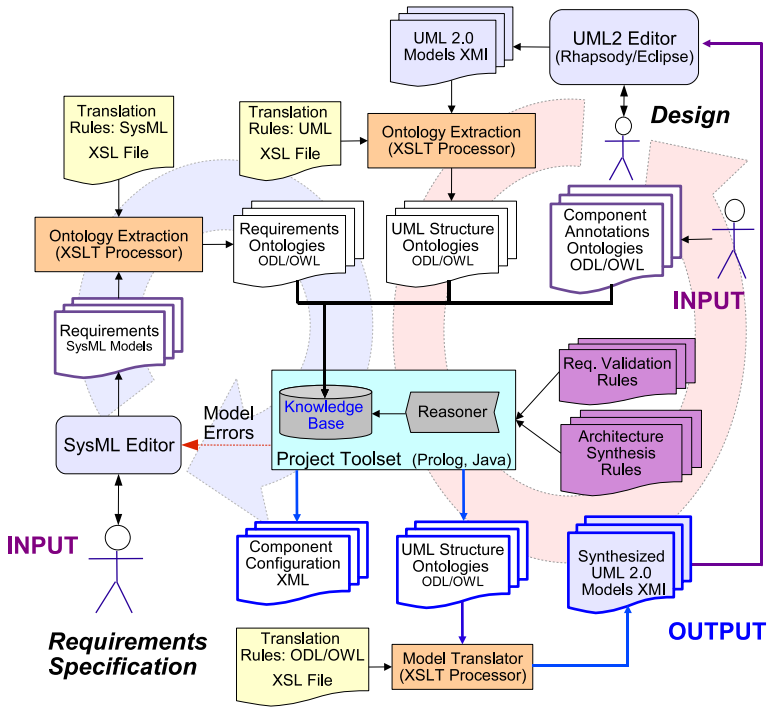

Figure 1. Requirements-Driven Design Automation framework architecture.

tomation from requirements: requirements modeling, design modeling, model verification, and architecture synthesis.

The various framework components communicate using the *OPP Design Language* (ODL).

The RDDA framework from Figure 1 provides rountrip engineering through two workflows, the *requirements workflow* and the *design workflow*.

The *requirements workflow* begins with requirements specialists describing product requirements using a modeling tool, in our case a SysML editor, such as Telelogic's Rhapsody [10]. 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 III-C.

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 [3] 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 III-A.

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 plugin. 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. A brief description

of how this is accomplished as part of the *design workflow* follows. 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. The *design workflow* processes will be addressed in more detail in an upcoming article.

The RDDA framework supports an iterative design process, where new requirements are added incrementally and the system architecture condenses through multiple model synthesis cycles. As part of future work, we will integrate this framework in the Eclipse environment, bypassing the need for intermediate XMI conversion, as Eclipse supports programmatic access to internal UML 2 model representations. A useful feature of this framework we contemplate is the ability to *synchronize* the structure of designated placeholder (auto-generated) blocks with requirements models as they change.

Currently, our framework supports only model validation based on features, requirements constraints, and structural constraints. A future direction for research for us is to add mechanisms for validating behavior model constraints – checking inter-component compatibility, based on sequence diagrams and state transition diagrams.

III. Requirements Specification and Validation

This section begins with a short description of the ODL ontology for requirements model, continues with the methodology for model verification, and ends with the model specification in SysML.

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 2. 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), and 4) requirement statement management (versioning, tracking, dependencies).

The ODL metamodel ontologies are developed with Protegé [1]. 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 1.

Table 1
The main 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. A requirement statement in English begins in Figure 3 with a % sign and is followed by its ODL representation written as the equivalent Prolog clause stored in the knowledge base. Some ODL statements related to versioning and background knowledge are omitted. The Prolog predicate notation is more compact and more readable than the OWL XML syntax, and it preserves the OWL $\langle Property, Subject, Object\rangle$ triple semantics. The Triple20 RDF/OWL Prolog library emulates the XML namespace notation $ns:ID$ that simplifies URI representation considerably. The $individual(classname, instance)$

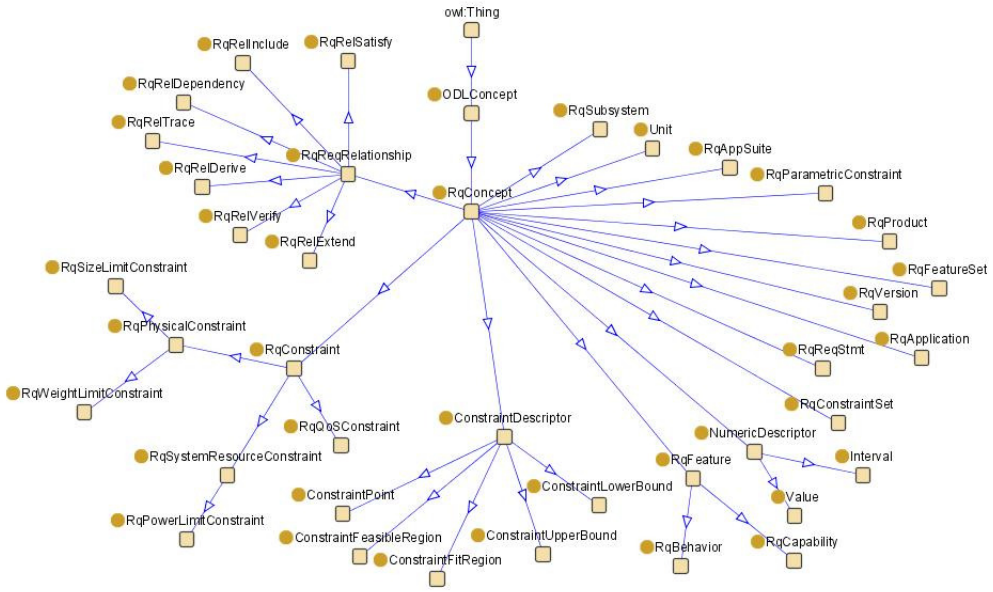

Figure 2. A snapshot of the OWL class hierarchy from the ODL metamodel. Edges represent the OWL subclass relationship.

predicate declares class members.

```
% 1.1 The Product is the Cell Phone.
individual(odl:'RqProduct', lbs:'Product_CellPhone').

% 1.2 The CellPhone product has a GPS subsystem.
individual(odl:'RqSubsystem', lbs:'Subsystem_GPS').
odl:hasSubsystem(lbs:'Product_CellPhone', lbs:'Subsystem_GPS').

% 1.3 The GPS subsystem provides Location Service.
individual(odl:'RqCapability', lbs:'Service_Location').
odl:providesFeature(lbs:'Subsystem_GPS', lbs:'Service_Location').


% 1.4 The GPS subsystem provides Proximity Service.
individual(odl:'RqCapability', lbs:'Service_Proximity').
odl:providesFeature(lbs:'Subsystem_GPS', lbs:'Service_Proximity').

% 1.5 The GPS subsystem provides location accuracy of maximum 10 m.
individual(odl:'ConstraintUpperBound', lbs:'CO_GPS_LocErrorMax').
individual(odl:'Interval', lbs:'Interval_GPS_LocError').
individual(odl:'Value', lbs:'value_10m').
individual(odl:'RqQoSConstraint', lbs:'QoSConstraint_LocationError').

odl:dependsOn(lbs:'Service_Location',
              lbs:'QoSConstraint_LocationError').

odl:hasSubject(lbs:'CO_GPS_LocErrorMax', lbs:'Subsystem_GPS').
odl:hasConstraintObject(lbs:'CO_GPS_LocErrorMax',
                        lbs:'QoSConstraint_LocationError').
odl:hasNumericDescriptor(lbs:'CO_GPS_LocErrorMax',
                         lbs:'Interval_GPS_LocError').
odl:maxValue(lbs:'Interval_GPS_LocError', lbs:'value_10m').
odl:numericValue(lbs:'value_10m', 10).
```

Figure 3. Example requirements statements (ODL triples) in Prolog clause notation.

The example statements consist of instance (individual) declarations and property declarations. Statement 1.1 says that Product_CellPhone is a product. Statement 1.2 says that Product_CellPhone has a GPS subsystem. 1.3 says that the GPS subsystem provides location service, and statement 1.4 specifies that the GPS subsystem provides proximity service. These services are defined as instances of the $RqCapability$ class. Statement 1.5 specifies a QoS constraint – an upper bound on the maximum acceptable GPS localization error.

The lbs:'CO_GPS_LocErrorMax' instance from class $ConstraintUpperBound$ links the necessary constraints properties. Property $hasSubject$ refers to the subsystem on which the constraint applies – the GPS subsystem. The $hasConstraintObject$ property links to the semantics of this constraint (it it a *location error* constraint). The $hasNumericDescriptor$ property points to an $Interval$ instance that specifies the $maxValue$ instance with its numeric property representing 10 meters. (The statements for the unit – meter – are omitted.)

Let us consider a requirements model $\mathcal{M}_{\mathcal{R}} = \langle P, A, S, F, C, N_C, R \rangle$, 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, N_C 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 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.

A. 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 BBN's Vowlidator OWL checker [4].

Table 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	$\hat{d_s}(s_1, s_2) \iff u_s^*(s_1, s_2) \vee d_s^*(s_1, s_2)$
s requires feature f	$f_r(s, f)$
s provides feature f	$f_p(s, f)$
s depends on feature f	$d_{sf}(s, c) \iff f_r(s, f) \vee f_p(s, f)$
f depends on constraint c	$d_{fc}(f, c)$
constraint descriptor cd	$cd(s, c, nd)$

Then, 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.

A Prolog rule that checks for missing declaration of a subsystem is:

```
checkComp_Subsys :-
  (hasIndividual(odl:'RqSubsystem', S) ;
  hasIndividual(odl:'RqProduct', S)),
  odl:hasSubsystem(S, Sub),
  \+ hasIndividual(odl:'RqSubsystem', Sub).
```

The ';' operator stands for logical OR in Prolog and the '+' operator is logical not. This rule is equivalent with the following logical sentence:
$\forall s \in P \cup S, d_s(s, sub) \wedge sub \notin S \Rightarrow$
$missingSubsystem(Sub)$
Similar rules are defined to identify other missing or incomplete model elements.

A second set of rules search for cases where a subsystem s_1 $requires$ a feature that is not $provided$ by any subsystems s_2 on which s_1 is dependent on (transitive and inclusion).
The rule in first order logic is:
$\forall s_1 \in P \cup S, f_r(s, f) \wedge \neg(\exists s_2 \in S, \hat{d_s}(s_1, s_2)$
$\wedge f_p(s_2, f)) \Rightarrow missingRequiredFeature(s, f)$.

B. Consistency Verification

Consistency verification involves these checks: a) model structure sanity, b) constraint validation.

a) *Model structure sanity checks*. Rules that verify the requirements model internal structure: dependency loops ($\exists s \in S, \hat{d_s}(s, s)$) and subsystem unique ownership ($\forall s_2 \in S, \exists! s_1 \in S \cup P, u_s(s_1, s_2)$).

b) *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 of this rule follows:

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

Additional policies for consistency checking between required and required constraints will be developed as part of the ongoing project.

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.

To explain the rules for checking the numeric constraints let us assume the predicates from the above Constraint Matching Rule are satisfied. In addition, nd_1 and nd_2 specify intervals $[m_i, M_i]$, for i=1,2, respectively. Value points are also supported by the ODL ontology, in this case nd_i is described by a number v_i, i=1 or 2.

The checking rules for *valid QoS constraints* are listed in Table 3. UB stands for UpperBound constraint descriptor, LB for lower bound.

Different policies are created similarly for checking system resource constraints, as numeric resource limits have different meaning.

Table 3
Consistency checking rules for QoS constraints. The rules indicate valid cases or conflicts ('C'). The corresponding numeric descriptor are intervals $[m_i, M_i]$ or point values v_i for $i = 1, 2$.

cd_1 \ cd_2	*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	$v_1 = v_2$

We list in Figure 4 the Prolog rule for checking all UB–UB constraint descriptors that involve features required by a subsystem S.

```
checkConsist_constraintUpperBound(S) :-
        odl:requiresFeature(S, F), odl:dependsOn(F, Constraint),
        hasIndividual(odl:'RqConstraint', Constraint),
        odl:hasSubject(CDReq, S),
        odl:hasConstraintObject(CDReq, Constraint),
        individual(odl:'ConstraintUpperBound', CDReq),
        odl:hasNumericDescriptor(CDReq, NDReq),
        odl:maxValue(NDReq, ValueReq),
        odl:numericValue(ValueReq, MaxReq),

        odl:hasConstraintObject(CDProv, Constraint),
        CDProv \== CDReq,
        individual(odl:'ConstraintUpperBound', CDProv),
        odl:hasSubject(CDProv, SDep),
        dependsOnTrans(S, SDep),
        odl:hasNumericDescriptor(CDProv, NDProv),
        odl:maxValue(NDProv, ValueProv),
        odl:numericValue(ValueProv, MaxProv),
        % negate the valid UB-UB QoS constraint rule:
        MaxReq < MaxProv.
```

Figure 4. Prolog rule for checking all UpperBound-UpperBound constraints that involve each feature required by subsystem S.

Backward-chaining inference in Prolog provides a very 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.

```
% Req 2.4
%  The LBS application provides localization
%  accuracy of 5 m. (this is an
% UpperBound-UpperBound consistency conflict)
individual(odl:'ConstraintUpperBound'
          , lbs:'CO_AppLocError_Max').
individual(odl:'Interval',
           lbs:'Interval_AppLocErrorMax').
individual(odl:'Value', lbs:'value_5m').

odl:hasSubject(lbs:'CO_AppLocError_Max',
               lbs:'LBSApplication').
odl:hasConstraintObject(lbs:'CO_AppLocError_Max',
              lbs:'QoSConstraint_LocationError').
odl:hasNumericDescriptor(lbs:'CO_AppLocError_Max',
                lbs:'Interval_AppLocErrorMax').
odl:maxValue(lbs:'Interval_AppLocErrorMax',
             lbs:'value_5m').
odl:numericValue(lbs:'value_5m', 5).
```

Figure 5. Prolog clauses from the requirements model defining an UpperBound location error constraint description on the LBS application that conflicts with the location error constraint assumed by the GPS subsystem from Figure 3.

Figure 5 lists Prolog clauses describing an UpperBound constraint on the LBS application localization error. The 5 meter error upper bound required by the application conflicts with the 10 meter upper bound provided by the GPS subsystem, shown in Figure 3. Such errors can be easily overseen in specification of complex applications. The rule from Figure 4 will find the conflicting constraints and the RDDA framework will allow the user to fix the specification.

C. Requirements Specification with SysML

For requirements specification, SysML offers requirements diagram to represent textual requirements and the relationships between them. SysML can represent these in a graphical, tabular, or tree structure format. This type of diagram is used to create taxonomy of the captured requirements statements.

Our method relies on using the text from the requirements blocks to specify requirements model ODL statements, instead of natural language statements. The text grammar represents ODL OWL triplets $\langle Property, Subject, Object\rangle$ in the form "*Object type = Subject Property Object*". For example, we could have *Feature = GPS providesFeature Location Proximity*, where the OWL triple object type is $RqFeature$, the subject is the GPS subsystem, the property is $providesFeature$ and the object is $LocationProximity$. The triple subject and object indicate model element IDs. The object type can specify a system, subsystem, product, component, constraint, feature, or capability. One improvement that will be implemented is to specify the property as stereotype.

For the transformation from SysML to ODL ontology we first export the SysML model to XMI. The Saxon XSLT processor translates the XMI code, including the specially formatted requirements diagram statements, to ODL format, which is then loaded into the Prolog knowledge base.

Figure 6 illustrates a requirements diagram for a location-based mobile application running on a cell phone product. These requirement are also shown in Prolog format in Figure 3.

Requirement 1.1 is at the highest level, where we specify that the system we are trying to build is a Phone. From this requirement, 1.2 is derived or decomposed, where we detail the fact that the Phone needs to provide a GPS. Statements 1.3 and 1.4 specify the services that need to be implemented, location and proximity services. The functionality of both these requirements is extended by constraining the GPS to provide a location accuracy of maximum 10 meters and a location query response time of maximum 10 seconds. Beside these extensions, requirement 1.3 is also extended by 1.7 and 1.8 specifying some extra capabilities for the GPS, such as altitude, course, and speed information.

If 1.3 and 1.4 could not have been performed unless location accuracy and query response time constraints were valid, then we would have had ≪include≫, but since 1.3 and 1.4 can exist without the need for 1.5 and 1.6 to be realized, we use stereotype ≪extends≫. Requirement 1.5 is dependent on 1.6 because, if the response time is greater then 10s and the user is moving, then the accuracy is affected. The shorter the response time is, the better the location accuracy is.

We notice how requirements 1.3 and 1.4 are traced into the model – the two use cases in the LBS model. We also observe that the GPS sub-system satisfies the QoS requirements 1.5 and 1.6, and the functional requirements 1.7 and 1.8. A test case called $QueryResponseTimeTests$ is used to verify if requirement 1.6 has been fulfilled or not. The ID in the requirement blocks is often used for integration with other requirement management tools.

We recognize that this method for requirements specification is just a first step. While it permits integration of concepts visually modeled in SysML, textual specification for ODL statements, especially for numeric constraints, can be tedious. We consider implementing a SysML/UML Profile to handle the specification of constraints, capabilities, and features, as opposed to having them represented as stereotypes or directly in the requirement text. There are two ways: using the lightweight extension mechanism – we adapt the UML semantics without changing the UML metamodel, and using the heavyweight extension mechanism, where we adapt the UML semantics by extending the standard metamodel. The former mechanism is supported by UML/SysML through built-in mechanisms such as Stereotypes and Tagged Values. The latter approach would extend integrated visual modeling for ODL from a common modeling tool.

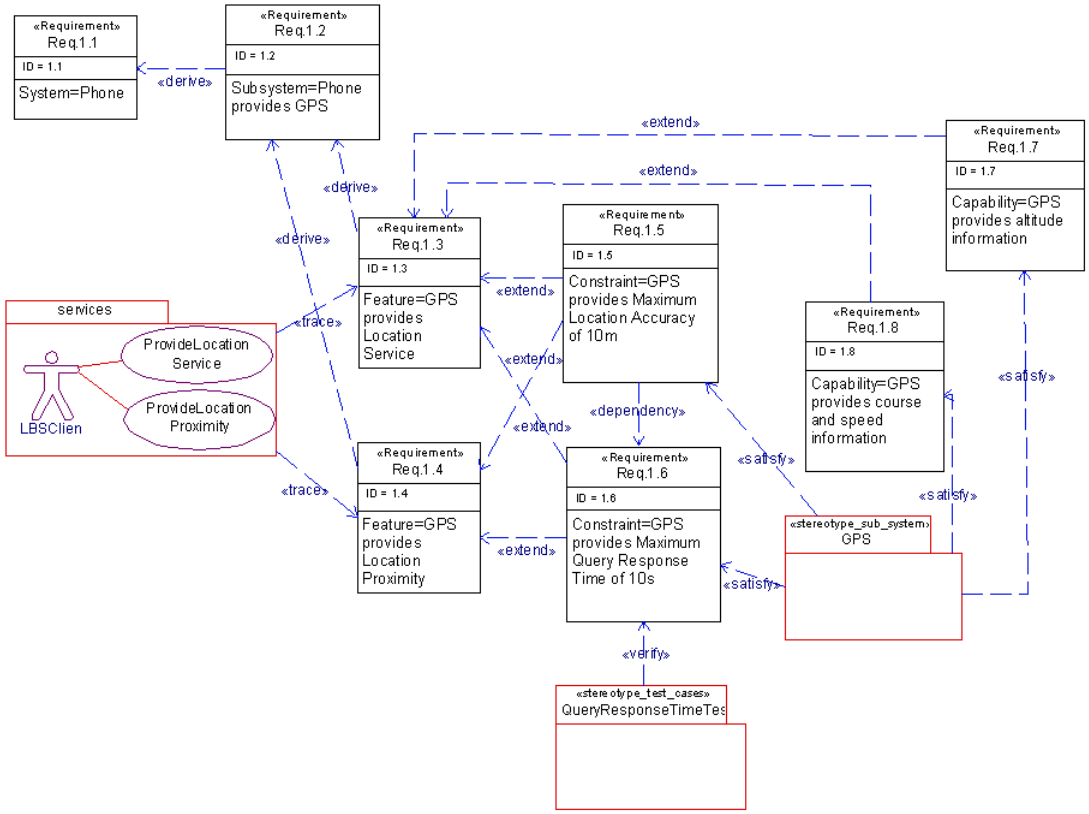

Figure 6. A SysML requirements diagram describing requirements and constraints for an LBS application corresponding to the ODL statements in Figure 3.

IV. Related Work

This section summarizes related work and provides pointers to more information. The work in [9] propose a Semantic Web approach for maintaining software, using the Resource Description Framework (RDF) for representing information related to software components, and the OWL for describing the relationships between those software components, metrics, tests, and requirements. Queries formulated in the SPARQL query language extract information about the state of the software system from the RDF knowledge base. The proposed solution deals only with software maintenance and does not address requirements specification/validation and UML design automation.

The work in [15] proposes a technique in which a specification is derived from a requirement. The author's approach is based on the problem frame concept 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 leads 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 [11], 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 proposed requirements analysis methodology. Using a domain ontology comprising domain specific concepts and relationships, and inference rules together with an interpretation function, the authors perform semantic processing of requirements models. This research is different from our approach, as the ontology class concepts Kayia et al. define belong to the application domain. The ODL ontology is more general and the RDDA framework as actually independent on the application domain.

Raven [2] 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 [5] is the acquisition and management of requirements using semantic web techniques. 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.

Similar to [5], the authors in [6] put forward means of taking advantage of Wikis to gain semantic knowledge of different information. In [6], 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 work in [7] 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++.

Kof proposes in [12] natural language processing methods for 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 the 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. The last stage is finding associations between those extracted terms.

V. Conclusions

In this paper we described 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.

These mechanisms are part of the RDDA framework that develops methodologies for system design automation from requirements. The framework has the necessary capabilities to be integrated with a SysML or UML modeling tool.

In the requirements specification and validation area, in the future we will improve the SysML requirements specification method, specifically, we will look into developing a new SysML profile for modeling requirements concepts. We will also continue to add new rules for consistency verification. A long-term goal is to integrate all tools for requirements specification, verification, and design synthesis into a common platform based on Eclipse.

Acknowledgments

This work could have not been possible without the generous support from Mr. Jaime Borras, Motorola Corporate Senior Fellow and Senior Vice President of Technology, and his colleagues from the Advanced Technology Group.

References

[1] Protége: ontology editor and knowledge-base framework. protege.stanford.edu.

[2] RAVEN: Requirements modeling tool from Ravenflow. http://ravenflow.com/.

[3] SWI Prolog. http://www.swi-prolog.org/.

[4] Vowlidator OWL Checker. http://projects.semwebcentral.org/projects/vowlidator/.

[5] Sören Auer and Klaus-Peter Fhnrich. SoftWiki – Agiles Requirements-Engineering fr Softwareprojekte mit einer groen Anzahl verteilter Stakeholder. In *Software Engineering 2006*, Leipzig, Germany, June 2006.

[6] Sören Auer and Jens Lehmann. What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. In *European Semantic Web Conference (ESWC'07)*, volume 4519, pages 503–517. Springer, LNCS, 2007.

[7] 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 *Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*, pages 51–58, 2003.

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

[9] D. Hyland-Wood, D. Carrington, and S. Kaplan. Enhancing software maintenance by using semantic web techniques. In *International Semantic Web Conference (ISWC)*, 2006.

[10] I-Logix/Telelogic. The Rhapsody UML Modeling Tool. http://www.ilogix.com/sublevel.aspx?id=53.

[11] Haruhiko Kaiya and Motoshi Saeki. Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach. In *Fifth International Conference on Quality Software (QSIC 2005)*, 2005.

[12] Leonid Kof. Natural Language Processing for Requirements Engineering: Applicability to Large Requirements Documents. In *Automated Software Engineering, Proceedings of the Workshops*, Linz, Austria, Sept. 2004.

[13] OMG. Omg systems modeling language. http://www.omgsysml.org/.

[14] OMG. The Unified Modeling Language. http://www.uml.org.

[15] Robert Seater, Daniel Jackson, and Rohit Gheyi. Requirement Progression in Problem Frames: Deriving Specifications from Requirements. *Requirement Engineering Journal (REJ)*, 12(2):77–102, 2007.

[16] W3C. The ontology web language. http://www.w3.org/2004/OWL.