 THE RESEARCH LEADING TO THESE RESULTS HAS RECEIVED FUNDING FROM THE EUROPEAN UNION'S SEVENTH FRAMEWORK PROGRAM (FP7/2007-2013) FOR CRYSTAL – CRITICAL SYSTEM ENGINEERING ACCELERATION JOINT UNDERTAKING UNDER GRANT AGREEMENT N° 332830 AND FROM SPECIFIC NATIONAL PROGRAMS AND / OR FUNDING AUTHORITIES.

CRitical S**YST**em Engineering **A**cce**L**eration

Architecture of the Tool Chain for the Multi-Mode Navigation System

D206.021

DOCUMENT INFORMATION

Project	CRYSTAL
Grant Agreement No.	ARTEMIS-2012-1-332830
Deliverable Title	Architecture of the Tool Chain for the Multi-Mode Navigation System
Deliverable No.	D206.021
Dissemination Level	RE
Nature	R
Document Version	V3.0
Date	2014-04-30
Contact	Tomáš Kratochvíla
Organization	HON
Phone	+420 532 115 530
E-Mail	Tomas.Kratochvila@honeywell.com

AUTHORS TABLE

Name	Company	E-Mail
Nikola Beneš	Honeywell	Nikola.Benes@honeywell.com
Jan Beran	Honeywell	Jan.Beran@honeywell.com
Vít Koksa	Honeywell	Vit.Koksa@honeywell.com
Tomáš Kratochvíla	Honeywell	Tomas.Kratochvila@honeywell.com

REVIEW TABLE

Version	Date	Reviewer
1.0	11.4.2014	Jan Beran (internal review)
2.0	29.4.2014	Anne Monceaux (external reviewer)
3.0	30.4.2014	Jiří Barnat (external reviewer)

CONTENT

Content of Figures

Content of Tables

1 Introduction

1.1 Role of deliverable

The purpose of this document is to define the architecture of the tool chain that will be used to develop multi-mode navigation system. The engineering methods are described in Crystal Deliverable D206.010.

1.2 Structure of this document

We have divided the tool chain architecture into 3 parts. Each part is described in its own chapter:

2 **Architecture of the Ontology Engineering Tool Chain** – creates and manages the domain ontology. This tool chain is used by domain ontology experts only and is not supposed to be done for every project, since many projects will share the same domain ontology.

3 **Architecture of the Development Tool Chain** – model-based development process tool chain is supposed to be used for every project. It uses the domain ontology for a given domain for initial requirement authoring. We use Honeywell model-based development called 3 View System Engineering process.

4 **Architecture of Verification Tool Chain** – verification and validation of the artefacts developed by the development tool chain. We are focused on automated formal verification.

2 Architecture of the Ontology Engineering Tool Chain

This tool chain is used by domain ontology experts only. The idea is that the domain ontology is created once for each domain and only limited work is needed to manage the domain ontology to be up to date with the current state of the domain. This process is not supposed to be performed for each project, since many projects will share the same domain ontology.

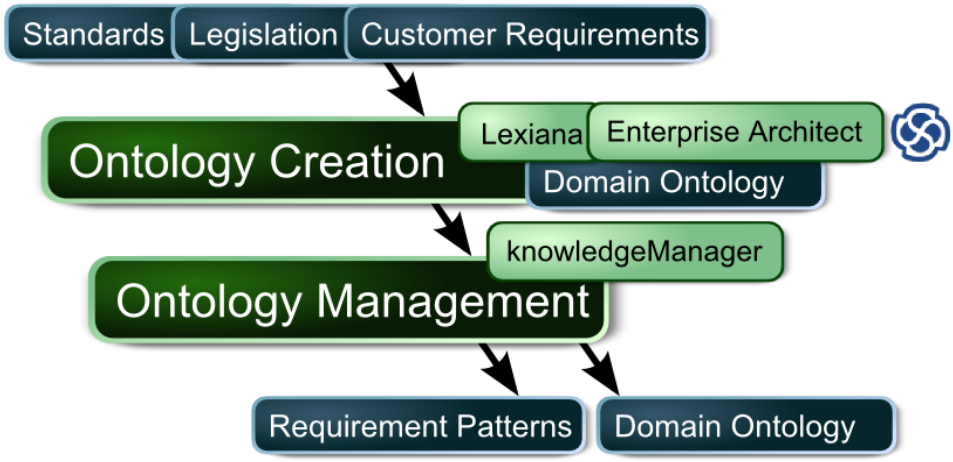

Figure 1 – Ontology Engineering Tool Chain

The tool chain will create ontologies for different domain granularities – from general ontology and aerospace ontology to specific domain ontologies (for example AHRS domain only). Since there are multiple approaches how to semi-automatically merge domain ontologies [Kotis, 2006], hence it will be possible to create exactly the ontology for any specific domain. For example, when the system under development integrates two subsystems into one, the ontologies for the two domains of the subsystems will be merged. Every domain ontology should have completely defined its domain using metadata showing which sources were used to create it. The general ontology might be used to show differences between more specific ontologies only.

Honeywell has a special tool suite consisting of Lexiana and Enterprise Architect scripts to semi-automatically derive requirements model from domain descriptions: standards, legislation, customer requirements, etc. Lexiana generates a list of suggested concepts and relations between the concepts from the domain descriptions. Then the domain ontology expert imports the requirement model into Enterprise Architect and makes the requirement model compact and consistent using several specialized Honeywell scripts. These compact requirement models will be then translated to the thesaurus.

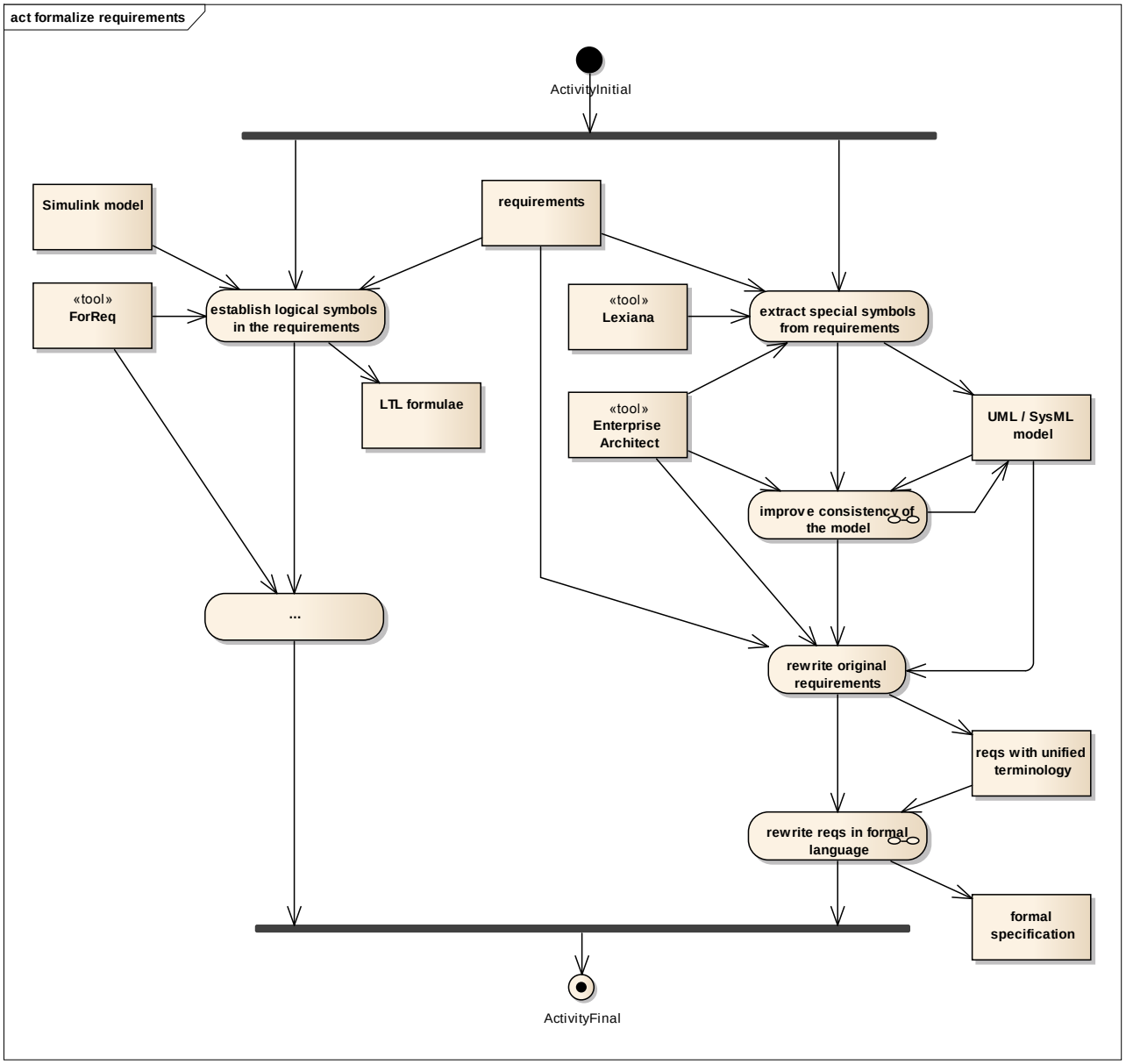

Figure 2 – Activity diagram of requirement formalization.

2.1 Lexiana

The extraction of interesting concepts from the text is supported by the tool Lexiana. This tool calls the Stanford Parser, which for English sentences generates their grammatical parsing trees. The grammatical parsing trees are processed by Lexiana. The output of Lexiana is an XMI file with the UML model of the extracted information, i.e.

- classes (their colours indicate the abundance of the term in the text – from green, which corresponds to 1 occurrence, to magenta, which corresponds to the most occurrences)

- packages of classes (package sizes suitable for further manual processing, packaging based simply on the initial letters of the class names)
- tentative relations between the classes,
- original sentences,
- realization relations from classes to sentences (this kind of traces is useful throughout the evolution of the model, but can also help to easily make the original text more consistent by using a unified terminology, see the SQL script copyDependenciesWithSentences below),
- diagrams with the most frequent concepts,
- diagrams with concepts whose names contain common substrings.

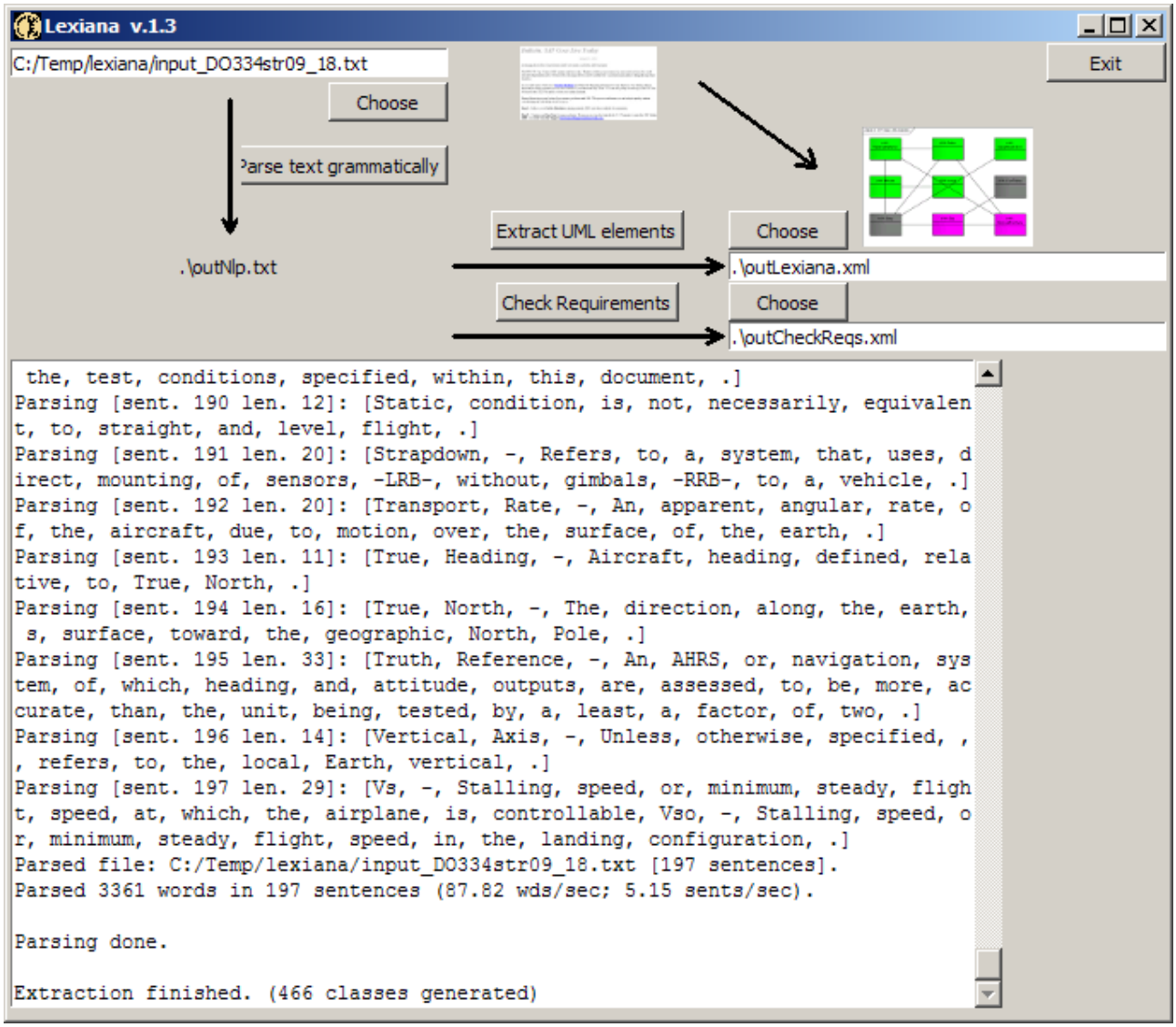

Figure 3 – The GUI of Lexiana tool.

The classes are generated from certain noun phrases. For each sentence, the anonymous undirected relationships are generated between each two classes contained within the sentence. Most of these tentative relationships are usually removed later during the manual processing of the model.

2.2 Enterprise Architect Scripts

The Enterprise Architect (EA) is a robust tool for UML or SysML modelling. The requirements model generated by Lexiana is imported into EA. The generated UML model (the imported elements and diagrams) is a good starting point and spares time of the analyst, but this model still usually needs substantial improvements. The creation of a mature ontology from the original model is facilitated by several utilities, which were incorporated as either SQL scripts or automation scripts within the Enterprise Architect. Some of these scripts provide useful information on the model, while other scripts are called to update the model. The scripts can speed up some activities by orders of magnitude, remove tedious work and help to avoid omissions.

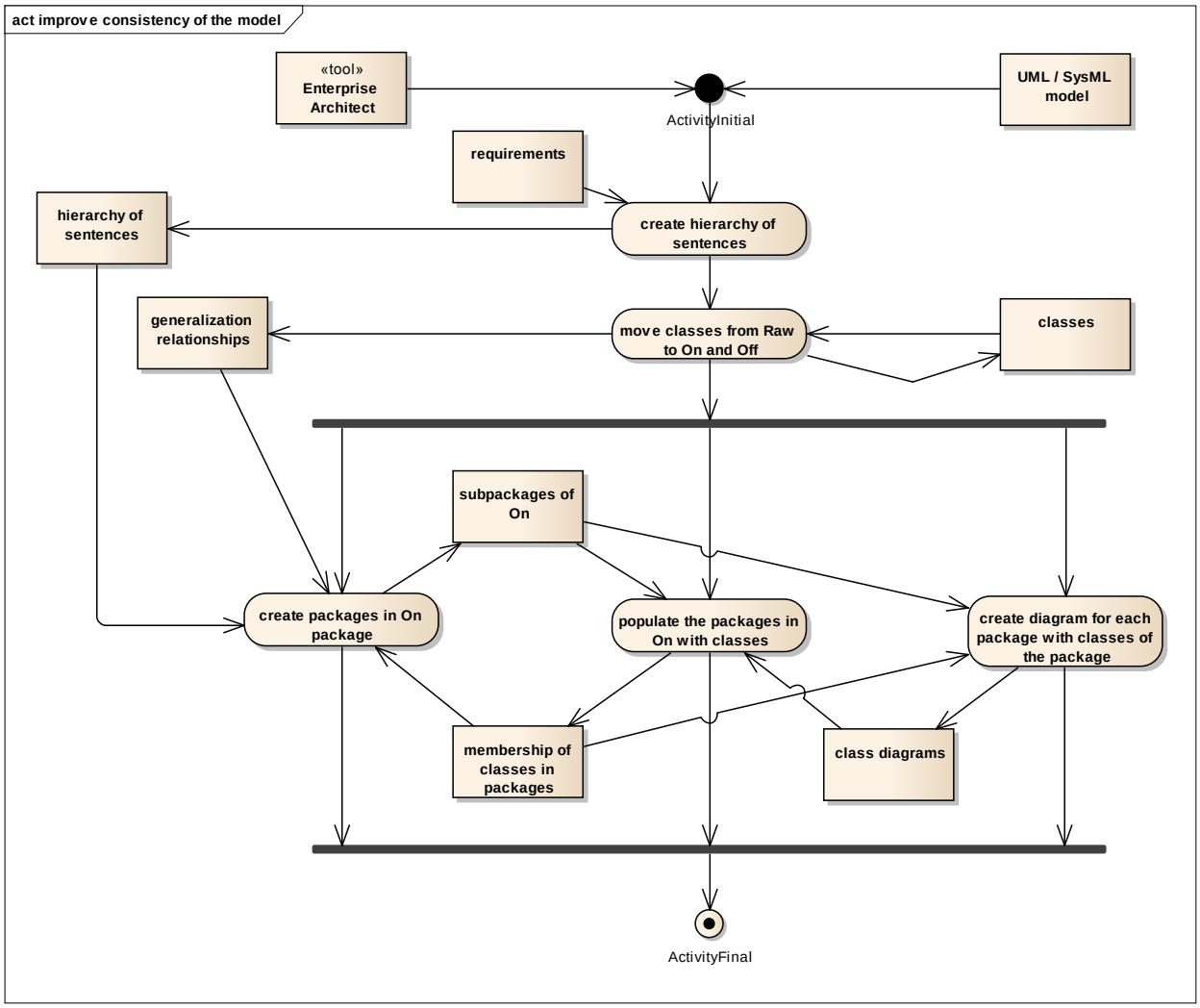

Figure 4 – Creation of the thesaurus in the Enterprise Architect

In the figure above the following activities are depicted:

- **Create hierarchy of sentences** – the requirements are packaged according to the original document structure.
- **Move classes from Raw to On and Off** – originally, all classes are under the package Raw; move them to either On (important classes) or Off (thrown out classes). The most useful automation scripts (see below) are Lxn-copyLinks for merging or splitting classes, and Lxn-listReqsOfClass for listing the related requirements. The generated “stem” diagrams are indispensable for this activity, as they display together the classes with common substrings.
- **Create packages in On package** – usually there is too much classes in the On package. Subpackages can be e.g. generated by the Lxn-mirrorReqPackageToObjPackage automation script (see below), or created around classes which have numerous subclasses.
- **Populate the packages in On with classes** – when the structure of packaging was mirrored from the packaging of the requirements, the Lxn-addInhabitants automation script can help to move the relevant classes to the appropriate package.
- **create diagram for each package with classes of the package** – place the classes of the package into the diagram of the package. Use the scripts Lxn-addTentativeNeighbours, Lxn-addNeighboursOfClass, Lxn-addSuperclassesOfClass to populate the diagram.

2.2.1 SQL scripts

Most of the SQL scripts are generally usable. The SQL scripts query the database with the model. No modifications of the database content are allowed via this mechanism.

Example of what the **copyDependenciesWithSentences** script produces is in shown in the following figure.

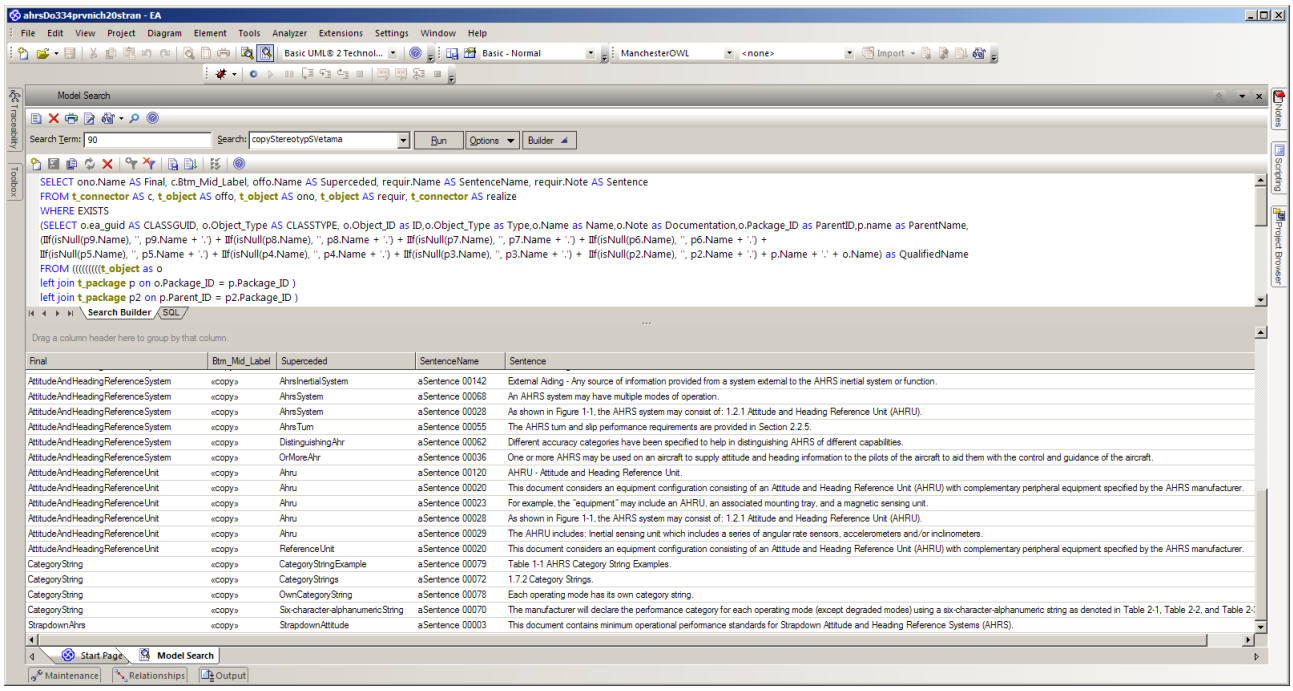

Figure 5 – output of copyDependenciesWithSentences script.

These scripts were written to support the work with the ontology:

- **Attributes** – lists all attributes in the model containing a given string (passed to the script as the Search Term).
- **ClassesExceptOff** – for a given string lists all matching names of classes in the model with the exception of the classes contained in the "Off" package.
- **Connectors** – lists all associations in the model whose name matches the given string.
- **findPackageByName** – lists package_ID for all packages with the given name (the model can contain packages with the same name and sometime it is necessary to point to a specific one via the package_ID)
- **classesInSubpackages** – lists all the classes contained in the package given by its package_ID or in any of its subpackages.
- **qualifiedClassesInSubpackages** – dtto, + qualified class name (e.g. Model.Domain Model.PackageLevel1.PackageLevel2.ClassName).
- **mandatoryRequirements** – lists all requirements whose Status is "Mandatory".
- **copyDependenciesWithSentences** – (Lexiana related) for classes in the given package (package_ID) the script lists the superceded "synonyms" of the classes and the sentences, where these superceded terms were present.

Note:

As the SQL implementation does not support transitive closure of the relationships, some scripts are limited in the depth of nesting (classesInSubpackages, qualifiedClassesInSubpackages, copyDependenciesWithSentences).

2.2.2 Automation scripts

The automation (javascript) scripts are primarily intended to automate the work with ontologies that are generated by Lexiana. Some of them might be useful outside of this context, e.g. Lxn-addSuperclassesOfClass, Lxn-hideForeignRelations, Lxn-addNeighboursOfClass, Lxn-underspecifiedRelations. These scripts usually retrieve or modify information about the model.

Example: what the Lxn-1_listCommonReqs script writes in the Output pane is shown in the following figure. In fact, some things contained in the figure are the result of applying several other scripts listed below.

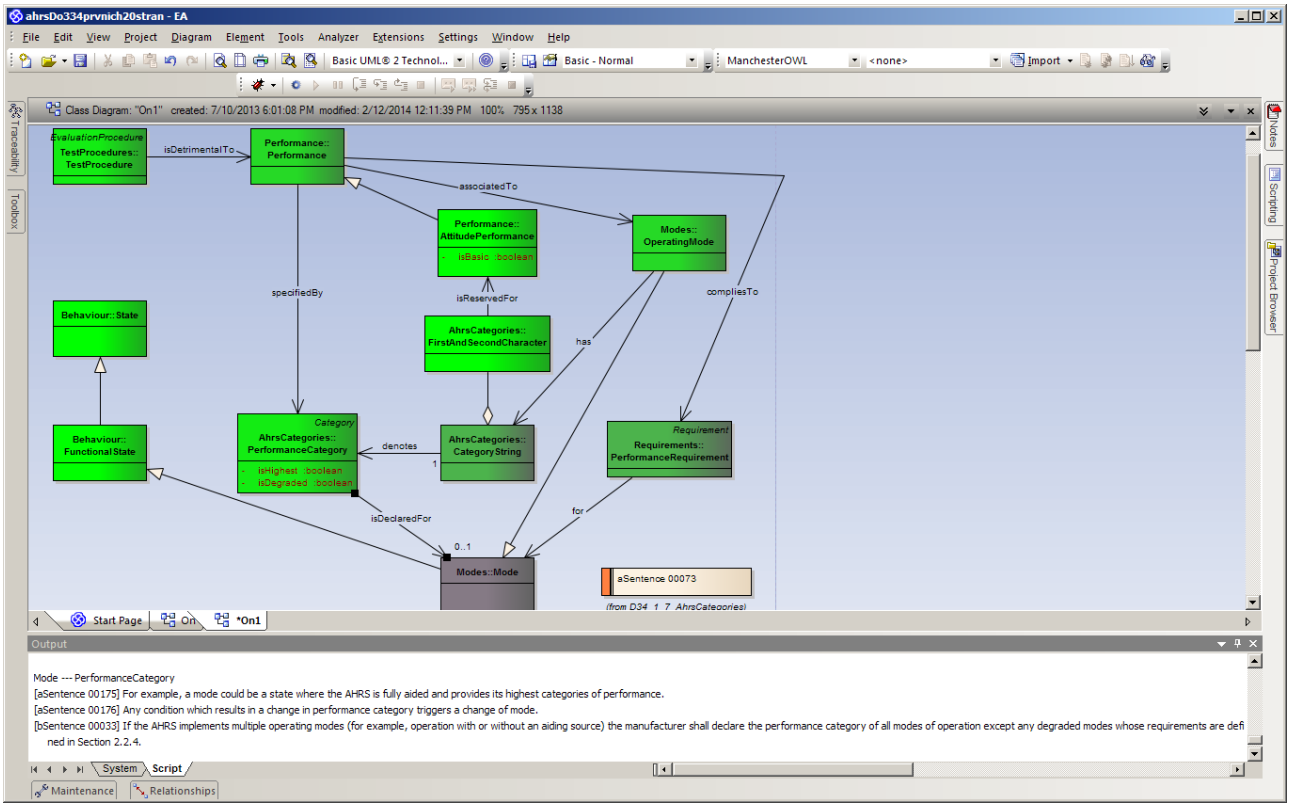

Figure 6 – Work with automation scripts in Enterprise Architect.

- **Lxn-1_listCommonReqs** – (used in diagram) for the selected association writes all requirements which contain both classes related by the association.
- **Lxn-2_listCommonReqsDepth** – (diagram) as above, but considers also all combinations of subclasses of the two associated classes (explosion prevented by the limitation to just the children and grandchildren).
- **Lxn-addInhabitants** – (diagram) for the selected requirement(s) adds all classes to the diagram which Realize the requirement.

- **Lxn-addNeighbours** – (diagram) for the current package adds classes to the diagram, which are related to the classes in the package.
- **Lxn-addNeighboursOfClass** – (diagram) for the selected class(es) adds all classes to the diagram which are related by some kind of association to the selected class.
- **Lxn-addPrefix** – (used in Project Browser) .. changes names of requirements contained in the selected package, e.g. "Sentence 00001" -> "bSentence 00001"
- **Lxn-addSuperclassesOfClass** – (diagram) for the selected class(es) it adds all their superclasses (recursively to the topmost classes) to the diagram.
- **Lxn-addTentativeNeighbours** – (diagram) special case of what Lxn-addNeighbours does.
- **Lxn-copyLinks** – (Project Browser) package selected, for each class in the package, if there is a <<copy>> dependency from the class, the relationships of the class are all copied to the target of the <<copy>> link, then the stereotype <<toBeRemoved>> is added to the source class.
- **Lxn-cutOff** – (Project Browser) for the selected package, for each class in the package each link is removed if it leads to a class contained in an "Off" (trashbin) package.
- **Lxn-hideForeignRelations** – (diagram) hides all relations between classes where neither of the two related classes belongs to the current package.
- **Lxn-listReqsOfClass** – (diagram) for the selected class related requirements are reported in the Output pane.
- **Lxn-mirrorReqPackageToObjPackage** – (Project Browser) creates new packages and distributes the classes into the packages depending on the packaging of requirements within a selected package.
- **Lxn-underspecifiedRelations** – (Project Browser) list unspecified (anonymous) relationships of classes within the selected package and its subpackages.

2.2.3 Traceability view

The Enterprise Architect can provide a lot of different views on the model data. The Traceability view proved to be especially useful while working with the UML ontology. This view provides a similar presentation of the data and relationships between them as can be seen in ontology tools (e.g. Protege). The elements contained in the view can be opened or added to the current diagram.

The diagram in Figure 7 was easily created from the Rotorcraft class by using the Lxn-addSuperclassesOfClass automation script.

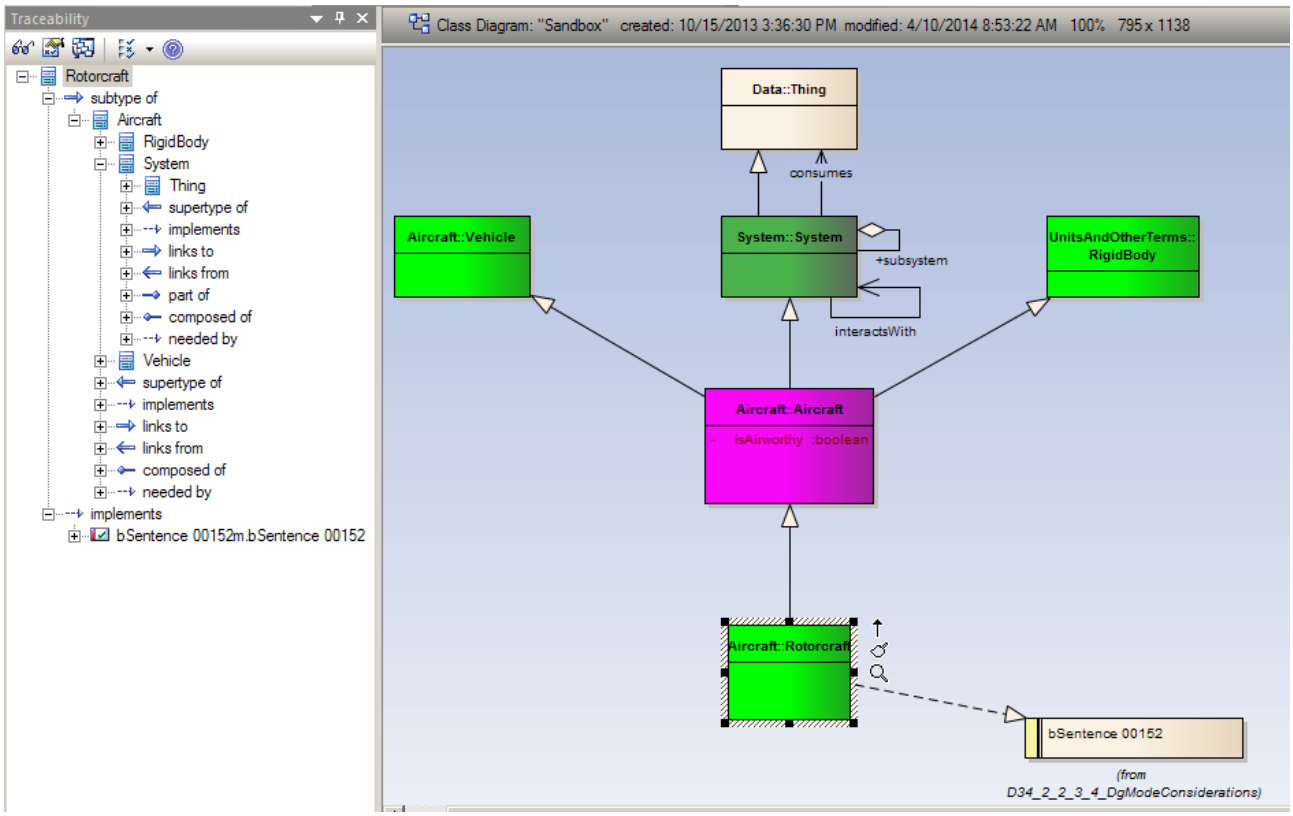

Figure 7 – Traceability view on the left and the corresponding elements in the diagram.

2.2.4 Example

An ontology in the Distributed Ontology Language (DOL) consists of modules formalised in basic ontology languages, such as OWL (based on description logic) or Common Logic (based on first-order logic with some second-order features). These modules are serialised in the existing syntaxes of these languages in order to facilitate reuse of existing ontologies. DOL adds a meta-level on top, which allows for expressing heterogeneous ontologies and links between ontologies. Such links include (heterogeneous) imports and alignments, conservative extensions (important for the study of ontology modules), and theory interpretations (important for reusing proofs). Thus, DOL gives ontology interoperability a formal grounding and makes heterogeneous ontologies and services based on them amenable to automated verification. [Lange, 2012].

Lexiana extracted classes from the text of the paragraph in italics above and generated UML diagrams automatically. This is the diagram with all the extracted classes:

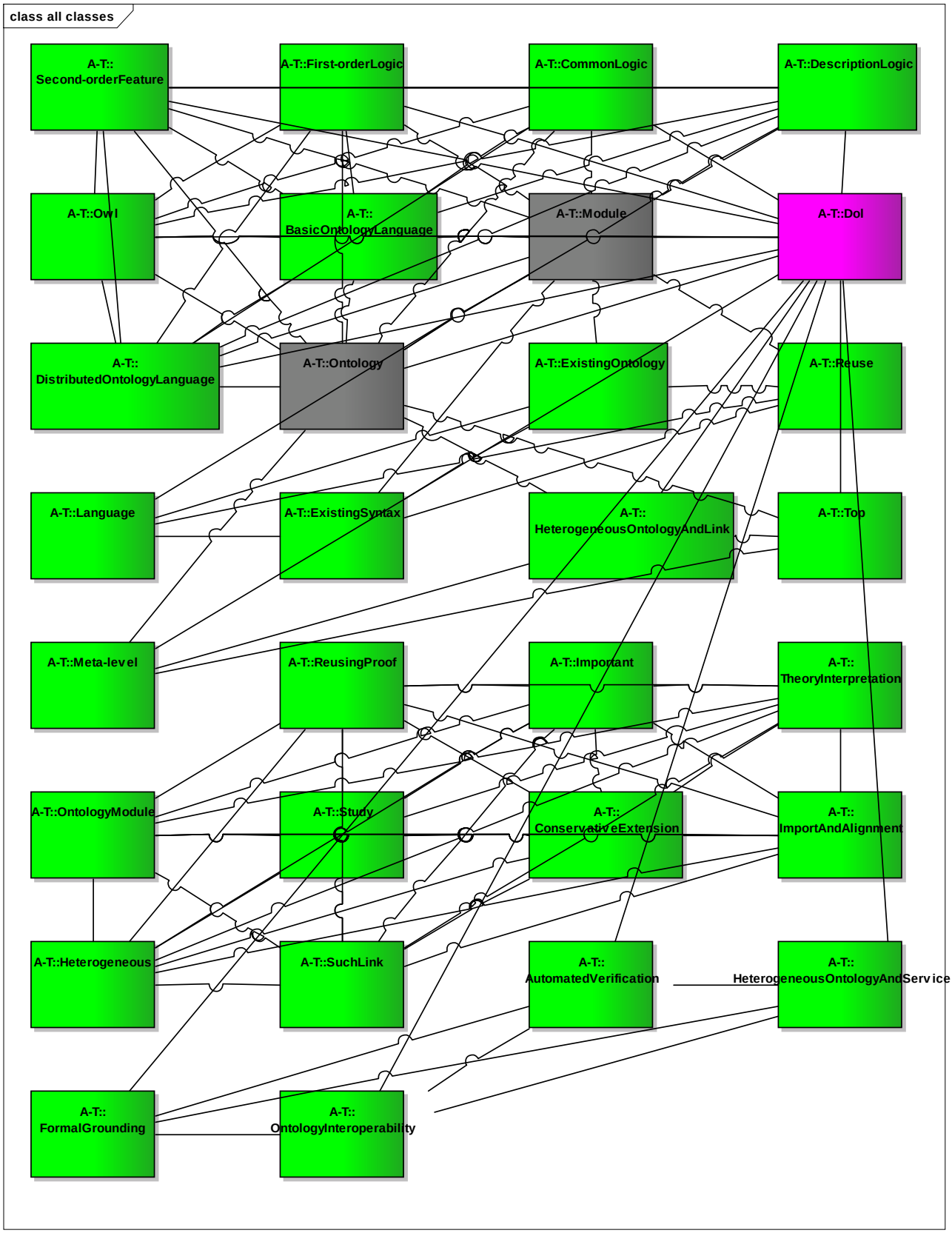

Figure 8 – A diagram generated by Lexiana as shown in the Enterprise Architect.

After manual changes (which took less than 2.5 man-hours) supported by the other generated diagrams and available scripts the UML ontology depicted on the following two figures was obtained:

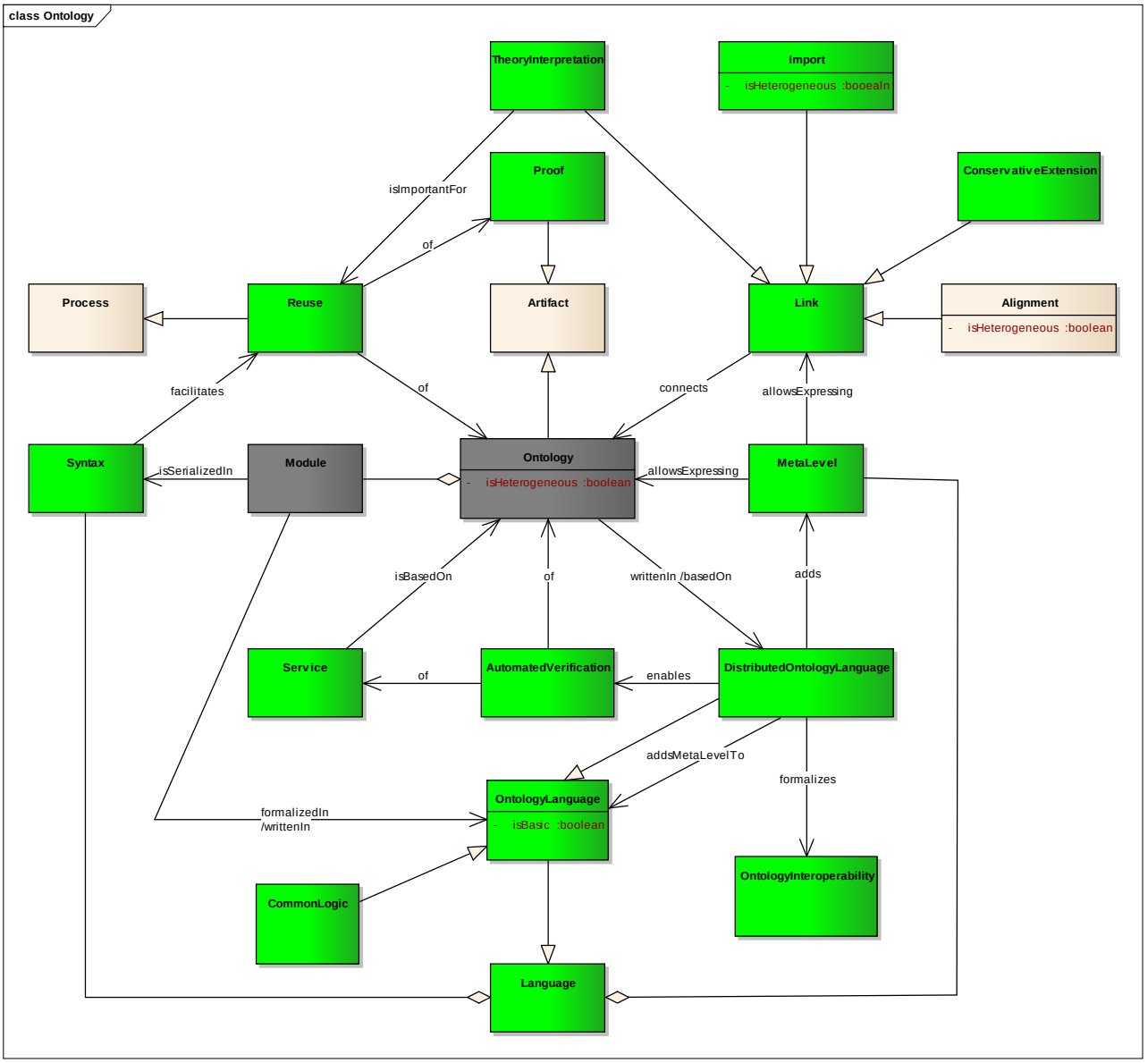

Figure 9 – Resulting UML ontology after manual modifications, part 1.

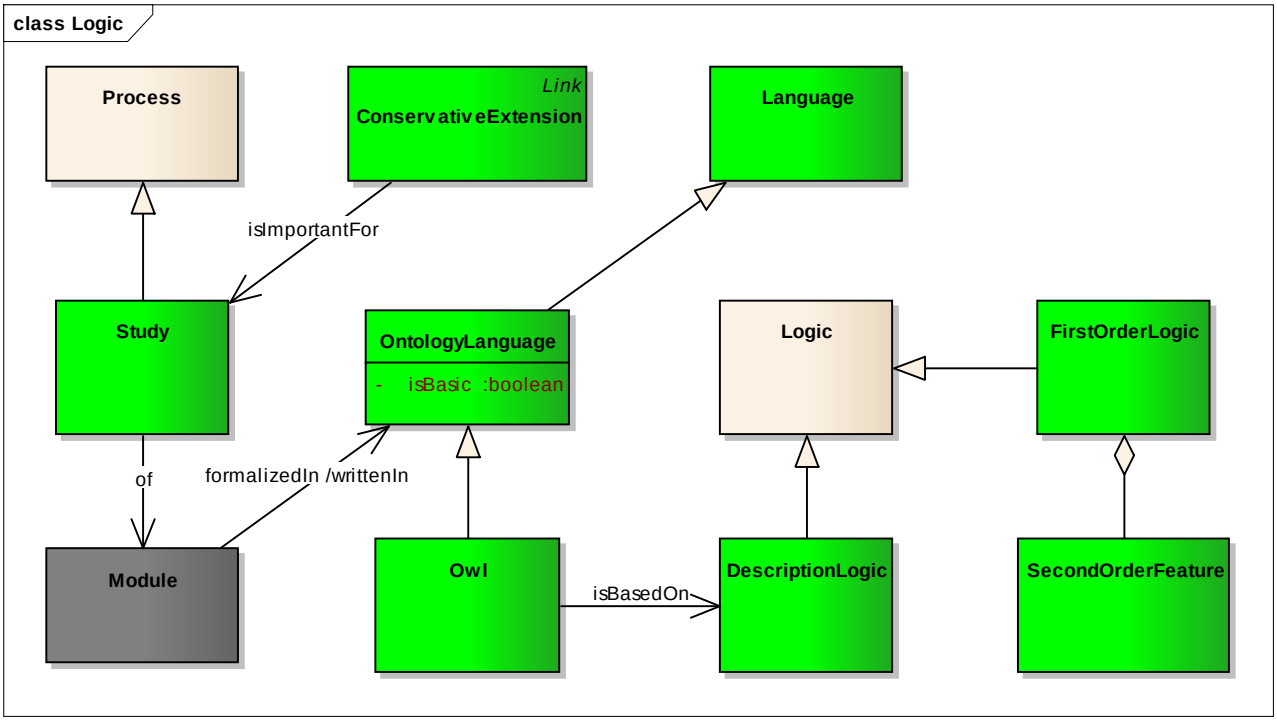

Figure 10 – Resulting UML ontology after manual modifications, part 2.

2.3 knowledgeMANAGER

Thesaurus generated from Enterprise architect using Honeywell scripts will be imported to knowledgeManager. Thesaurus should conform to ISO 25964 and ANSI/NISO Z 39.19:

http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53657

http://www.niso.org/apps/group_public/download.php/12591/z39-19-2005r2010.pdf

However, there are some differences. For example ANSI/NISO Z 39.19 defines that the following abbreviations BTP (broader term partitive) and NTP (narrower term partitive) **may** denote hierarchical whole-part relationship while knowledgeManager uses WH (whole) and PA (part) abbreviations instead.

Thesaurus file (.the) contains terms and its relationships. For example:

Airplane
PA: Engine
PA: Wings
PA: Navigation System
Navigation System
NT: Inertial Navigation System

List of the thesaurus (.the) file terms and its abbreviations we plan to use:

BT	Broader term (used for generalization)
NT	Narrower term (used for generalization)
PA	Part (used for aggregation)
WH	Whole (used for aggregation)
other	User defined terms

Other terms could be listed in knowledgeManager: Domain Management -> Light Ontology -> Semantics:

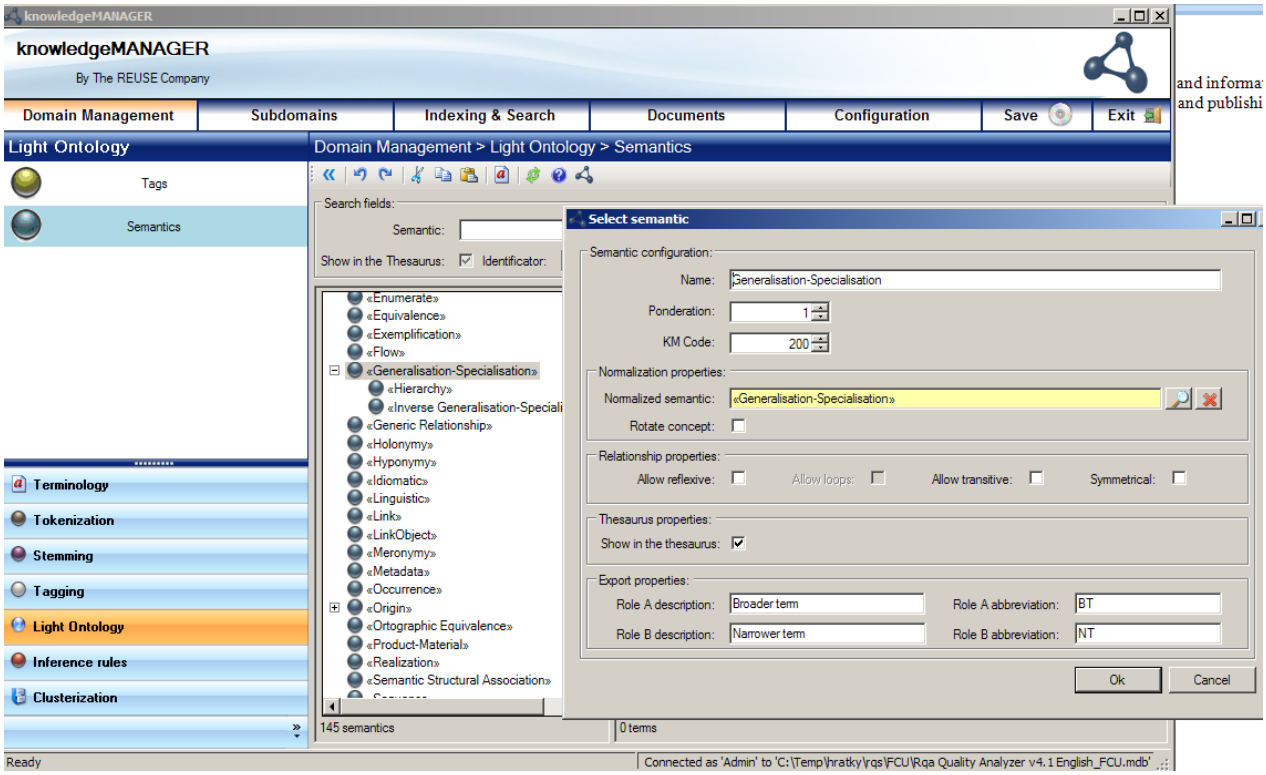

Figure 11 – Thesaurus semantics of the relationships between terms.

Since this process is supposed to be performed only once per domain and by ontology experts only, hence there is no need for tight integration like in development or verification process.

There are several technical issues, which we will have to solve together with REUSE Company in order to get integrated tool chain. Some of the issues were already solved and the affected tools were fixed during our cooperation within Crystal project. The main issues are:

1. The knowledgeManager supports repetitive rules for defining requirement patterns only and grammatical recursion rules cannot be used. Therefore we cannot capture requirement pattern in the form for example: P :== not P | P and P | P or P | term.

This is needed for example to define any logical/arithmetical propositions or any general requirement pattern.

2. While Honeywell Lexiana tool and the UML ontology support multiple relations between two terms, the knowledgeManager supports just one relation. There are many terms even in AHRS domain, which have multiple relations. For example:
    a. Algorithm consumes Data vs. Algorithm produces Data
    b. System is under Condition vs. System detects Condition
    c. System derives Output vs. System invalidates Output
    d. System transitions to Mode vs. System implements Mode vs. System operates in Mode
    e. Equipment requires Performance vs. Equipment provides Performance
    f. Equipment Manufacturer reads Equipment Design vs. Equipment Manufacturer modifies Equipment Design
    g. Manufacturer specifies Data vs. Manufacturer invalidates Data
    h. Time Period starts at Time vs. Time Period ends at Time.

The knowledgeManager will be used to create requirement patterns (boilerplates) suited for each domain. The requirement patterns will be used in the development tool chain described in the next chapter.

3 Architecture of the Development Tool Chain

This Chapter describes the envisioned architecture of the development tool chain. The engineering methods and tools are depicted in Figure 12 in green boxes. Input and output artefacts are denoted in blue boxes.

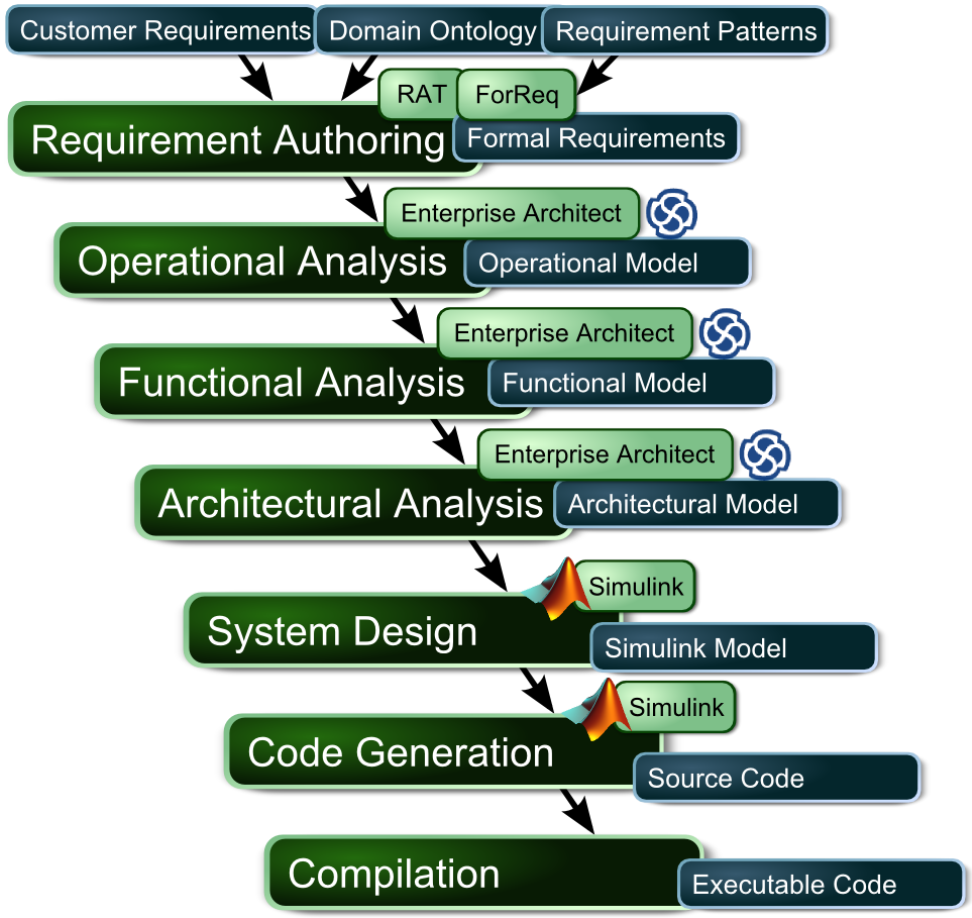

Figure 12 – Development and verification tool chains together.

There will be only small change in development process we propose: requirement authoring will be done using domain ontology and formal requirements will be created in order to enable automatic formal verification and to remove requirement ambiguity.

From the ontology engineering tool chain we expect to get domain ontology and requirement patterns. Afterwards, Requirement Authoring Tool (RAT) can be used to write requirements based on the requirement patterns. ForReq tool (described in Crystal Deliverable D206.010) will formalize the requirements and run automatic verification as described in chapter 4.

Since we provide formal verification of behavioural requirement (functional requirements which define input-output behaviour of the system), hence we need to know which requirement pattern was used to write these requirements. The reason why we need to know which requirement pattern was used is that the behavioural requirement needs to be

mapped to some temporal logic (for example Linear Temporal Logic) so that the requirement can be automatically formally verified.

There are a few options how to achieve the mapping:

1. Improve RAT tool with the functionality that it can export both requirement and its requirement pattern (for at least certain types of requirement patterns).
2. Extend ForReq tool to enable context-aware auto-completion of requirements based on requirements patterns so that it can substitute RAT functionality completely.
3. Improve ForReq tool with the requirement pattern recognition system. Since some requirements might be written based on multiple different requirement patterns, hence this might not be possible at all.

Moreover, in the system design phase we need to map the formal requirement to the design artefacts (variables, states, etc.) so that it can be automatically verified. There are two options how to achieve this mapping:

1. Export design artefacts (variables, states, etc.) from ForReq to extend current domain ontology so that RAT can guide the user to use actual system entities while writing requirements. The problem with this approach is that knowledgeManager does not currently support recursive definition in requirement patterns.
2. Extend ForReq tool to enable context-aware auto-completion of requirements based on requirements patterns so that it can substitute RAT functionality completely.

When the formal requirements are derived the development process continues with Honeywell Three View System Engineering process which is model based system engineering which created 3 views of the system: operational, functional and architectural model.

At the end, the system design is created in the form of Simulink model and executable code is automatically generated using Honeywell Autocode Manager and Simulink Real-Time Workshop.

4 Architecture of Verification Tool Chain

Verification tool chain will verify the formal requirements provided in Requirements Interchange Format (ReqIF) and system design in Simulink model format.

Since Open Services for Lifecycle Collaboration (OSLC) was selected as a cornerstone technology for tool integration, we are proposing integration architecture in our limited scope and will cooperate with SP6 on Crystal-wide common integration approach.

The proposed OSLC integration technology will enable easy interoperability and exchangeability of the tools if necessary.

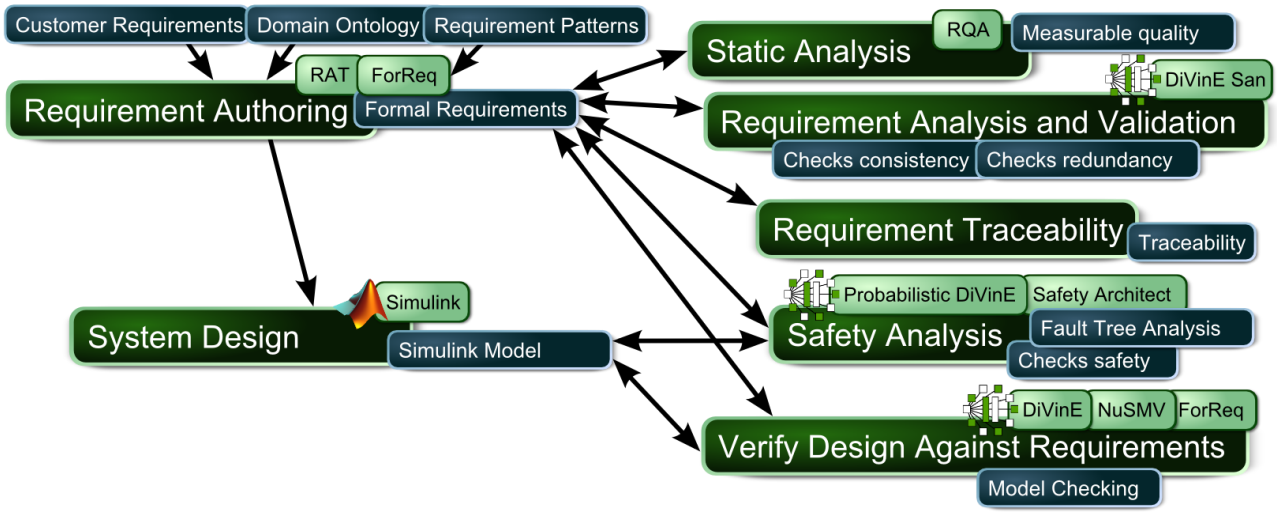

Figure 13 – Verification tool chain architecture with the verification methods.

Static Analysis will be performed by Requirement Quality Analyzer tool from Reuse in order to measure the requirement quality and make sure that the requirements conform to requirement standards. Since RQA is tightly integrated with RAT there will be no need for additional integration effort.

DiVinE Sanity Checking will for all formalized requirements check its sanity, which is composed of two formal verification techniques:

- Consistency checking determines if each requirements subset could be satisfied by some abstract system.
- Redundancy checking determines for each pair of requirements if one requirement does not imply the other. In that case the other requirement is redundant.

Complete description of the sanity checking approach is in [Barnat, 2013].

Requirement Traceability shall be provided by ForReq tool. The formal requirements have direct traces to the system design (in order to enable automatic verification) and also to legacy requirements if any. Also, formal requirements have traces to its formal specification for example in the form of Linear Temporal Logic. Moreover, each verification result will have traces to the requirements and system design.

Safety Analysis will consider the system design extended with an error model and fault injections specification. This extended model will be used to perform minimal cut-set analysis, fault trees creation, and probabilistic analysis. The probabilistic analysis will be done by Probabilistic DiViNE. Verification of the extended model will be used to assess correctness of Fault Detection, Fault Isolation and Recovery (FDIR) subsystems.

Verify Design against Requirements will for each fully formalized and selected requirement call available verification tools (explicit-state model checker DiVinE, symbolic model checker NuSMV, data-symbolic model checker DiVinE, test case generation tool) to automatically verify it on the provided system design. Complete description of the verification method is in [Barnat, 2012].

4.1 Automated Verification

The automated verification tasks will be executed on automation servers in parallel which will be integrated using OSLC with the ForReq tool. Therefore, multiple verification tasks will be executed in parallel on multiple automation servers. This will greatly speed up the verification process. The only verification task that cannot be automated is static analysis using RQA. The architecture should support also multiple ForReq client tools to perform independently and compete for the resources of available automation servers.

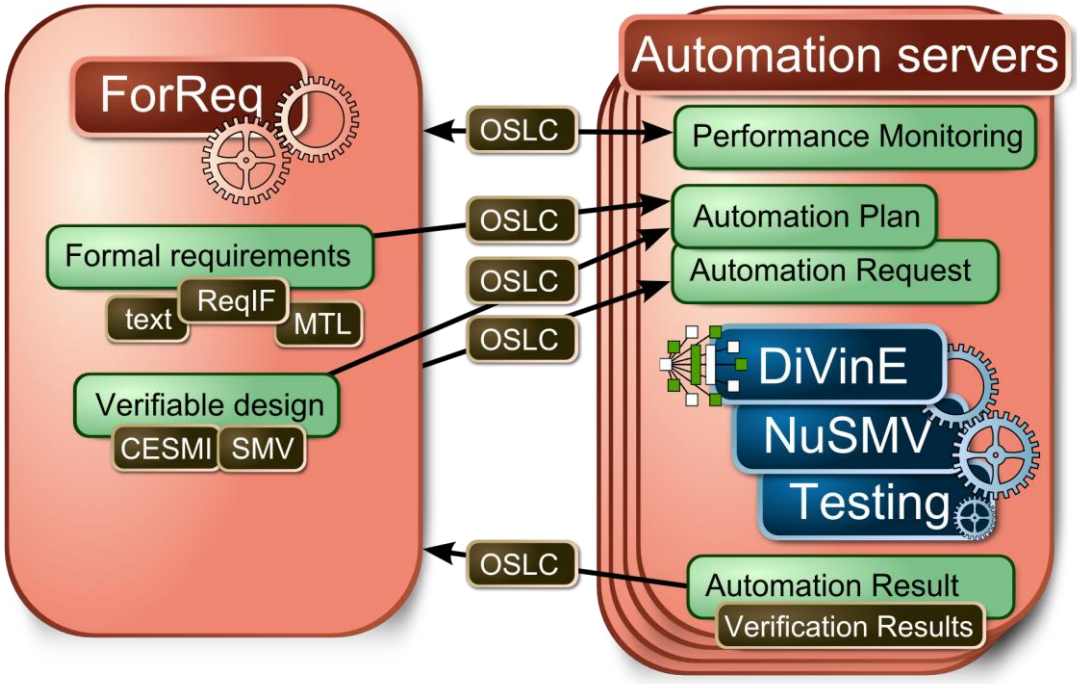

Figure 14 – Integration of ForReq tool and automation servers based on OSLC.

4.1.1 Automation Plan, Request and Result

The ForReq tool will for each verification task that can be automated create an OSLC automation plan and request and send it to best available automation server. Than the verification tools executes the automation plan and sends back the automation results which are reported to the user.

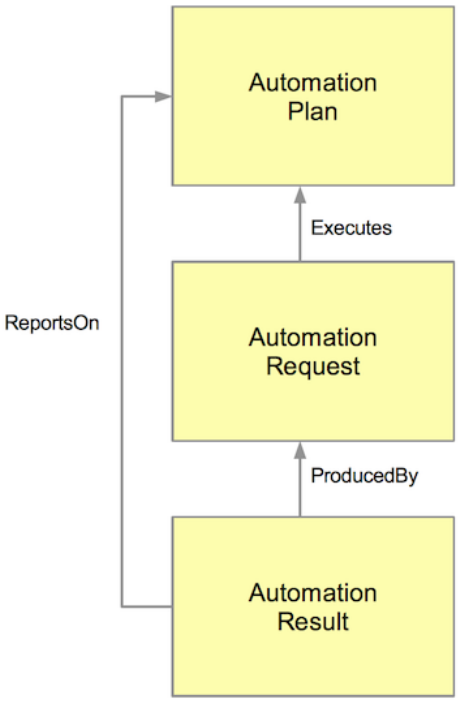

Figure 15 – Relationships among of the OSLC automation specification resources.

Complete specification of the OSLC automation can be found at:

http://open-services.net/wiki/automation/OSLC-Automation-Specification-Version-2.0/

4.1.2 Automation Server Performance Monitoring and Selection

The ForReq tool will have a database of automation servers and will for each verification task select the best automation server. It will choose the server with the most unused CPU and real Memory based on the estimation needs for given verification task, from servers which have the verification tool ready.

To enable this functionality, ForReq will monitor the automation servers using OSLC Performance Monitoring Specification:

http://open-services.net/wiki/performance-monitoring/OSLC-Performance-Monitoring-Specification-Version-2.0

For example this OSLC Resource: Performance Monitoring Record will notifies ForReq that the verification server has 3152 MB of available memory and more than 7 available CPUs:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        xmlns:dcterms="http://purl.org/dc/terms/"
        xmlns:rddl="http://www.rddl.org/"
        xmlns:qudt="http://qudt.org/1.1/schema/qudt"
```

```
        xmlns:pm="http://open-services.net/ns/perfmon#"
        xmlns:ems="http://open-services.net/ns/ems#">

  <rdf:Description rdf:about="http://server.address/perf#mem">
    <rdf:type rdf:resource="http://open-services.net/ns/ems#Measure" />
    <dcterms:title>Real Free Memory</dcterms:title>
    <ems:metric rdf:resource="pm:RealMemoryUsed" />
    <ems:unitOfMeasure rdf:resource="dbp:MegaByte" />
    <ems:numericValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
    3152</ems:numericValue>
  </rdf:Description>
  <rdf:Description rdf:about="http://server.address/perf#cpuu">
    <ems:numericValue
rdf:datatype="http://www.w3.org/2001/XMLSchema#double">157.34</ems:numericValue>
    <ems:unitOfMeasure rdf:resource="dbp:Percentage"/>
    <ems:metric rdf:resource="http://open-services.net/ns/perfmon#CpuUsed"/>
    <dcterms:title>CPU Utilization</dcterms:title>
    <rdf:type rdf:resource="http://open-services.net/ns/ems#Measure"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://server.address/perf#cpus">
    <ems:numericValue
rdf:datatype="http://www.w3.org/2001/XMLSchema#double">8</ems:numericValue>
    <ems:unitOfMeasure rdf:resource="dbp:quantity"
    <ems:metric rdf:resource="http://open-services.net/ns/perfmon#Cpus"/>
    <dcterms:title>Number of CPU</dcterms:title>
    <rdf:type rdf:resource="http://open-services.net/ns/ems#Measure"/>
  </rdf:Description>
</rdf:RDF>
```

5 Conclusion

We have described the architecture of the tool chain divided into 3 logical parts. We have shortly described functions of individual technology bricks which we plan to integrate into the tool chain. We have concentrated on inputs and outputs of each technology brick in order to ensure robust tool integration.

We have proposed a few possible options for the requirement authoring architecture. The final approach will depend on the results of our cooperation with REUSE Company. We are very demanding on the functionality of their requirements technology bricks in order to be sure that we can use the full potential of domain ontology engineering.

We have proposed tool integration based on Open Services for Lifecycle Collaboration (OSLC) as a selected cornerstone technology for tool integration. A tight cooperation with the technology bricks providers is a must order to achieve the fully functional tool chain.

6 Terms, Abbreviations and Definitions

3VSE	**Three View System Engineering – Honeywell model-based development process**
AHRS	**Attitude and Heading Reference System**
BTP	**Broader Term (Partitive)**
CTL	**Computational Tree Logic**
DiVinE	**Distribute Verification Environment – model checker from Masaryk University**
DODT	**Tool that semi-automatically transforms natural language requirements into semi-formal boilerplate requirements using domain ontology.**
EA	**Enterprise Architect – a tool from SPARX Company**
FDIR	**Fault Detection, Fault Isolation and Recovery Techniques**
ForReq	**Formalization of Requirements – internal Honeywell tool**
FTA	**Fault Tree Analysis**
GNSS	**Global Navigation Satellite Systems**
GUI	**Graphical User Interface**
IRS	**Inertial Reference Systems**
kM	**knowledgeManager – a tool from REUSE Company**
LTL	**Linear Temporal Logic**
NRP	**Narrower Term (Partitive)**
OSLC	**Open Services for Lifecycle Collaboration**
PA	**Part**
ReqIF	**Requirement Interchange Format (adopted as formal specification by OMG)**
RAT	**Requirement Authoring Tool – a tool from REUSE Company**
RQA	**Requirement Quality Analyzer – a tool from REUSE Company**
San	**Sanity checking of requirements (consistency, vacuity and completeness)**

SQL	**Structured Query Language**
SysML	**Systems Modelling Language**
UML	**Unified Modelling Language**
WH	**Whole**
XMI	**XML Metadata Interchange**

Table 6-1: Terms, Abbreviations and Definitions

7 References

[Barnat, 2012]	*Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs.* **J. Barnat and J. Beran and L. Brim and T. Kratochvila and P. Rockai**: *Formal Methods for Industrial Critical Systems (FMICS 2012)*, Springer, 2012, volume 7437 of LNCS, 78-92.
[Barnat, 2013]	*Checking Sanity of Software Requirements.* **Barnat, Jiří, Bauch, Petr and Brim, Luboš.** Brno: 2013.
[Kotis, 2006]	*Towards Automatic Merging of Domain Ontologies: The Hcone-Merge Approach.* **Konstantinos Kotis, George A. Vouros, Konstantinos Stergiou .** *Web Semant.* 4, 1 (January 2006), 60-79.
[Lange, 2012]	*LoLa: A Modular Ontology of Logics, Languages, and Translations.* **Christoph Lange, Till Mossakowski, and Oliver Kutz**: 2012, *http://www.informatik.uni-bremen.de/~till/papers/womo2012.pdf*