Grant Agreement nº	732463
Project Acronym:	OpenReq
Project Title:	Intelligent Recommendation Decision Technologies for Community-Driven Requirements Engineering
Call identifier:	H2020-ICT-2016-1
Instrument:	RIA (Research and Innovation Action)
Topic	ICT-10-16 Software Technologies
Start date of project	January 1st, 2017
Duration	36 months

D5.2: Requirements Dependency Engine Version 1

Lead contractor: UH

Author(s): UH, UPC

Submission date: December 2017

Dissemination level: PU

Project co-funded by the European Commission under the H2020 Programme.

Abstract: A brief summary of the purpose and content of the deliverable.

OpenReq is a project that aims to enhance requirements engineering activities. This document describes the first version of the services implementing the requirements knowledge representation and dependency management. These services are collectively referred as Dependency engine. The architecture of Dependency engine is described here and an example is provided in Appendix. The source code of Dependency engine is available under permissive open source licenses, and has been initially stored into the project internal TULEAP source code repository.

This document by the OpenReq project is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Unported License.

This document has been produced in the context of the OpenReq Project. The OpenReq project is part of the European Community's H2020 Programme and is as such funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability is respect of this document, which is merely representing the authors view.

TABLE OF CONTENTS

1 Introduction 5

2 OpenReq context of Dependency engine 7

3 Systems context and behavior 9

4 Key concepts of Dependency engine 10

5 Logical view of Dependency engine 11

5.1 Milla 11

5.2 Mulperi 12

5.3 SpringCaaS 13

6 Behavior of Dependency engine 15

7 Development views 17

7.1 Milla development view 17

7.2 Mulperi development view 18

7.3 SpringCaaS Development view 20

8 Dependency engine APIs 21

Appendix 1: MulSON 24

Appendix 2: Map application example 25

Use cases of MapExample 26

Requirements in textual form 27

MulSON data exchange 28

Feature diagram 32

Kumbang feature model 33

Appendix 3: Demo user interface 37

Appendix 4: Glossary 39

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

LIST OF FIGURES

Figure 1. The context of Dependency engine in the OpenReq project 7
Figure 2. The systems in the context of Dependency engine 9
Figure 3. Logical view of Dependency engine 11
Figure 4. Milla can rovide integration with different RMS 12
Figure 5. The logical view of Mulperi 13
Figure 6. Sequence diagram of a basic scenario 16
Figure 7. Milla development view 17
Figure 8. Mulperi development view 19
Figure 9. SpringCaaS development view 20

Figure A1. Use cases of MapExample illustrating the core functionality and use 26
Figure A2. Tabulated textual requirements 27
Figure A3. A MulSON representation of MapExample 31
Figure A4. A feature diagram of MapExample 32
Figure A5. A generated Kumbang feature model representation of MapExample 36
Figure A6. A screenshot of the demo user interface 38

1 Introduction

OpenReq is a project that aims to enhance requirements engineering activities. The focus areas cover activities over the entire requirements engineering life-cycle, including the areas of requirements identification, classification and decision making support. The improvements we are looking for can be achieved through improved processes, methods and tools.

The work package WP5 “Requirements knowledge and dependency management” of OpenReq focuses on the phases when requirements have been elicited and even preliminarily analyzed in order to assess their validity and improve their quality. The requirements are treated as a whole, covering the different kinds of relations between requirements including even references that some of the requirements are so close to each other that they can be considered similar or even duplicates. We collectively refer any such relation between requirements as *interdependency.* We use the term interdependency to emphasize that we currently focus only on requirement level artifacts rather than more general dependencies or traceability between requirements and other artifacts. Interdependencies need to be taken into account especially in requirements prioritization and product management so that requirements are not considered only as isolated and singular entities. Interdependencies also affect the management decisions, such as the order of implementing the requirements. We specifically focus on release management in which requirements are assigned to be implemented in certain order, within a discrete release intervals and by a certain point of time.

This document describes the version 1.0 of the *Dependency engine*[1] of the OpenReq project. Dependency engine is used here to refer to the set of services implementing the requirements knowledge representation and dependency management. The services in Dependency engine are independent but collaborate in an orchestrated manner and shall provide microservice RESTful interfaces to the other services of OpenReq. This document focuses on Dependency engine architecture although it also briefly describes the detailed design decisions and general context including the other, interconnected systems. Only the version 1.0 of Dependency engine is covered that shall be the minimal viable product. This version is based on implementing most features described in the grant agreement task descriptions and requirements from other work packages by the end of August 2017.

Dependency engine operates primarily with, and in the context defined by the state-of-the-practice large-scale requirements management systems (RMSs). An example of dedicated RMS is Doors but issues trackers, such as Jira, are also used especially in large scale open source projects. Such RMS document and manage the requirements of a system under development. Essentially, a requirement in the RMS consists of: a unique ID; a phrase, figure or something to describe its content; properties or meta-data; and interdependencies to other requirements as a special class of properties.

[1] For more general approach for and architecture of Dependency engine beyond version 1.0, see “D5.1 OpenReq Approach for Requirements Knowledge and Dependency Management”

The objective of Dependency engine is to provide an advanced means to manage interdependencies between the requirements in the RMSs of a large-scale project. In so doing, Dependency engine automatically constructs and manages a model of the requirements on the basis of individual requirements that already have interdependencies expressed in their description or properties. The term *manage* means taking care of the consistency and other holistic properties of the entire model and to enable various analyses. Specifically, Dependency engine manages the interdependencies that should be taken into account in release management. Therefore, Dependency engine, on the one hand, interfaces with a RMS, and, on the other hand, provides an interface for querying about the structure, such as what requirements are interdependent on each other. However, Dependency engine is not a modeling tool, rather, it is a model management tool. It is possible to change relationships, e.g., on the basis of analyses, but changes are not in the core of functionality covered in Dependency engine.

The source code of the dependency engine is available under the open source licenses, and is initially stored into the project internal TULEAP source code repository.

Some parts use EPL (Eclipse Public License) and other parts use the BSD 3-clause license. These licenses are compliant with each other. In addition, the source code uses open source software that is compliant with these licenses such as Choco solver under the BSD license.

Related OpenReq deliverables:

- D5.1. A more general overview of planned functionality of dependency engine beyond version 1.0 and a more detailed description of the approach in general. Both D5.1 and D5.2 are planned as self-describing documents, hence some similarities or even overlapping among them in some sections.
- D1.4. A general overview of the OpenReq infrastructure beyond WP5 and a more detailed description of infrastructure, such as Swagger.

2 OpenReq context of Dependency engine

The context diagram in Figure 1 describes Dependency engine as a black box service and represents the OpenReq project context around Dependency engine.

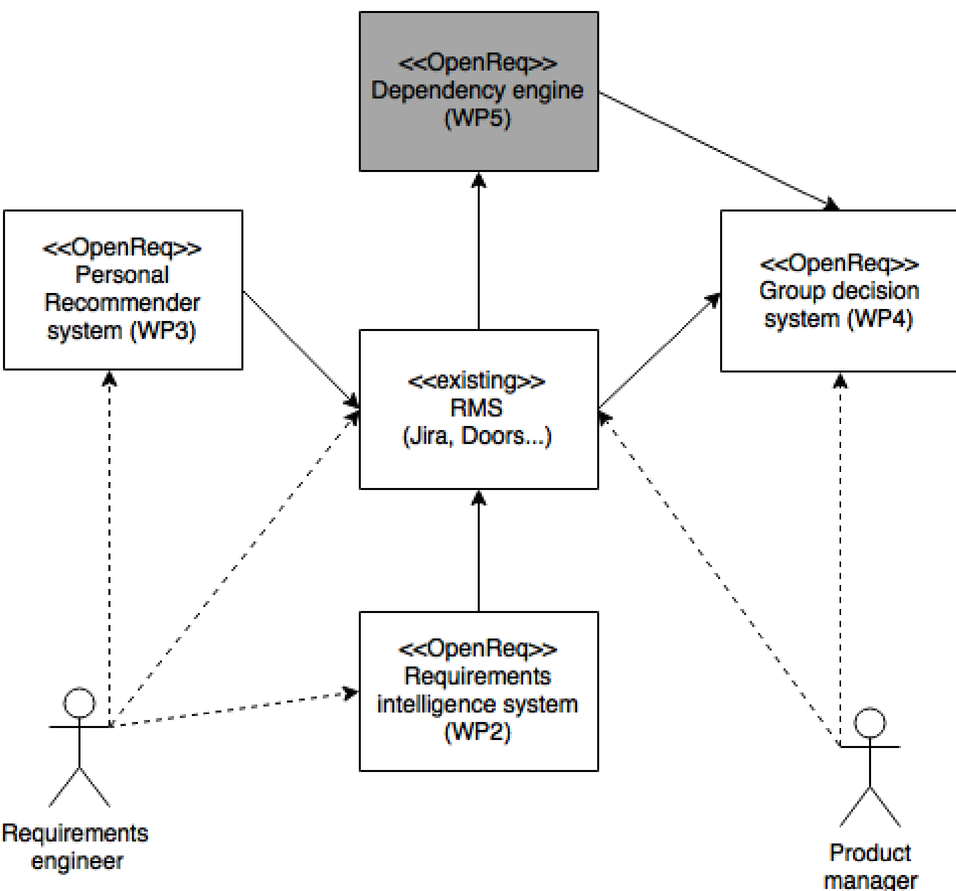

Figure 1. The context of Dependency engine in the OpenReq project
The solid lines depict data flow and the dashed lines depict user interaction.

Dependency engine operates in the context with other services and systems with functionality and purpose from Dependency engine viewpoint as follows:

- ***Personal recommender system*** (WP3) assists in requirements analysis activities (such as define, reuse, screen, understand, evaluate, quality assure), partially automating some of the tasks. Thus, it can change the content, formulation, interdependencies and any other properties of a requirement. It is also possible to change related requirements so that similar or related requirements are merged or regrouped thus affecting Dependency engine or utilizing the interdependency knowledge.
- ***Group decision system*** (WP4) is used, e.g., in release management to make decisions about what requirements are to be implemented and in which release. Group decision system then interacts with Dependency engine to find out interdependencies between requirements and the properties of requirements.
- Existing ***RMS*** (*Requirements management system)*. OpenReq project relies on existing RMS and other tools. The notable examples of state of the practice systems as RMS are

Jira and Doors. Dependency engine then integrates with a compatible RMS so that a RMS is the primary storage of requirements information. There can be other existing tools, such as Eclipse through which the integration can be done. The integration is not in the scope of Dependency engine per se, but rather in the scope of the trials of OpenReq.

- ***Requirements intelligence system*** (WP2) operates in the requirements elicitation phase analyzing explicit feedback, such as facebook, twitter or blogs and analyzing implicit feedback, such as usage of software though monitoring. Dependency engine does not directly interact with Requirements intelligence but relies on the requirements stored in the RMS.

There can also be ***an OpenReq database or repository*** that Personal recommender system (WP3) and Requirements intelligence (WP2) use to store the requirements. The OpenReq database has then parallel and equal functionality comparable to a RMS, i.e., the OpenReq database has from the point of view of Dependency engine similar functionality to store and manage requirements and their properties and metadata as RMS. Therefore, we do not differentiate here the potential OpenReq database but presume that there is one primary storage place referred to as a RMS. That is, a RMS refers hereafter to either a dedicated, existing RMS or an OpenReq specific storage.

Dependency engine has no direct (human) user interaction other than for IT system administration tasks and potentially a possibility to trigger and demonstrate functionality. There are two stakeholder roles that indirectly interact with Dependency engine:

- A ***requirements engineer*** is responsible for eliciting, analyzing, and managing the requirements, including the properties and interdependencies of the requirements. In particular, the requirements engineer is responsible for the changes. This task is assisted by Personal recommender system (WP3) and Requirements intelligence system (WP2), and the requirements are stored in a RMS.
- A ***product manager*** makes decisions about what requirements are implemented. A release manager is, in fact, a group of people or she interacts with a group of people. Therefore, the actions can include, e.g., voting or other negotiations for which Group decision engine (WP4) is developed for. Release manager also uses the existing RMS where the requirements are stored.

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

3 Systems context and behavior

Dependency engine operates in a context of other systems, namely a RMS and release management system, or product management system in general as shown in Figure 2. The systems are logically different from the point of view of Dependency engine and, therefore, treated as two systems. In practice, Jira, for example, can take the role of the both systems.

Figure 2 also depicts the key activities for which the numbers indicate the logical order during requirements engineering and product management.

1. Requirements are specified and stored in an existing RMS during the requirements engineering phase.
2. Requirements are synchronized to Dependency engine automatically and a model including interdependencies is generated in order to enable analyses.
3. Release management makes decisions about the product and releases.
   a. By selecting and negotiating requirements stored in the RMS.
   b. By (automatically) checking interdependencies of the selected requirements from Dependency engine.

Consequently, Dependency engine takes care of two main tasks. First, on the basis of requirements in RMS, Dependency engine generates a model of requirements that is a declarative knowledge representation that allows carrying out inferences on interdependencies utilizing existing inference techniques and tools. Second, on the basis of constructed model, Dependency engine allows, e.g., a release management system to query and reason about interdependencies, such as checking which other requirements are interdependent with the selected requirements and checking if the interdependencies of the selected requirements conflict with each other. The stakeholder roles are the same as described above.

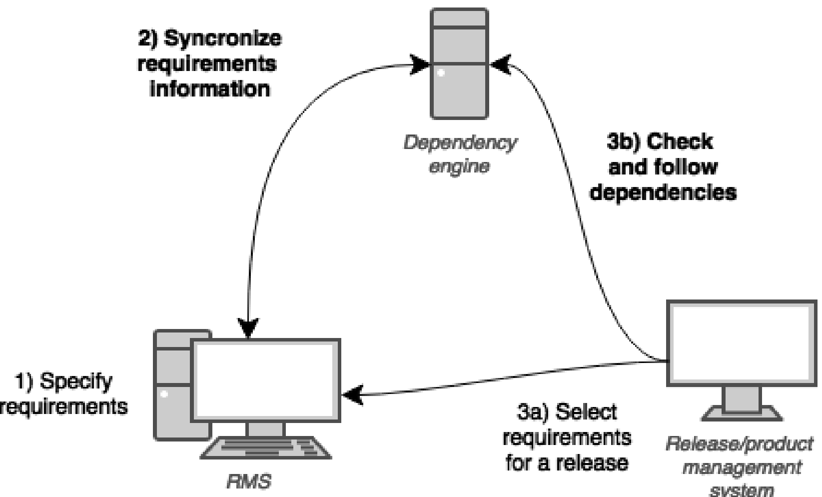

Figure 2. The systems in the context of Dependency engine

4 Key concepts of Dependency engine

We briefly summarize the key concepts that are applied in Dependency engine below.

A requirement is considered to be an entity that we refer to be so called “roadmappable”; a requirement is an entity that shall or shall not be realized at a certain point of time. Dependency engine treats the content (or body text) of a requirement ignorantly, in a black-box manner. Each requirement may have properties. First, some of the properties are about the requirement itself such as priority, release, effort or assignee. These properties are essentially attribute-value pairs in which the value can be, e.g., a number, enumerated value or free form text. Second, some of the properties are about the interdependencies of a requirement to another requirement. More specifically, a more general requirement can consist of a set of more detailed requirements that we refer to as “part-of” hierarchy. For example, an epic can consist of user stories or a user requirement can consist of a set of technical requirements. Alternatively, a requirement can have other kinds of interdependencies to other requirements beyond part-of structure such as “requires” another requirement, or “conflicts with” another requirement.

A RMS focuses on managing each individual requirement, including their properties and interdependencies. However, the entire product or project is defined holistically by a set of requirements as roadmappable entities with properties, and interdependencies to each other that constitute a model. Such a model is similar to a feature model. Therefore, a corresponding feature model is constructed automatically by generating and using a 1-to-1 mapping from requirements to features. The rationale for constructing a feature model is that a feature model is a well-researched approach that is provided with various kinds of analysis as well as existing analyses and inference tooling. A feature model representation is also the basis for performing release planning, diagnosis and repair in general.

For example, release management includes then a configuration problem in a very simple form: For a selected set of requirements, find other requirements that need to be taken into account because of the interdependencies. To assist in solving this configuration problem using Dependency engine, the selected requirements in the release management are mapped to features similarly as in the case of RMS, and the configuration problem is solved in the generated feature model utilizing existing configuration algorithms, technologies and tools.

5 Logical view of Dependency engine

Dependency engine consists of several independent software services. These services collaborate in an orchestrated manner and provide together the functionality of Dependency engine. The services will be described in the following sections and summarized in Figure 3 below. The design paradigm adheres to microservices in the sense that the interfaces of the services are listening for input messages and respond to each input message with an output message. Each service is a Spring Boot service of its own. The exposed external interfaces follow the REST paradigm. The messages are based on HTTP messages. A message takes a payload in a textual format rather than, e.g., binary object. Dependency engine supports a number of input formats. However, a JSON specification called MulSON is currently the most dependable and human readable format as well as used during the development. Therefore, we refer to MulSON throughout this document as an exemplar or archetypical format. For further details about MulSON, see Appendix 1.

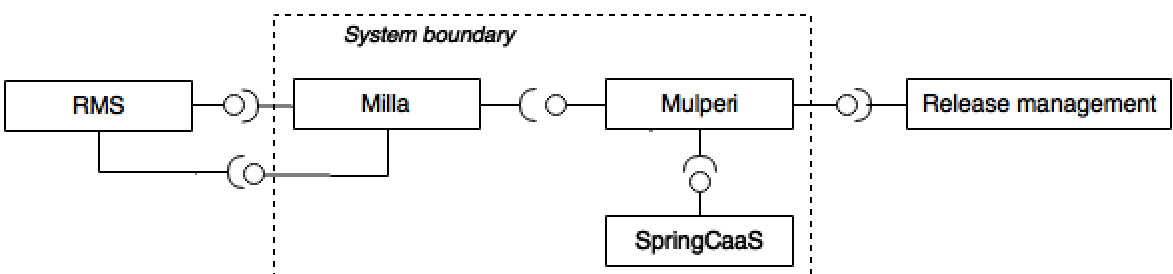

Figure 3. Logical view of Dependency engine
Consisting of three services, an external and existing RMS and release management.

5.1 Milla

Milla is the active orchestrator service for RMS and Dependency engine, but concerned only about the functionality to construct a feature model representation from requirements and not about the analyses or queries for release management (Figure 4 below). Another main role of Milla is to operate as a placeholder for additional, auxiliary functionality in order to keep the other services of Dependency engine, especially Mulperi, simple and not to introduce too much functionality in them, or not to introduce too many microservices having similar functionality.

Milla can retrieve requirements from a RMS and send the retrieved requirements further to Mulperi. Currently, an integration to Jira is implemented by accessing Jira's Query API. Alternatively, Milla can operate as a pipe-and-filter style service that accepts messages in different formats, converts the messages to proper format and further forwards them to Mulperi. We have experimented with ReqIF and MulSON but MulSON is a more dependable format. Consequently, Milla exposes the interfaces for both directions (provided and required) outside of dependency engine system boundary to retrieve or receive requirements. The output messages of Milla contain requirements in MulSON that are sent to Mulperi for further processing.

An example of auxiliary functionality of Milla is that it can be used for testing purposes. For example, the current functional testing and demonstration graphical user interface is implemented in Milla which will be replaced by OpenReq user interface in another service.

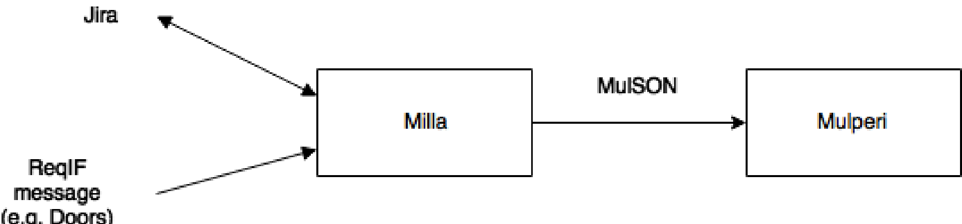

Figure 4. Milla can rovide integration with different RMS
Either by fetching or receiving requirements in different formats such as by integration with Jira's query interface or receiving ReqIF messages from different RMS. The requirements are formatted as MulSON and sent to Mulperi.

5.2 Mulperi

Mulperi (Figure 5) is a service taking care of the functionality for constructing a model from requirements that explicates the knowledge about interdependencies. Therefore, Mulperi also provides interfaces for the operations to the resulting model, such as in the case of querying about interdependencies in release management. The model is constructed utilizing feature modeling concepts and tools. However, the feature model is used mainly as an internal representation of Mulperi and in the internal interaction with SpringCaaS but interfaces to Mulperi in the broader context of Dependency engine operate on a basis of requirements in JSON messages.

The model is constructed from the requirements that Mulperi receives from a RMS through Milla. Each requirement corresponds with a feature of a feature model, the properties of a requirement correspond with the attributes of a feature, and relationships between requirements correspond with relationships in the feature model. In particular, the "part-of" relationships of requirements constitute the tree-hierarchy of a feature model. The resulting feature model is expressed in Kumbang language and send to SpringCaaS embedded in an XML message. For further background and details about adopted feature model technology, see D5.1.

Mulperi is not completely stateless, though. First, retrieving requirements and constructing a model upon each request would be too time consuming and could have performance issues. Therefore Mulperi stores the feature model representation of requirements in an internal data structure in order that analyses can be carried out without the need to always retrieve the requirements data from RMS. In particular, Mulperi stores the IDs of requirements as well as generates a unique ID for the entire model for valid referencing. Second, analyses for the entire model, such as checking requirements consistency, can be carried out as a batch process. Third, several different product management queries and reasonings can be made to the same model without a need to change or retrieve the model between queries.

The second main task of Mulperi is to facilitate various kinds of analyses and queries for the requirements, e.g., by requirements manager, product manager or release manager. By so doing, Mulperi exposes an interface for querying about the requirements such as it can be queried about direct consequences, or completeness of a release. For example, if a requirement A requires requirement B and requirement C is optional, selecting requirement would result in proposing to

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

select requirement B because it is a direct consequence. However, the selection is not complete because requirement C cannot be deduced being included or excluded. At the technical level, Mulperi exposes JSON-based REST interfaces through which queries using the requirements can be made. The queries are converted to XML format and for the inference required by queries Mulperi utilises SpringCaaS.

Consequently, logically two different interfaces are realized in one service. Mulperi exposes an interface, on the one hand, to receive the requirements from a RMS that are then used to construct a consistent feature model representation of requirements to explicate interdependencies. On the other hand, Mulperi exposes an interface for querying about the requirements taking into account interdependencies for the purposes, such as release management of OpenReq WP4. The interfaces are realized in one service because some data, such as the model and requirement IDs are cached locally for performance reasons. Mulperi does not change any information in the requirements, and, therefore, there is no need to update the RMS.

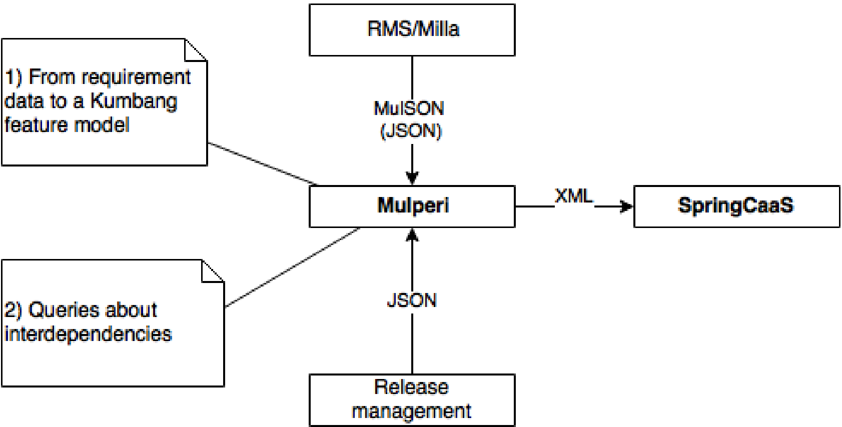

Figure 5. The logical view of Mulperi
Providing two logically different interfaces and utilization of SpringCaaS in the context of Dependency engine.

5.3 SpringCaaS

SpringCaaS (Spring boot Configurator as a Service) takes care of the required inference in Dependency engine. SpringCaaS takes as an input a feature model expressed using Kumbang language and embedded in an XML message. SpringCaaS constructs a declarative knowledge representation of the feature model as a Constraint Satisfaction Problem (CSP). SpringCaaS is an improved version of CaaS[2] realizing a configuration as a service paradigm. While the basic approach has remained similar, there have been technological revisions done for SpringCaaS. For example, SpringCaaS uses Choco as the inference engine in order to allow different kinds of

[2] Varvana Myllärniemi, Mikko Ylikangas, Mikko Raatikainen, Jari Pääkkö, Tomi Männistö, and Timo Aaltonen. 2012. Configurator-as-a-service: tool support for deriving software architectures at runtime. In *Proceedings of the WICSA/ECSA 2012 Companion Volume* (WICSA/ECSA '12). pp. 151-158. DOI=http://dx.doi.org/10.1145/2361999.2362031

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

analysis as well as for a more permissive license. The query interface has also been extended to support additional use cases for release management purposes.

SpringCaaS realizes twofold phases similarly as Mulperi but operates with the concepts of Kumbang feature model. First, SpringCaaS constructs a CSP program on the basis of a feature model. During the construction, SpringCaaS also analyses the resulting model for consistency. Second, SpringCaaS provides an interface for querying about the feature model such as direct consequences, and finding a consistent configuration.

In general, SpringCaaS is hidden behind Mulperi in the case of Dependency engine, but because each of the services are independent, SpringCaaS can also be used directly. In the case of OpenReq, it is more convenient to use SpringCaaS through Mulperi rather than directly. Therefore, we do not specify interfaces in more detail but presume that SpringCaaS is used through Mulperi.

Basically, SpringCaaS supports also Answer Set Programming (ASP) using “Smodels” engine but due to licensing issues arising from GPL, the support is currently discontinued. The current implementation does not include Smodels, and all functionalities do not work with Smodels as the inference engine.

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

6 Behavior of Dependency engine

The behavior of Dependency engine is illustrated using a scenario and respective sequence diagram below.

The scenario is loosely based on the Qt trial of OpenReq to make scenario concrete although Dependency engine per se is a general purpose service. The necessary Jira plugins are not a part of Dependency engine.

The scenario is:

> *A requirements manager specifies the requirements in Jira that is integrated with Dependency engine using a specific Jira plugin. Release manager makes decisions on releases by selecting the requirements 'A' and 'B' to a release in Jira. Dependency engine advises the release manager about a requires-interdependency found from the set of planned requirements for the release that results in a proposal to add another requirement 'C' to the release.*

The concrete steps are shown in a sequence diagram (Figure 6) and described below. The sequence is divided into two sequences.

The upper sequence in Figure 6 describes the scenario of a requirements engineer.

1. The requirement engineers store requirements in Jira. They analyze the requirements in order to assure the quality of requirements including defining the interdependencies.
2. The requirements engineers inform Milla about new requirements so that the Jira plugin sends a notification message.
3. Milla fetches all requirements of a project from Jira using Jira's query API by a REST call and receives the requirements in a response message as JSON payload.
4. Milla parses requirements information to an internal format understandable by Mulperi.
5. Milla encapsulates the appropriately formatted requirements into a message that it sends to Mulperi.
6. Mulperi constructs a feature model representation in Kumbang language from the requirements.
7. Mulperi sends the resulting Kumbang model in a XML message to SpringCaas.
8. SpringCaaS generates a CSP from the feature model. At the same time, SpringCaaS also carries out certain analyses such as checks consistency.

The lower part of the Figure 6 describes the basic scenario for release manager making queries about interdependencies.

1. Product manager makes a decision that the requirements A and B should be in a release
2. Jira plugin notices new release decision and makes a query for Dependency engine Mulperi service to resolve any interdependencies as direct consequences.
3. Mulperi converts the query to Kumbang XML format understandable by SpringCaaS.
4. Mulperi send the XML-based message to SpringCaaS
5. SpringCaaS reads the message and calculates interdependencies as a transitive closure using Choco solver.
6. SpringCaaS returns interdependencies as an XML-based return message.

7. Mulperi returns the message to the Jira plugin that made the original query as a JSON message.
8. The Jira plugin notifies the product manager about the found interdependency to the requirement 'C' that suggest including it to the release.

Figure 6. Sequence diagram of a basic scenario

7 Development views

The development view of each service shows the key classes, methods and variables.

7.1 Milla development view

Milla has a controller only for model management purposes, not for querying about the model. Models reuse existing data models and need to be generated on the basis on the specific integration. The figure below illustrates the integration with Jira and MulSON.

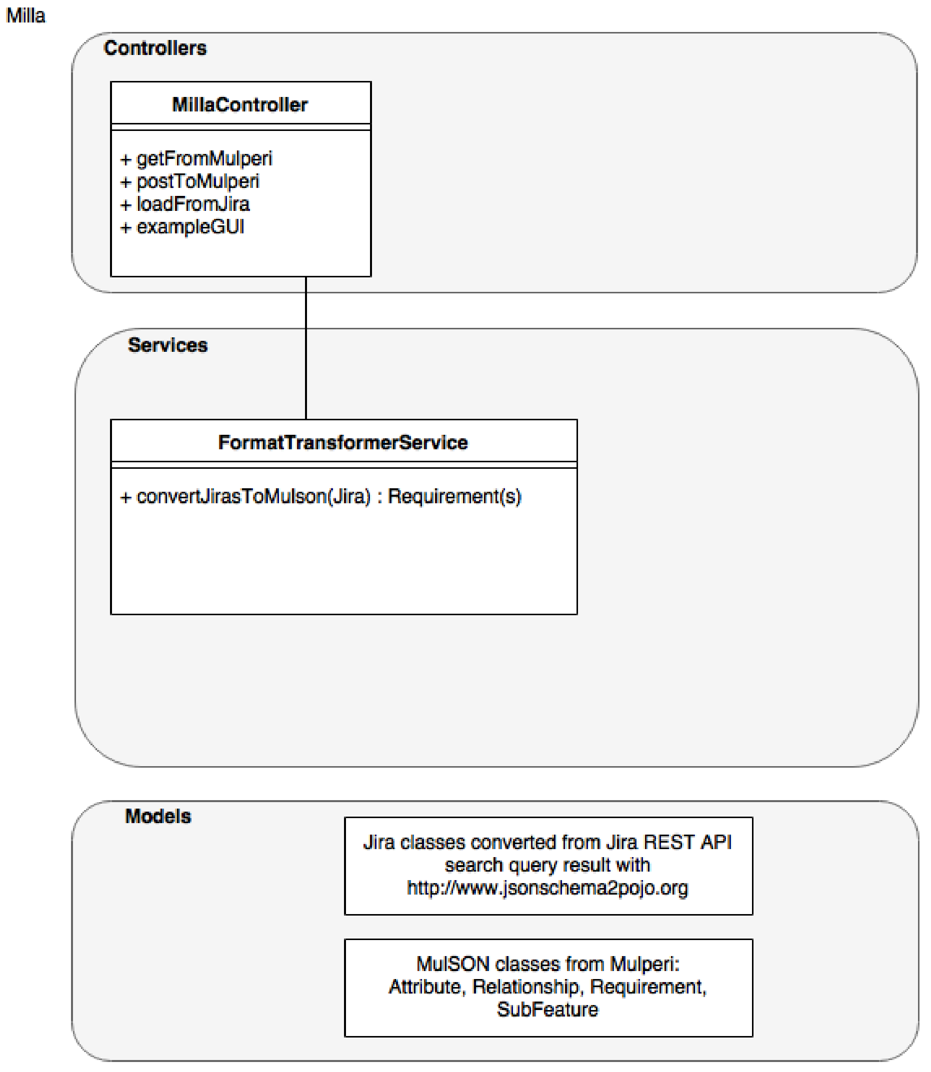

Figure 7. Milla development view

7.2 Mulperi development view

Figure 8 shows the development view of Mulperi by a class diagram. There are two controllers taking care of request on the external interfaces that then call internal services to implement the required functionality. Models describe the data stored in Mulperi.

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

Mulperi

Controllers

ModelController
- caasAddress: String
+ mulson
+ reqif
+ getModel

ConfigurationController
- caasAddress: String
+ requestConfiguration
+ isConsistent
+ findDirectConsequences

Services

ReqifParser
+ parse(String)

KumbangModelGenerator
+ generateKumbangModelString (Par:

CaasClient
+ uploadConfigurationModel
+ getConfiguration

FormatTransformerService
+ mulsonToKumbang
+ reqifToKumbang
+ featuresToConfigurationRequest
+ xmlToFeatures

Utils
+ diffFeatures

Repositories

ParsedModelRepository
(+ many inherited methods)
+ findFirstByModelName

Models

models.reqif

SpecRelation
- id: String
- type: String
- source: SpecObject
- target: SpecObject

source
target
1 parent to *

SpecObject
- id: String
+ getForeignId: String
+ getRequires: List<SpecObject>
+ getCardinality: String

Attribute
- id: String
- type: String
- value: String

models.mulson

Attribute
- name: String
- values: List<String>
- defaultValue: String

Requirement
- requirementId: String
- name: String
+ hasRelationshipType(String): bool
+ getParent: String
+ getRequires: List<String>
+ getCardinality: String

Relationship
- targetId: String
- type: String

models.kumbang

Attribute
- type: String
- role: String
- values: List<String>
- defaultValue: String

Feature
- type: String
- name: String

SubFeature
- types: List<String>
- role: String
- cardinality: String
+ toString

Constraint
- ifPresent: String
- thenRequired: String
- negated: boolean
+ toString

ParsedModel
- modelName: String
- comment: String

Component
- componentType: String

models.selections

AttributeSelection
- name: String
- value: String

1 has subfeatures *

FeatureSelection
- type: String
- name: String
- isSoft: boolean

Selections
- featureSelections: List<FeatureSel
- calculationConstraints: List<Calcul

CalculationConstraint
- attName: String
- operator: String
- value: String

Figure 8. Mulperi development view

7.3 SpringCaaS Development view

SpringCaaS combines the controller functionality to one class. The figure below illustrates the key classes of SpringCaaS service.

SpringCaaS

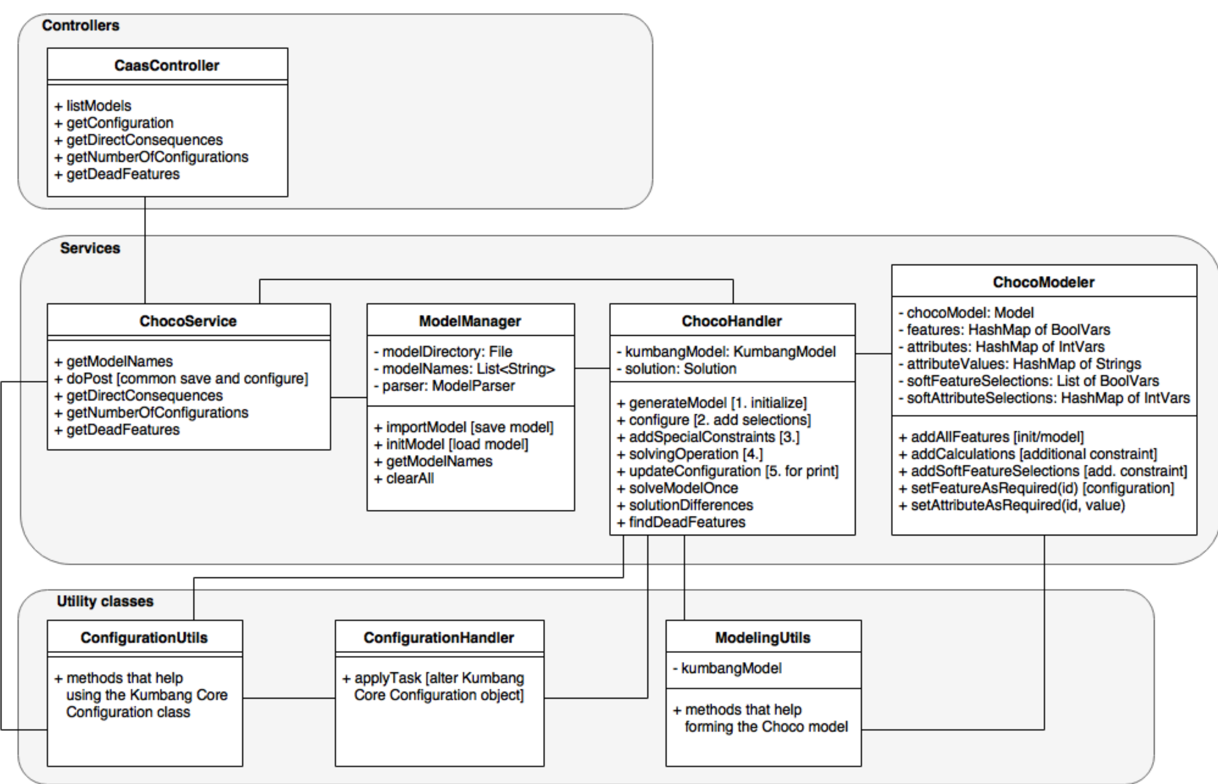

Figure 9. SpringCaaS development view

8 Dependency engine APIs

Dependency engine has principally two primary APIs for external use:

- Milla has an interface for synchronizing the requirements in a RMS.
- Mulperi has an interface for carrying out the queries.

The full documentation of these APIs is provided in Swagger[3] documentation. Respectively, other APIs are provided with a documentation using Swagger, even if they are meant internal to Dependency engine. Therefore, we only give a very simple example below. In fact, in the following we actually describe the direct use of Mulperi's API for simplicity rather than utilizing Milla. We apply directly a MulSON model that Milla would generate from different requirements formats and forward to Mulperi.

The example application ABCExample consists of requirements A, B and C, where A requires C. ABC is a general root. Each requirement can have properties, but for the purpose of demonstrating the API call we omit them here. See a more concrete example in Appendix 2. In MulSON, the requirements model is:

```
[
    {
      "requirementId": "ABC",
      "name": "ABCExample",
      "subfeatures": [
            {
                  "types": ["A"],
                  "role": "r1",
                  "cardinality": "0-1"
            },
            {
                  "types": ["B"],
                  "role": "r2",
                  "cardinality": "0-1"
            },
            {
                  "types": ["C"],
                  "role": "r3",
                  "cardinality": "0-1"
            }
      ]
    },
    {
      "requirementId": "A",
      "name": "a",
```

[3] For description of Swagger and its use, see OpenReq deliverable "D1.4 Project standards and infrastructure document"

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

```
        "relationships": [
              {
              "targetId": "r3",
              "type": "requires"
              }
        ]
    },
    {
        "requirementId": "B",
        "name": "b"
    },
    {
        "requirementId": "C",
        "name": "c"
    }
]
```


This model is posted to Mulperi (to simplify the call below, we do not repeat the above MulSON/JSON payload):

```
curl -X POST --header 'Content-Type: application/json' --header
'Accept: text/plain' -d '
```

The address for posting is, e.g., in development environment

```
http://localhost:8091/models/mulson
```

The successful response code of the HTTP is 201 and contains the ID of the model, in this case ID_1539060640.

When selecting the requirement A to a release, interdependencies can be be checked. This is done by posting the selection of A to Mulperi and asking consequences:

```
curl -X POST --header 'Content-Type: application/json' --header
'Accept: text/plain' -d '[ {"type": "A"} ]'
'http://localhost:8091/models/ID_1539060640/configurations/consequence
s'
```

Again, the payload is in JSON and the URL contains the model ID.

The successful HTTP response code is 200 and contains in its body the required requirements:

```
Consequences found successfully.

Features added in configuration:

A
ABC
C
```

Here C is added because the requires interdependency in A states that it should be selected. ABC is also selected because it is the root of the entire application even though it is not a requirement per se.

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

Appendices

The appendices give an informative explanation for the architecture of Dependency engine and a glossary of key terminology.

Appendix 1: MulSON

MulSON is an in-house developed simple, human readable JSON format for exchanging requirements, used especially for importing requirements to Mulperi (MulSON = Mulperi Submit Object Notation). The design of Dependency engine per se is not specific for MulSON and, e.g., a ReqIF parser already exists. MulSON is used in the development phase because of the simplicity and readability needs. Therefore, it is also used in our examples. MulSON is subject to change and the exact specification for Mulson shall be defined more precisely during the OpenReq project.

Some key design principles of MulSON

- “Part of” hierarchy is constructed by the subfeature concept and term.
- A parent requirement declares its child requirements
- Two level hierarchy: A grandparent cannot define its grandchildren. However, a grandparent can define its children who can further declare its children i.e. resulting in transitively the grandchildren of the grandparent.
- All requirements must be declared.
- Currently supported interdependencies other than above part-of are ‘requires’ and ‘incompatible’. Relationships are binary.
- Example:

```
[
   {
      "requirementId": "Req1ID",
      "name": "Req1Name",
      "subfeatures": [
      {
            "types": ["SubReqID"],
            "role": "SubReqrole",
            "cardinality": "0-1"
      },
      ]
   },
   {
      "requirementId": "SubReqID",
      "name": "SubReqName",
    } ]
```

Appendix 2: Map application example

A map application example (MapExample hereafter) is used to illustrate and concretize Dependency engine and the work done in OpenReq. The intention is to show how a set of use cases is represented as textual requirements and further using a feature model representation for which different kinds of analyses and interdependency queries can be carried out.

The premise of MapExample is that it is a new product development (NPD) project starting from the scratch meaning that the existing, already implemented features do not need to be considered.

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

Use cases of MapExample

MapExample consist of four general use cases as illustrated in Figure A1. The named lines between use cases illustrate interdependencies.

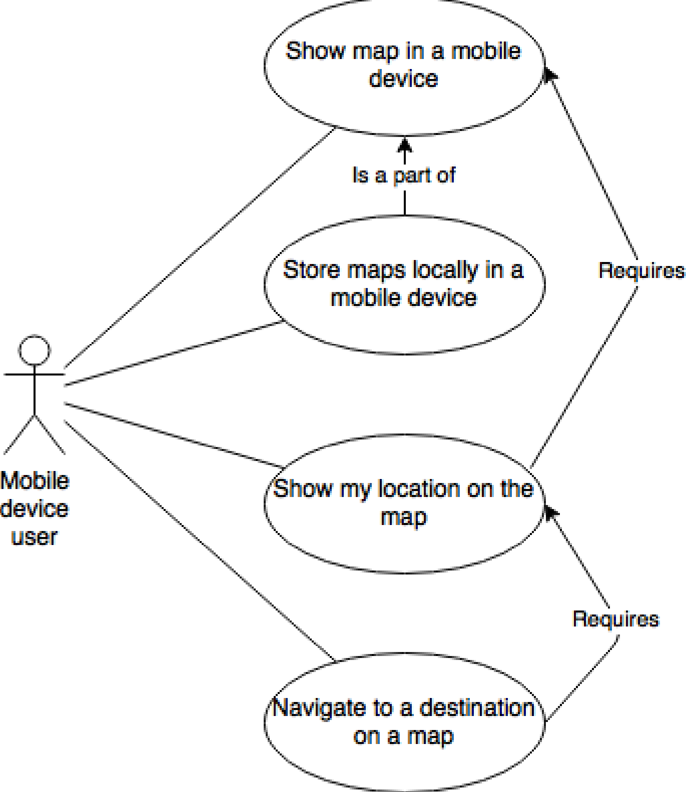

Figure A1. Use cases of MapExample illustrating the core functionality and use

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

Requirements in textual form

The requirements for MapExample are tabulated in Figure A2 below roughly corresponding to the use cases. Each requirement has an arbitrary string as a unique ID that has no semantics such as 'R1'. Full description is the actual requirement text. We also provide short names for the full descriptions although such short names are not actually necessary as Dependency engine relies on IDs. However, for the clarity of this example, it is sometimes more understandable to use these short names rather than IDs. Finally, the requirements have a set of properties. Some properties such as the target release are not defined but shall be defined during release planning. Interdependencies are shown in a dedicated column as a special property. We have refined Location requirement to two alternative technical requirements GPS and GLONASS that are non-exclusive.

ID	Full description	Name	Interdependency	Priority	Effort	Release
R1	Show map from a cloud service	ShowMap	Contains optional part of R1	1	10	-
R2	To store maps locally	CacheMap		2	5	-
R3	Show user location	Location	Requires R1 Alternative parts R3a or R3b.		-	-
R3a	Determine location using GPS	GPS		1	9	-
R3b	Determine location using GLONASS	GLONASS		2	8	-
R4	Navigate to a selected destination	Navigate	Requires R3	2	15	-

Figure A2. Tabulated textual requirements

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

MulSON data exchange

The requirements of map example are stored to and transferred within Dependency engine using a structured message format. Below is shown MulSON format. Note that we use in “name” the short names rather than full descriptions for presentation clarity. In practice, Mulperi relies only on IDs. We represent priority as a changeable property that has been assigned a value. Effort is a fixed property that cannot be changed but has the value as defined above. The release property can have values 1, 2 or 3, but none of the values are assigned as defined above.


```
[
   {
      "requirementId": "MapAppFeature",
      "name": "MapApplication",
      "subfeatures": [
      {
            "types": ["R1"],
            "role": "r1",
            "cardinality": "1-1"
      },
      {
            "types": ["R3"],
            "role": "r3",
            "cardinality": "0-1"
      },
      {
            "types": ["R4"],
            "role": "r4",
            "cardinality": "0-1"
      }
      ]
   },
   {
      "requirementId": "R1",
      "name": "ShowMap",
      "subfeatures": [
      {
            "types": ["R2"],
            "role": "r2",
            "cardinality": "0-1"
      }
      ],
      "attributes": [
      {
            "name": "Priority",
```


© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

```
            "values": ["1", "2", "3"],
            "defaultValue": "1"
      },
      {
            "name": "Release",
            "values": ["1", "2", "3"]
      },
      {
            "name": "Effort",
            "values": ["10"]
      }
      ]
   },
   {
      "requirementId": "R2",
      "name": "CacheMap",
      "attributes": [
      {
            "name": "Release",
            "values": ["1", "2", "3"]
      },
      {
            "name": "Effort",
            "values": ["5"]
      },
        {
            "name": "Priority",
            "values": ["1", "2", "3"],
            "defaultValue": "2"
      }
      ]
   },
   {
      "requirementId": "R3",
      "name": "Locations",
      "subfeatures": [
      {
            "types": ["R3a", "R3b"],
            "role": "r3_navitech",
            "cardinality": "1-2"
      }
      ],
      "attributes": [
      {
```

```
            "name": "Release",
            "values": ["1", "2", "3"]
      },
      {
            "name": "Priority",
            "values": ["1", "2", "3"]
      }
      ]
   },
   {
      "requirementId": "R3a",
      "name": "ShowLocation using GPS",
      "relationships": [
            {
            "targetId": "r1",
            "type": "requires"
            }
      ],
      "attributes": [
      {
            "name": "Release",
            "values": ["1", "2", "3"]
      },
      {
            "name": "Effort",
            "values": ["9"]
      },
      {
            "name": "Priority",
            "values": ["1", "2", "3"],
            "defaultValue": "1"
            }
      ]
   },
   {
      "requirementId": "R3b",
      "name": "ShowLocation using GLONASS",
      "relationships": [
            {
            "targetId": "r1",
            "type": "requires"
            }
      ],
      "attributes": [
```

```
      {
            "name": "Release",
            "values": ["1", "2", "3"]
      },
      {
            "name": "Effort",
            "values": ["8"]
      },
      {
            "name": "Priority",
            "values": ["1", "2", "3"],
            "defaultValue": "2"
      }
      ]
   },
   {
      "requirementId": "R4",
      "name": "Navigate",
      "relationships": [
            {
            "targetId": "r3_navitech",
            "type": "requires"
            }
      ],
      "attributes": [
      {
            "name": "Release",
            "values": ["1", "2", "3"]
      },
      {
            "name": "Effort",
            "values": ["15"]
      },
            {
            "name": "Priority",
            "values": ["1", "2", "3"],
              "defaultValue": "2"

            }
      ]
   }
]
```


Figure A3. A MulSON representation of MapExample

Feature diagram

A feature diagram is a graphical representation of a feature model. A feature diagram does not contain the properties, such as priority and release, but is an illustration for easier understanding and communication. Dependency engine does not in fact even construct such as feature diagram. The feature diagram of MapExample is shown in figure below.

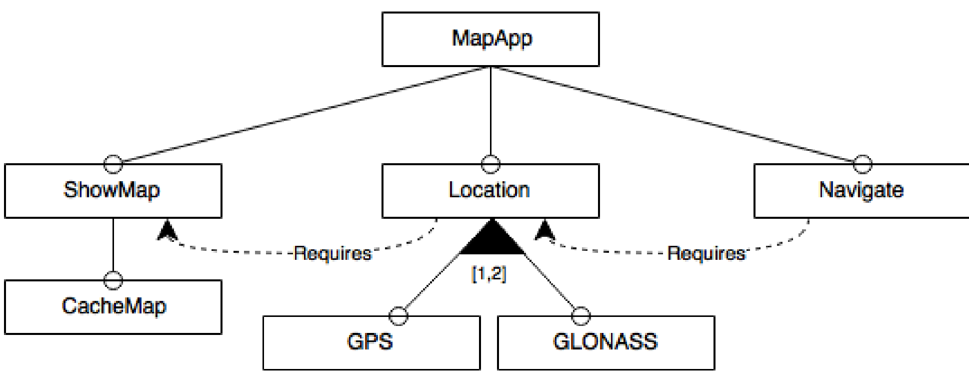

Figure A4. A feature diagram of MapExample

Kumbang feature model

Kumbang model describes the structure as defined by individual requirements (or features). Kumbang is generated from above mentioned MulSON.

```
Kumbang model ID1730205567
    root component ID1730205567
    root feature ID1730205567

//---components-----

component type ID1730205567 {
}

//---features-----

feature type ID1730205567 {
   subfeatures
         MapAppFeature MapAppFeature[0-1];
}

feature type MapAppFeature {
   subfeatures
         R1 r1[1-1];
         R3 r3[0-1];
         R4 r4[0-1];
}

feature type R1 {
   subfeatures
         R2 r2[0-1];
   attributes
         Priority Priority; //default = 1
         Release Release;
         Effort Effort;
}

feature type R2 {
   attributes
         Release2 Release;
         Effort2 Effort;
         Priority2 Priority; //default = 2
}
```

```
feature type R3 {
   subfeatures
          (R3a, R3b) r3_navitech[1-2] {different};
   attributes
          Release3 Release;
          Priority3 Priority;
}

feature type R3a {
   constraints
          present(r3_navitech) => present(r1);
   attributes
          Release4 Release;
          Effort3 Effort;
          Priority4 Priority; //default = 1
}

feature type R3b {
   constraints
          present(r3_navitech) => present(r1);
   attributes
          Release5 Release;
          Effort4 Effort;
          Priority5 Priority; //default = 2
}

feature type R4 {
   constraints
          present(r4) => present(r3_navitech);
   attributes
          Release6 Release;
          Effort5 Effort;
          Priority6 Priority; //default = 2
}

//---attributes-----

attribute type Priority = {
   1,
   2,
   3
}

attribute type Release = {
```

```
    1,
    2,
    3
}

attribute type Effort = {
    10
}

attribute type Release2 = {
    1,
    2,
    3
}

attribute type Effort2 = {
    5
}

attribute type Priority2 = {
    2,
    1,
    3
}

attribute type Release3 = {
    1,
    2,
    3
}

attribute type Priority3 = {
    1,
    2,
    3
}

attribute type Release4 = {
    1,
    2,
    3
}

attribute type Effort3 = {
```

```
   9
}

attribute type Priority4 = {
   1,
   2,
   3
}

attribute type Release5 = {
   1,
   2,
   3
}

attribute type Effort4 = {
   8
}

attribute type Priority5 = {
   2,
   1,
   3
}

attribute type Release6 = {
   1,
   2,
   3
}

attribute type Effort5 = {
   15
}

attribute type Priority6 = {
   2,
   1,
   3
}
```

Figure A5. A generated Kumbang feature model representation of MapExample

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

Appendix 3: Demo user interface

The demo user interface is a reference and test implementation of the Mulperi and Milla APIs -- Dependency engine per se is not supposed to have an user interface but user interaction takes place, e.g., in the dedicated OpenReq interface or plugins. The user interface is currently implemented in Milla's source code available in, e.g., http://localhost:9203/example/gui. It is written as a simple HTML page and a single vanilla Javascript file.

Under the “Generate model” section (upper part of Figure A6), different formats can be pasted into the Payload textarea and sent to a specific URL, for example Milla's Jira parser, Milla's MulSON relay or straight to Mulperi. After a successful model generation, the Model name in “Select requirements” is populated as per the API response (middle part of Figure A6).

When selecting the requirements (lower part of Figure A6), the first thing to do is to Get options that returns a hierarchy of the selectable requirements and their properties. One can then select any requirements and then press the Get configuration button that results in Milla calling Mulperi and further SpringCaaS. If the configuration is successful, all inferred requirements and properties are selected. In case of an error or an impossible configuration, an error message is displayed. One or more calculation constraints can be added by the Add sum constraint button. Autosubmit and autohide features enable to customize the behavior of the user interface that some may find convenient.

A video of the user interface in action with the Map Application and Qt Jira use cases can be found on the YouTube channel of Empirical Software Engineering research group of University of Helsinki. https://www.youtube.com/channel/UC5fXWiZVFSTTFe_U1EfpU-Q

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE

Generate model

Milla URL: localhost:9203/relay/mulson

[
  {
    "requirementId": "MapAppFeature",
    "name": "MapApplication",
    "subfeatures": [
      {
        "types": ["R1"],
        "role": "r1",
        "cardinality": "1-1"
      },
      {
        "types": ["R3"],
        "role": "r3",

Payload:

Submit

Select requirements

Mulperi URL: http://localhost:9202/

Model name: ID_821951394

Get options | Get configuration | Autosubmit on | Autohide on

Add sum constraint

Effort | < | 34

- ☑ root/ID_821951394
  - ☑ MapAppFeature/MapAppFeature
    - ☑ r1/R1
      - Priority: 1
      - Release: 1
      - Effort: 10
      - ☐ r2/R2
    - ☑ r3/R3
      - Release: 1
      - Priority: 1
      - ☐ r3_navitech/R3a
      - ☑ r3_navitech/R3b
        - Release: 1
        - Effort: 8
        - Priority: 1
    - ☑ r4/R4
      - Release: 1
      - Effort: 15
      - Priority: 1

Figure A6. A screenshot of the demo user interface

Appendix 4: Glossary

General terminology	
Configuration	An instance derived from a Kumbang model. Contains individual features (corresponding to requirements) arranged in the part-of hierarchy and properties of the individual features Configurations returned by SpringCaas are consistent.
(Configuration) Model	A representation of requirements/features and their relations to each other (parent/child, depends on, is incompatible with, etc).
Major software components / Modules	
Mulperi	A Java Spring server software that converts different input formats into 1) Kumbang models and 2) configuration or analysis requests - and forwards them both to CaaS
SpringCaaS (CaaS2017/CaaS)	Configurator-as-a-Service, processes Kumbang models and creates configurations using an inference engine. CaaS2017 is a newer version with Choco Solver support and SpringCaaS is a Spring Boot version of CaaS2017.
Choco Solver	An inference engine library component used by SpringCaaS. Supersedes Smodels. BSD licence. www.choco-solver.org
Smodels	Older inference engine used by CaaS, replaced by Choco Solver in newer versions. GPL 2 licence. ASP-based.
Milla	A service, whose purpose is to isolate development of system specific integrations from general purpose Mulperi and SpringCaaS service. Supports volatile APIs and converts them to MulSON for Mulperi. A relay software that can be used to alter data before sending it to Mulperi.
File formats	
MulSON	Mulperi Submit Object Notation, Mulperi's native requirement input format, in JSON
ReqIF	Requirements Interchange Format, a standardized alternative to MulSON, in XML. Currently not fully supported.
Kumbang	A software product line/variability modeling conceptualization and language. The Dependency Engine applies features and related concepts of Kumbang to model and analyze Requirements.

© HITEC, TUGRAZ, ENG, UPC, VOGELLA, SIEMENS, UH, QT, WINDTRE